Journal of Artificial Intelligence Research 75 (2022) 1103-1137 Submitted 03/2022; published 11/2022

Asymmetric Action Abstractions for Planning in Real-Time Strategy Games

Rubens O. Moraes RUBENS.MORAES@UFV.BR
Departamento de Informática,
Universidade Federal de Viçosa, Brazil

Mario A. Nascimento MARIO.NASCIMENTO@UALBERTA.CA
Department of Computing Science
University of Alberta, Canada

Levi H. S. Lelis LEVI.LELIS@UALBERTA.CA
Alberta Machine Intelligence Institute (Amii)
Department of Computing Science
University of Alberta, Canada

Abstract

Action abstractions restrict the number of legal actions available for real-time planning in 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 which we call asymmetric action abstraction. 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. Two of our algorithms are adaptations of algorithms developed for searching in action-abstracted spaces, Portfolio Greedy Search and Stratified Strategy Selection, and the other two are adaptations of an algorithm developed for searching in unabstracted spaces, NaïveMCTS. An extensive set of experiments in a real-time strategy game shows that search algorithms using asymmetric abstractions are able to outperform all other search algorithms tested.

1. 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 action abstractions (Hawkin, Holte, & Szafron, 2011) to reduce the number of legal actions available to the player. In RTS games, player-actions are represented as a vector of unit-actions, where each entry in the vector represents an action for a unit the player controls. Action abstractions reduce the number of player legal actions by reducing the number of legal ac-

©2022 AI Access Foundation. All rights reserved.

tions 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 (2013) introduced a method for building action abstractions through a set of scripts. A script $\bar{\sigma}$ is a function mapping a game state and a unit to a unit-action. 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 $\mathcal{P}$. Since the actions of all units are restricted by the set of scripts, we call an action abstraction generated with Churchill and Buro's scheme a uniform action abstraction (or uniform abstraction for short).

Uniform abstractions were shown to be successful in practice, especially in large RTS games (Churchill & Buro, 2013). This is because RTS games can have very large action spaces, and the problem's real-time constraints often allow search algorithms to explore only a small fraction of all actions before deciding on which one to perform next—uniform abstractions allow algorithms to focus their search on actions deemed as promising by the set of scripts. The drawback of uniform abstractions is that they equally restrict the actions of all units, which might remove strong strategies from the set of strategies the search algorithm is able to consider. For example, depending on the time of the game, the player might benefit from considering a larger set of actions for the units involved in combat, while worker units do not need to consider actions other than those involved in collecting resources. Search algorithms employing uniform abstractions divide their search effort equally with all aspects of the game, independently whether the aspects are important or not.

We introduce an action abstraction scheme we call asymmetric action abstractions (or asymmetric abstractions for short). In contrast to uniform abstractions that restrict the number of actions of all units, asymmetric abstractions restrict the number of actions of only a subset of units. As a result, asymmetric abstractions allow the search effort to be distributed unevenly amongst the units, thus allowing the search algorithm to "pay more attention" to different aspects of game. Asymmetric abstractions also retain the uniform abstractions' feature of allowing the search algorithm to focus on actions deemed as promising by a set of scripts. This is because a number of units still have their action set reduced by the set of scripts. We hypothesize that algorithms searching in asymmetrically abstracted action spaces can derive stronger strategies than algorithms searching in either uniformly abstracted and unabstracted action spaces.

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) (Churchill & Buro, 2013), and Stratified Strategy Selection (SSS) (Lelis, 2017). 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) (Ontañón, 2013, 2017) to search in unabstracted spaces. In addition to the two variants of NaïveMCTS that search in asymmetrically abstracted action spaces, we introduce a NaïveMCTS baseline that, similarly to PGS and SSS, use uniform abstractions; we call this baseline A1N.

We evaluate our algorithms in μRTS (Ontañón, 2013), 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. Moreover, the game is much simpler than commercial video games, which allows us to evaluate different algorithms without the technical difficulties typical of commercial video games. Finally, the μRTS codebase[1] contains most of the current state-of-the-art search algorithms, including the systems used in μRTS competitions (Ontañón, Barriga, Silva, Moraes, & Lelis, 2018), thus facilitating our empirical evaluation. An extensive set of experiments shows that the algorithms that search in asymmetrically abstracted action spaces outperform, in terms of matches won, their counterparts that search in unabstracted and uniformly abstracted ones.

Although we present our abstraction schemes and search algorithms in the context of RTS games, our ideas and algorithms can also be applied in other scenarios. For example, a robotic system that 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 action spaces is at core of the agent of the commercial card game Prismata (Churchill & Buro, 2015). The idea we introduce of searching in asymmetrically abstracted action spaces might also be used in such card games to enhance the strength of their agent. For example, the agent could benefit from an algorithm that discovers finer plans for the "more important" cards.

This paper extends two conference publications (Moraes & Lelis, 2018; Moraes, Mariño, Lelis, & Nascimento, 2018a). Moraes and Lelis (2018) introduced asymmetric abstractions and the versions of PGS and SSS for searching in such trees, GAB and SAB, while Moraes et al. (2018a) introduced the versions of NaïveMCTS that use uniform and asymmetric abstractions, A1N, A2N, and A3N. In this paper we (i) provide a more detailed explanation of algorithms searching with asymmetric action abstractions, (ii) test empirically a larger set of strategies for generating such abstractions, and (iii) directly compare the methods introduced in both previous works in the domain of μRTS, a full-fledged RTS game. Previously, GAB and SAB had only been tested in simpler RTS combats, and they had not been compared with the asymmetric variants of NaïveMCTS. Our results show that the three best performing algorithms employ asymmetric action abstractions, thus supporting our hypothesis. Namely, A3N obtains the highest winning rate amongst all algorithms tested, with GAB and SAB being the second and third best performing algorithms, respectively.

This paper is organized as follows. In Section 2 we review works related to real-time planning in RTS games as well as works related to action abstractions in extensive-form games. In Section 3 we define RTS games as an extensive-form game and review search algorithms used for planning in unabstracted spaces. In Section 4, we describe the asymmetric abstractions we introduce in this paper. Section 5 introduces the four algorithms for searching in asymmetrically abstracted action spaces. Finally, we evaluate the introduced algorithms in Section 6 with an extensive set of experiments on μRTS.

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

2. Related Work

The action abstraction schema and the search algorithms we present in this paper are related to other search algorithms developed for RTS games. Our work is also related to learning-based systems for RTS games and to action abstraction schemes used in other contexts such as for approximating optimal solutions in extensive-form games such as Poker (Billings, Burch, Davidson, Holte, Schaeffer, Schauenberg, & Szafron, 2003) and for solving classical planning problems (Hoffmann & Nebel, 2001). We review each of these lines of research.

2.1 Search-Based Approaches for RTS Games

Before the invention of action abstractions induced by scripts, state-of-the-art algorithms included search methods for unabstracted spaces such as Monte Carlo (Chung, Buro, & Schaeffer, 2005; Sailer, Buro, & Lanctot, 2007; Balla & Fern, 2009; Ontañón, 2013) and Alpha-Beta (Churchill, Saffidine, & Buro, 2012). Due to the large number of actions available, Alpha-Beta and Monte Carlo methods tend to perform well only in small-scale games.

In addition to PGS (Churchill & Buro, 2013), Justesen et al. (2014) proposed two variations of Upper Confidence bounds applied to Trees (UCT) (Kocsis & Szepesvári, 2006) for searching in uniformly abstracted trees: script-based and cluster-based UCT. Wang et al. (2016) introduced Portfolio Online Evolution (POE) a local search algorithm also designed for uniformly abstracted trees. While Wang et al. showed empirically that POE is able to outperform Justesen's algorithms, Lelis (2017) showed that PGS and SSS are able to outperform POE. All these results were obtained in a testbed named SparCraft (Churchill & Buro, 2013). SparCraft is an efficient combat simulator for the combat scenarios that arise in the commercial game of StarCraft. While Justesen et al.'s and Wang et al.'s algorithms can be modified to also search in asymmetrically abstracted trees, we use PGS and SSS instead as those were shown to perform better in SparCraft (Lelis, 2017).

Moraes, Mariño, and Lelis (2018b) showed that PGS can suffer from a pathological issue that can cause it to find worse strategies if granted more computation time. The root of PGS's pathology is in its scheme for approximating a best response to the opponent's strategy if PGS also approximates a best response to the player's strategy. Moraes et al. (2018b) introduced Nested-Greedy Search (NGS), an algorithm that is similar to PGS, but that does not suffer from the pathology. They also showed that the pathology does not occur if one fixes the opponent's strategy while searching for a player-action with PGS.

Recently, Lelis (2020) presented a unifying perspective for several of the algorithms mentioned above through an algorithm named GEX (General Combinatorial Search for Exponential Spaces). In addition to A1N, A2N, and A3N, which are the variants of NaïveMCTS that we introduce in this paper, Lelis showed that PGS, SSS, POE, NGS, algorithms that search in uniformly abstracted spaces, can also be seen as a special case of GEX.

Scripts have also been used to guide the search by means other than action abstractions. Puppet Search (PS) (Barriga, Stanescu, & Buro, 2018) defines a search space over the parameter values of scripts. Strategy Tactics (STT) (Barriga, Stanescu, & Buro, 2017) combines PS's search in the script-parameter space with a NaïveMCTS search in the original state space for the combat units. Strategy Creation via Voting (SCV) generates scripts via voting (Silva, Moraes, Lelis, & Gal, 2019), which are then used to play the game. In contrast

with PS, STT, and SCV that generate novel scripts during the game, the algorithms we introduce in this paper use a set of scripts to generate action abstractions.

Yang and Ontañón (2019) introduced an algorithm named Guided Naïve Sampling (GNS) that also searches in a form of asymmetric action abstraction. Similarly to A3N, GNS is also based on NaïveMCTS. GNS evaluates an action returned by a script when first expanding a node in its MCTS search. In subsequent evaluations, GNS chooses an action to be evaluated according to Naïve Sampling with probability ϵ and chooses an action from a script, chosen uniformly at random, with probability $1 - \epsilon$. Similarly to A3N, when using Naïve Sampling, GNS considers actions outside the pool of those given by the uniform abstraction. The core difference between GNS and A3N is that the latter only considers actions that are not part of the uniform abstraction for a subset of units, while the former considers actions that are not part of the uniform abstraction for all units. GNS has the theoretical advantage of being guaranteed to converge to an optimal solution because it considers all actions, while A3N has the practical advantage of allowing the algorithm to focus on a subset of components during search. Our experiments show that A3N compares favorably against GNS.

2.2 Learning-Based Approaches for RTS Games

Another line of related research uses learning to control 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). Search algorithms tend to scale more easily to large-scale combat scenarios than these learning-based methods. While the former can effectively handle battles with more than 100 units (Churchill & Buro, 2013; Lelis, 2017), the latter are usually tested on battles with no more than 50 units.

An exception in terms of both scalability and strength is AlphaStar (Vinyals, Babuschkin, Czarnecki, Mathieu, Dudzik, Chung, Choi, Powell, Ewalds, Georgiev, Oh, Horgan, Kroiss, Danihelka, Huang, Sifre, Cai, Agapiou, Jaderberg, ..., & Silver, 2019), the first system able to defeat professional players in the commercial game of StarCraft II. Despite the noteworthy achievement, AlphaStar's approach might not be easily applicable in other scenarios. It requires a large set of matches played by humans to train an initial version of the system, which is then improved through self-training over the course of several days over a non-trivial and specialized computational infrastructure. Search-based approaches, such as the ones we investigate, require no training at all and can be run over commodity computing devices. The ideas we present in this paper are orthogonal to those presented in learning-based papers as search and learning may be combined into stronger agents.

2.3 Action Abstractions in Extensive-Form Games

Action abstractions have also been applied to imperfect-information games, mostly in computer Poker (Hawkin et al., 2011; Hawkin, Holte, & Szafron, 2012). Although simultaneous-moves games such as the ones we deal with in this paper can also be modeled as an imperfect-information extensive-form game (Bosanský, Lisý, Lanctot, Cermák, & Winands, 2016), the abstraction methods used to develop Poker playing agents differ from the approaches we consider and introduce in this paper. The action abstraction methods introduced in the

context of Poker assume function-based action schema, where one searches for a subset of parameter values in a function-based action. For example, the methods Hawkin et al. (2011, 2012) introduced choose the number of chips (parameter) to be considered in a betting action (function) in the game of Poker. The action spaces of RTS games we consider in this paper are much larger than those considered in Poker and one would have to substantially reduce the original space of an RTS game before it can be solved. As a result, the abstracted game would likely be radically different from the original one, which could render optimal strategies for the abstracted game too weak to be useful in the original game. Libratus, a system that defeated professional Poker players relied on domain-specific action abstractions created by analyzing the bet sizes used in different parts of the game by Poker agents that took part in the Annual Computer Poker competition (Brown & Sandholm, 2018). Therefore, Libratus' action abstraction scheme is not directly applicable to other games.

As an example outside computer Poker research, Sandholm and Singh (2012) introduced an approach to construct state and action abstractions with bounded loss. In their work, the goal is to find, in an offline procedure, an optimal strategy for the abstracted game so that it can be used in the original game. In this work, we only approximate a solution to the abstracted game while satisfying harsh real-time constraints.

2.4 Action Abstractions in Classical Planning

Action abstractions are also commonly used to solve classical planning problems. The FF planning system defines a set of helpful actions, which are the actions used to solve a relaxed version of a planning problem (Hoffmann & Nebel, 2001). FF then uses a hill-climbing algorithm restricted to the set of helpful actions to solve the original problem. The idea of considering only a subset of actions during planning is exactly the concept of action abstraction we consider in this paper. Fast Downward uses a concept similar to helpful actions, but with the name of preferred operators (Helmert, 2006). Richter and Helmert (2009) showed that preferred operators can improve the results in terms of number of problems solved and solution quality of different planning systems. While helpful actions and preferred operators were introduced and evaluated in the single-agent setting, in this work, we deal with the two-player setting. Also, while the action abstractions for classical planning are obtained by solving a relaxed version of the original planning problem, here we induce action abstractions by considering a set of simple strategies for playing the game.

3. Background

We define RTS games as an extensive-form game and introduce concepts needed in later sections. RTS games can be described as multi-unit zero-sum extensive-form games with simultaneous and durative actions (i.e., actions that require more than one time step to be completed), 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**. We assume that i is the player controlled by the search algorithms we describe and $-i$ is i's 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 $-i$. Every unit $u \in \mathcal{U}^s$ has properties that are specific to the game modeled. For example, a unit could include properties such as its x and y coordinates on the game's map, the unit's attack range, attack damage, and hit points.

- $s_{init} \in \mathcal{D}$ is the start state of the game.

- $\mathcal{A}(s) = \mathcal{A}_i(s) \times \mathcal{A}_{-i}(s)$ is the set of legal **joint actions** at state s. $\mathcal{A}_j(s)$ is the set of legal **actions** player j can perform at state s, with $j \in \{-i, i\}$. We write $\mathcal{A}$, $\mathcal{A}_i$, and $\mathcal{A}_{-i}$ instead of $\mathcal{A}(s)$, $\mathcal{A}_i(s)$, and $\mathcal{A}_{-i}(s)$ whenever the state s is clear from the context. Each action $a \in \mathcal{A}_j(s)$ is denoted by a vector of n_j **unit-actions** $(m_1, \cdots, m_{n_j})$, where $m_k \in a$ is the action of the k-th **ready unit** of player j. 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 player j as $\mathcal{U}^s_{j,r}$. For $k \in \mathbb{N}^+$ we use $a[k]$ to denote the action of the k-th ready unit. Also, for unit u, we use $a[u]$ to denote the action of u in a. We denote the set of unit-actions as $\mathcal{M}$, which is non-empty for ready units because a unit can always perform a "no action". We write $\mathcal{M}(s, u)$ to denote the set of legal actions of unit u at s.

- $\mathcal{B} : \mathcal{D} \rightarrow \mathcal{N}' \subseteq \mathcal{N}$ is a function that receives a state s and returns the subset of players with one or more ready units in s.

- $\mathcal{R}_i : \mathcal{F} \rightarrow \mathbb{R}$ is a **utility function** with $\mathcal{R}_i(s) = -\mathcal{R}_{-i}(s)$, for any $s \in \mathcal{F}$.

- The **transition function** $\mathcal{T} : \mathcal{S} \times \mathcal{A}_i \times \mathcal{A}_{-i} \rightarrow \mathcal{S}$ deterministically determines the successor state for a state s and the set of joint actions taken at s. Note that, since actions are durative, units might still be executing their unit-actions in a_i and a_{-i} in the state returned by $\mathcal{T}(s, a_i, a_{-i})$. If two or more unit-actions are conflicting (e.g., two units try to move to the same cell of the game's grid), then the transition function performs the action of only one of the units; the other units do not perform an action in that decision point of the game. The decision of which unit performs the action is arbitrary but deterministic. The transition function sometimes is referred to in the literature as the forward model.

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. 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. We can ensure that the trees are finite by determining a time limit to each game. Once the time limit is reached, all states become terminal. 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 maximize Ψ, while $-i$ tries to reach nodes that minimize Ψ. All Ψ functions we consider are play-out based. That is, given a state s to be evaluated, one simulates the game forward while following a fixed strategy for both players for a limited number of steps. Then, the state thus reached

in this simulation is evaluated according to a heuristic function and this evaluation is used as the estimated end-game value of s.

We call a **decision-point** of player j a state s in which j has at least one ready unit. In this paper, the search algorithm controlling the units of a player is invoked at every decision point n of the game tree.

A **player strategy** is a function $\sigma_i : \mathcal{S} \times \mathcal{A}_i \rightarrow [0, 1]$ for player i, which maps a state s and an action 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. A script $\bar{\sigma}$ also defines a player strategy. Since a script maps a state s and unit u to a unit-action for u to perform in s, a strategy can be defined if $\bar{\sigma}$ is used to define the unit-action of all units of a player. That way, a script defines a player-action to be performed in any state of the game.

The optimal **value of the game** rooted at s for player i is denoted as $V_i(s)$ and can be computed by finding a Nash Equilibrium profile for the players. Due to the problem's size and real-time constraints, it is impractical to find optimal profiles for most RTS games. State-of-the-art heuristic search approaches use abstractions to reduce the game tree size and then derive in real time player strategies from the abstracted trees.

3.1 Search Algorithms for Unabstracted Trees

In this section, we review two search algorithms used for planning in RTS games unabstracted trees: Alpha-Beta Considering Durations (ABCD) and Naïve Monte Carlo Tree Search (NaïveMCTS). We use ABCD and NaïveMCTS as the basis for the algorithms we introduce for searching in asymmetrically abstracted game trees.

3.1.1 Alpha-Beta Considering Durations (ABCD)

Minimax search with Alpha-Beta pruning (Knuth & Moore, 1975) (or Alpha-Beta for short) has been successfully applied to games such as Chess (Campbell, Hoane, & hsiung Hsu, 2002). Alpha-Beta computes the value of a game while pruning branches of the tree that are not reached in optimal play. As Knuth and Moore (1975) showed, Alpha-Beta can dramatically reduce the number of nodes expanded, compared to a regular minimax search.

Kovarsky and Buro (2005) 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 afterward. 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. Examples of policies π include random and alternate selections of who is to act first. Churchill et al. (2012) showed that Alpha-Beta with the alternate policy defeats the same algorithm with the random policy in RTS combats.

Next, we present Alpha-Beta's pseudocode for sequential-move games. Then, we discuss how the pseudocode is to be modified to account for Kovarsky and Buro's contributions.

Alpha-Beta, shown in Algorithm 1, receives as input a state as the root of the tree, a maximum depth d, bounds α and β, with the initial values of $-\infty$ and ∞, respectively, and an evaluation function Ψ. The value of d limits the depth in which Alpha-Beta's depth-first search will be performed. We use Algorithm 1 in an iterative-deepening manner, with the

Algorithm 1 Alpha-Beta

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

1: **if** $s \in \mathcal{F}$ or $d = 0$ **then**
2: **return** $\Psi(s)$
3: $j \leftarrow \mathcal{B}(s)$
4: **if** $j = -i$ **then**
5: $M \leftarrow \infty$
6: **for** each $a \in \mathcal{A}_{-i}(s)$ **do**
7: $M \leftarrow \min(\text{Alpha-Beta}(\mathcal{T}(s, a), d - 1, \alpha, \beta, \Psi), M)$
8: **if** $M \leq \alpha$ **then**
9: **return** M
10: $\beta = \min(\beta, M)$
11: **if** $j = i$ **then**
12: $M \leftarrow -\infty$
13: **for** each $a \in \mathcal{A}_i(s)$ **do**
14: $M \leftarrow \max(\text{Alpha-Beta}(\mathcal{T}(s, a), d - 1, \alpha, \beta, \Psi), M)$
15: **if** $M \geq \beta$ **then**
16: **return** M
17: $\alpha = \max(\alpha, M)$
18: **return** M

value of d passed as input to the 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 evaluation function Ψ estimates the minimax value of a state $s \in \mathcal{D}$; $\Psi(s) = \mathcal{R}_i(s)$, if $s \in \mathcal{F}$.

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 $-i$.

Alpha-Beta, as shown in Algorithm 1, assumes sequential-moves games. Simultaneous-moves games have states where players are able to act simultaneously (i.e., $\mathcal{B}(s) = \{i, -i\}$) and states where only one player acts (either $\mathcal{B}(s) = \{i\}$ or $\mathcal{B}(s) = \{-i\}$). Churchill

et al. (2012) introduced Alpha-Beta Considering Durations (ABCD), a version of Alpha-Beta that accounts for simultaneous-moves.

We need to perform two modifications in the pseudocode shown in Algorithm 1 to transform Alpha-Beta into ABCD. First, we replace the assignment $j \leftarrow \mathcal{B}(s)$ 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 $\mathcal{B}(s) = \{i\}$ or $\mathcal{B}(s) = \{-i\}$, then $\{\pi(s)\} = \mathcal{B}(s)$. If $\mathcal{B}(s) = \{i, -i\}$, 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. Since the game is artificially serialized by the search algorithm, in a state s that the players are to act simultaneously, the set of actions $\mathcal{A}_{-i}$ (resp. $\mathcal{A}_i$) should be computed before applying the action player i (resp. $-i$) is going to perform at s. 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. See the paper by Churchill et al. (2012) for a pseudocode of ABCD. The ABCD implementation we use in our experiments employs a transposition table to avoid expanding multiple paths leading to the same state (Atkin & Slate, 1988; Zobrist, 1990).

3.1.2 Naïve Monte Carlo Tree Search (NaïveMCTS)

Ontañón (2017) 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 (2017) 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:

Algorithm 2 NaïveMCTS

Input: State s, sampling strategies π_0, π_l and π_g, and evaluation function Ψ.
Output: Action a

```
1: root ← node(s)
2: while hasTime() do
3:     leaf ← SelectAndExpandNode(root, π0, πl, πg)
4:     v ← Ψ(leaf.state)
5:     propagateEvaluation(leaf, v)
6: return getMostVisitedAction(root)
```

- 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 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 (2017) 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 arbitrarily choosing which player-action to evaluate next as a vanilla implementation of a MCTS algorithm might do, NaïveMCTS uses NS to select player-actions composed of unit-actions that tend to yield good rewards. NaïveMCTS receives as input the current state s of the game and strategies π_0, π_l, and π_g, which are required by NS as discussed above. NaïveMCTS returns a player-action a for player i to be played at state s.

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). We denote the state that is represented by a node in the tree with the word "state" preceded by the name of the node and a dot (e.g., *root.state*). 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. The procedure returns a node that is added to the NaïveMCTS tree. 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). In contrast to ABCD, NaïveMCTS

Algorithm 3 SelectAndExpandNode

Input: A game tree node n_0 and sampling strategies π_0, π_l and π_g
Output: 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

ignores the possible delayed effects of actions, discussed in Section 3.1.1, caused by its artificial serialization of the game. The NS procedure is invoked to decide which player-action will be explored. That is, given current state $n_0.state$ and the player j to act at $n_0.state$, which is enforced by π to be either i or $-i$, but never both, NS returns a node n whose state $n.state$ is reached by applying a player-action (macro-arm) to $n_0.state$.

In line 3 the procedure checks if the node n returned by NS is already part of the MCTS tree (i.e., if n is amongst the children of n in the MCTS tree). Node n is part of the tree if NS chooses to exploit an existing macro-arm. In this case, SelectAndExpandNode is called recursively so that NS is invoked to select a player-action from n. If NS chooses to explore, then 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). Although unlikely due to the large number of different macro-arms, NS's policy might choose to explore and yet choose a node that is already in the MCTS tree. In its main procedure, now under the name of *leaf*, the newly added node is evaluated with a play-out based function Ψ, and the value v returned is used as the reward value for all macro-arms (player-actions) traversed from root of the MCTS tree to the leaf node added by SelectAndExpandNode. This is performed by procedure PropagateEvaluation (line 5), whose implementation is omitted in the interest of simplicity. Once NaïveMCTS 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, whose implementation is also omitted (line 6).

3.2 Search Algorithms for Uniformly Abstracted Trees

In this section we present Portfolio Greedy Search (PGS) and Stratified Strategy Selection (SSS), two algorithms developed for searching in uniformly abstracted trees. Before presenting PGS and SSS we discuss uniform abstractions generated by a set of scripts.

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 are returned by a script in $\mathcal{P}$, defined as,

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

Algorithm 4 Portfolio Greedy Search

Input: state s, default script $\bar{\sigma}_d$, set of scripts $\mathcal{P}$, time limit e, and evaluation function Ψ.
Output: action a for player i's units.

1: $\bar{\sigma}_i \leftarrow$ choose a script from $\mathcal{P}$ considering that $-i$ acts according to $\bar{\sigma}_d$ *//see text*
2: $\bar{\sigma}_{-i} \leftarrow$ choose a script from $\mathcal{P}$ considering that i acts according to $\bar{\sigma}_i$ *//see text*
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: **for** each $u \in \mathcal{U}^s_{i,r}$ **do**
6: **for** each $\bar{\sigma} \in \mathcal{P}$ **do**
7: $a'_i \leftarrow a_i$; $a'_i[u] \leftarrow \bar{\sigma}(s, u)$
8: **if** $\Psi(\mathcal{T}(s, a'_i, a_{-i})) > \Psi(\mathcal{T}(s, a_i, a_{-i}))$ **then**
9: $a_i \leftarrow a'_i$
10: **if** time elapsed is larger than e **then**
11: **return** a_i
12: **return** a_i

Definition 1 *A uniform abstraction Φ is a function that receives a state s, a player i, and a set of scripts $\mathcal{P}$, and it 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 search in a game tree in 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 unit-actions returned by the scripts in $\mathcal{P}$.

3.2.1 Portfolio Greedy Search

Churchill and Buro (2013) introduced Portfolio Greedy Search (PGS), a hill-climbing search procedure for uniformly abstracted trees. Algorithm 4 shows the pseudocode of PGS, which receives as input a state s, a default script $\bar{\sigma}_d$, a set of scripts $\mathcal{P}$, a time limit e, and an evaluation function Ψ. PGS returns an action vector a for player i to be executed in s.

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). Churchill and Buro (2013) called this initialization procedure the "seeding" of the players' actions. 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 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 (lines 11 and 12).

The action vector a_{-i} remains unchanged after its initialization (line 4). Although in its original formulation PGS alternates between improving player i's and player $-i$'s actions (Churchill & Buro, 2013), in their experiments, Churchill and Buro allow PGS to improve only player i's action while player $-i$'s is fixed. Moraes et al. (2018b) showed that, if set to improve both a_i and a_{-i}, PGS can suffer from the pathological "non-convergence problem". Due to this 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 order to overcome that issue, Moraes et al. introduced Nested-Greedy Search (NGS), an algorithm 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 to large games (Moraes et al., 2018b).

In addition to fixing the opponent's action during search, another difference between our implementation of PGS and its original formulation is that we allow it to search while there is time available (see while loop in Algorithm 4). In its original formulation, PGS performs a fixed number of iterations of the while loop, which is determined by an input parameter; Churchill and Buro allowed PGS to perform only one iteration in their experiments.

3.2.2 Stratified Strategy Selection

Lelis (2017) introduced Stratified Strategy Selection (SSS). Similarly to PGS, SSS performs a hill-climbing search. However, in contrast to 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, which receives as input the current state s, a default script $\bar{\sigma}_d$, a set of scripts $\mathcal{P}$, a time limit e, an evaluation function Ψ, and a type system for the units $\mathcal{U}_i$ at state s. SSS returns a player-action for i and a Boolean value c indicating whether SSS was able to complete one iteration over the set of types in T. The Boolean value c is used by SSS with Adaptive Type Systems (SSS+), explained below.

In its 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. 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 9 and 10). SSS returns the player-action a_i once it reaches the time limit e (lines 12 and 14). Similarly to PGS, the action a_{-i} is fixed throughout the search and SSS tries to approximate a best response to the opponent's action given by script $\bar{\sigma}_{-i}$.

Algorithm 5 Stratified Strategy Selection

Input: 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.

Output: action a for player i's units, Boolean c indicating if the algorithm finished a complete iteration over all types in T

```
1: σ̄_i ← choose a script from P considering that −i acts according to σ̄_d //see text
2: σ̄_{−i} ← choose a script from P considering that i acts according to σ̄_i //see text
3: a_i ← {σ̄_i(u_1), σ̄_i(u_2), ··· , σ̄_i(u_n)}, where u_1, u_2, ··· , u_n ∈ U^s_{i,r}
4: a_{−i} ← {σ̄_{−i}(u_1), σ̄_{−i}(u_2), ··· , σ̄_{−i}(u_m)}, where u_1, u_2, ··· , u_m ∈ U^s_{−i,r}
5: c ← false
6: for each t ∈ T do
7:    for each σ̄ ∈ P do
8:       a'_i ← a_i with the actions of all units u of type t replaced by σ̄(u)
9:       if Ψ(T(s, a'_i, a_{−i})) > Ψ(T(s, a_i, a_{−i})) then
10:         a_i ← a'_i
11:      if time elapsed is larger than e then
12:         return a_i and Boolean c
13: c ← true // iterated over all types
14: return a_i and Boolean c
```

Depending on the number and on the diversity of units (e.g., units with different attack ranges) present in the match, SSS might be unable to iterate through all types in T before reaching time limit e. This is because a large diversity of units result in more types being considered during search. If SSS is unable to iterate through all types, then it returns the unit-actions computed from script $\bar{\sigma}_i$ for units u whose type $T(u)$ was not accounted for during search. The assignment of $\bar{\sigma}_i$ might lead to a poor overall strategy as there could be better scripts that were not verified by the algorithm due to the lack of time. Aiming at preventing SSS from not iterating at least once over all types, Lelis (2017) 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. Instead of receiving one type system T, SSS receives a set of type systems with different granularities (i.e., different sizes). If the algorithm returns false with its finest type system (the one with the largest number of types), then in the next decision-point it uses a coarser type system. If the algorithm returns true while using a type system T and the number of types in a finer T' times the average running time required to evaluate a script for a type is lower than the time limit (i.e., SSS estimates that it can complete an iteration with T'), then it switches to using T'.

4. Asymmetric Action Abstractions

Uniform abstractions are restrictive in the sense that all units have their legal actions reduced to those specified by scripts. In this section 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 is 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 $\mathcal{U}'_i \subseteq \mathcal{U}^s_{i,r}$, where $\mathcal{U}^s_{i,r}$ is player's i ready units in s, 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. If the set of unrestricted units is equal to the set of units controlled by the player, then the asymmetrically abstracted tree is equivalent to the unabstracted tree, and if the set of unrestricted units is empty, the asymmetrically abstracted tree is equivalent to the uniformly abstracted tree induced by the same set of scripts. 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 their "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 for these units by accounting for a larger set of unit-actions for them.

Next, we introduce four algorithms that search in real time in asymmetrically abstracted game trees.

5. 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. 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.

5.1 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 (Churchill & 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 $\mathcal{U}^s_{i,r} = \{u_1, u_2, u_3\}$ 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 $v = \{\bar{\sigma}_1, \bar{\sigma}_2, \bar{\sigma}_1\}$ 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 $v[u_1]$ to denote the script in v 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 $v[u_2] = \bar{\sigma}_2$, 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 v.

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 v 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 $\bar{\sigma} = v[u]$, 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 v 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 (Moraes et al., 2018b).

In our original version of GAB and SAB we defined the MFT differently; c.f. Example 1 and Definition 3 of Moraes and Lelis (2018). In our original work we used a script known as NOKAV in lieu of the scripts in v and of the script $\bar{\sigma}_{-i}$ defining the opponent model. This is because the MFT was defined in the context of Sparcraft, a domain for which NOKAV is strong (Churchill & Buro, 2013). Our current definition is more general because it does not assume the existence of a strong script and it uses byproducts of the search performed in the first step (the scripts v and $\bar{\sigma}_{-i}$) to define the MFT.

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 in 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.

5.1.1 Baselines for GAB and SAB: GAB$_{\mathcal{P}}$, SAB$_{\mathcal{P}}$, GAS, and SAS

In this section 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, the first two baselines we introduce, which we name GAB$_{\mathcal{P}}$, SAB$_{\mathcal{P}}$, use a two-step search scheme. However, instead of searching in asymmetrically abstracted trees, they search in uniformly abstracted trees. The other two baselines, which we name Greedy Asymmetric Search (GAS) and Stratified Asymmetric Search (SAS), also search in asymmetrically abstracted trees, but instead of performing two steps, it searches in a single step.

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 $\mathcal{U}'$. 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 strategies that are not weaker than the strategies optimal algorithms using Φ can derive. 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.

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. Suppose that there was an unrestricted unit u for which $T(u) = T(u')$ and SAS selected an unit-action m to be executed by u that is not returned by any of the scripts in $\mathcal{P}$, i.e., $m \notin \{\bar{\sigma}(u) | \bar{\sigma} \in \mathcal{P}\}$. In this case, one way of guaranteeing SSS' principle that units of the same type execute the action returned by the same script is to ensure that each unrestricted unit has its own type.

5.2 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 difference 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.

5.2.1 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 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.

2. The number '3' in A3N is the version of the algorithm; versions 1 and 2 (A1N and A2N) are described in Section 5.2.1.

Algorithm	Steps	Action Space	Search Type	Stratified
ABCD	1	Unabstracted	AB	No
NaïveMCTS	1	Unabstracted	MCTS	No
PGS	1	Uniformly Abstracted	HC	No
SSS	1	Uniformly Abstracted	HC	Yes
GAB	2	Asymmetrically Abstracted	AB and HC	No
SAB	2	Asymmetrically Abstracted	AB and HC	Yes
$\text{GAB}_{\mathcal{P}}$	2	Uniformly Abstracted	AB and HC	No
$\text{SAB}_{\mathcal{P}}$	2	Uniformly Abstracted	AB and HC	Yes
GAS	1	Asymmetrically Abstracted	HC	No
SAS	1	Asymmetrically Abstracted	HC	Yes
A1N	1	Uniformly Abstracted	MCTS	No
A2N	1	Asymmetrically Abstracted	MCTS	No
A3N	1	Asymmetrically Abstracted	MCTS	No

Table 1: Algorithms we evaluate in this paper and their features.

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. 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.

Table 1 summarizes all algorithms we evaluate in Section 6. The table distinguishes the algorithms by the number of steps performed (e.g., GAB performs two steps by searching with ABCD and PGS), action space (unabstracted, uniformly abstracted, or asymmetrically abstracted), search type, which includes Alpha Beta (AB), Monte Carlo Tree Search (MCTS), or hill climbing (HC), and whether the algorithm uses a stratification or not.

6. Empirical Evaluation

We evaluate the algorithms proposed in this paper to search in asymmetrically abstracted action spaces on μRTS (Ontañón, 2013), a real-time strategy game; the game is detailed in Section 6.1. Our empirical evaluation is divided into three parts. First, we evaluate different strategies for selecting the set of unrestricted units (Section 6.2). Next, we evaluate GAB, SAB, and A3N against their baselines GAS, $\text{GAB}_{\mathcal{P}}$, SAS, $\text{SAB}_{\mathcal{P}}$, A1N and A2N (Section 6.3). Finally, we compare GAB, SAB, and A3N against state-of-the-art search algorithms for RTS games (Section 6.5).

6.1 μRTS

μRTS is a challenging open-source RTS game developed for research purposes (Ontañón, 2013). Figure 1 shows a screenshot of a μRTS state of a match played on a 8×8 map. Circles

and squares with a blue or red contour denote units that can be controlled by the blue and red players, respectively. Colored squares without a contour can be either numbered or unnumbered, with the former denoting resources (the number shows the quantity of resources available) and the latter denoting obstacles that cannot be traversed by units. Uncolored squares are traversable regions of the map. The letters are not part of the game and were added to ease the description of the units and game mechanics that follows.

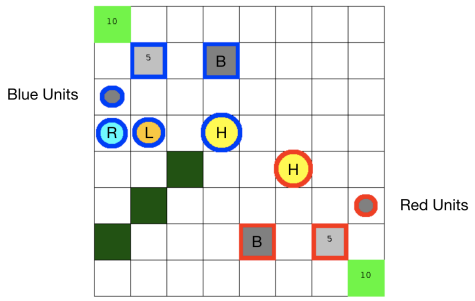

Figure 1: A μRTS state of a game in a 8×8 map.

Unit types Each player can train the following types of units: workers, light units, ranged units, heavy units, base, and barracks. The units that can move and attack are represented by a circle, the other units are represented by a square. The smallest circles (dark-gray) represent workers, yellow circles heavy units (with the letter 'H'), light-blue circles ranged units (with the letter 'R'), and small orange circles light units (with the letter 'L'). Light-gray squares represent bases (the number in the base shows the amount of resources available to the player). The dark-gray squares represent barracks (with the letter 'B').

Hit points Every unit has an amount of hit points that indicates the amount of damage the unit can suffer before being removed from the game. Workers and ranged units have fewer hit points than light and heavy units. The base has more hit points than any other unit. Some of the units can attack an enemy unit. If unit u attacks enemy unit u', then u reduces the hit points of u' according to its inflicted damage, which is determined by u's type. Workers and ranged units cause the least amount of damage, light and heavy units cause more damage. Workers, light, and heavy units u can only attack enemy units adjacent to the grid cell u occupies. Ranged units u can attack any enemy unit that is at an Euclidean distance of 3 grid cells or less from u.

Action scheme Most of the unit-actions require a single game cycle to be executed (RTS games typically have from 10 to 50 game cycles per second (Ontañón, 2017)), but some of the unit-actions (e.g., build a base) take several game cycles to complete (i.e., actions have different durations). Any unit can take a no-op action, which means that the unit waits until the next game cycle. All units other than base and barracks can move one grid cell at a time (up, down, left, and right). Only one unit can occupy a given grid cell at a time. If two units move simultaneously to the same grid cell, then the game server overwrites their actions with the no-op action.

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
basesWorkers24x24ABarracks	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

Table 2: 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.

Collecting and spending resources Bases and barracks cannot move nor attack, but the former can train workers and the latter can train light, heavy, and ranged units—at a cost of resources. Workers can build bases and barracks. Workers can also collect resources (one unit of resource at a time). Once collected, the resource must be delivered to the base by the worker. Once the collected resource is delivered at the player's base, it can be spent to train other units or build bases and barracks.

6.1.1 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 decision-point of the game.

Every match in our experiments is limited by a number of 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. (Barriga et al., 2018). The strategy played in μRTS varies depending on the map used. For example, "rush" strategies where the player trains weak units and send them out to quickly attack the opponent tend to work better in smaller maps, while strategies where the player evolves an economy before attacking the opponent tend to work better in larger maps. Table 2 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 and differ only on the initial position of the players in the 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 equal number of times. For example, if Algorithm 1 starts in location X with Algorithm 2 starting in location Y for five matches, we switch the starting positions for the remaining five matches.

The evaluation function Ψ used with our algorithms is the average result of two random play-outs. A random play-out evaluates a state s by simulating the game forward from s for

n game cycles with both players choosing actions randomly, thus leading to a state s'. Then, the evaluation of s is given by $\Phi(s')$, where Φ is a function introduced by Ontañón (2017). Φ computes a score for each player—$score(max)$ and $score(min)$, for players i and $-i$, respectively—by summing up the cost in resources required to train each unit controlled by the player weighted by the square root of the unit's hit points. The Φ value of a state is given by player max's score minus player min's score. Φ is then normalized to a value in $[-1, 1]$ through the following formula $\frac{2*score(max)}{score(min)+score(max)} - 1$.

We performed an experiment to choose the value of n each algorithm would use. Namely, we evaluated each algorithm played against different versions of itself in all maps used in our experiments. We evaluated versions of each algorithm where they used play-outs of length 50, 100, 150, and 200. Each algorithm played 10 matches (5 as player 1 and 5 as player 2) in each map against each version of itself, for a total of 30 matches in each map. We selected $n = 200$ for PGS, SSS, GAB, and SAB and $n = 100$ for A1N, A2N, and A3N.

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) (Stanescu, Barriga, Hess, & Buro, 2016; Silva et al., 2019). 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 tends 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 difference among them is the type of unit trained, WR trains workers units, LR trains light units, HR trains heavy units, and RR trains ranged units. The ABCD algorithm used in GAB and SAB uses the technique called scripted move ordering to allow for more pruning during search (Churchill et al., 2012). 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.

We use type systems that consider the following μRTS features: attack type (ranged or melee), mobility Type (stationary or mobile), hit points value. We use the following bucketing scheme for hit points (Lelis, 2017).

$$hp(u, l) = \left\lfloor \frac{hp(u)}{\lfloor \frac{hp_m(u)}{l} \rfloor} \right\rfloor .$$

Here, $hp(u)$ is the current number of hit points unit u has and $hp_m(u)$ is the maximum number of hit points the unit can have; the parameter l is an integer parameter. Larger values of l allows for a wider range of $hp(u, l)$-values, which result in finer type systems. For example, a type system that accounts for all features: (attack type, stationary, hit points with value of $l = 2$) assigns the following type to a light unit with 2 hit points: (melee, mobile, $\frac{2}{4/2} = 1$); $hp_m(u) = 4$ for light units. The type of a light unit with 4 hit points is: (melee, mobile, $\frac{4}{4/2} = 2$). SSS and SAB use a set of type systems of different granularity that are chosen depending on how fast the algorithm evaluates action assignments (Lelis, 2017). We use the following set of type systems, which are ordered from the coarsest (fewer types) to the finest type system (more types): $\{(T_0)$, (attack type), (attack type, mobility), (attack type, mobility, $hp(l = 1)$), (attack type, mobility, hp($l = 2$)), (attack type, mobility, hp($l = 3$))$\}$. Here, T_0 is a type system that assigns all units to the same type.

Our results are 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.

6.2 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. The intuition behind FC is to allow a finer control for the units that are distant from other ally units and might be unprotected.

2. **Closest to Centroid (CC)**. CC selects the N units that are closest to the centroid of all player i's units. This strategy serves as a baseline for FC.

3. **Closest to Enemy (CE)**. CE selects the N units that are closest to an enemy unit at every decision-point. Similarly to FC, the intuition behind CE is to allow a finer control for the units that are likely to be more threatened by enemy units.

4. **Farthest from Enemy (FE)**. FE selects the N units that are the farthest from an enemy unit. The intuition is that by providing a finer control to units that are far of the enemy, one might achieve a better overall positioning of units.

5. **Less life (HP-)**. HP- selects the N units with the lowest hit points. The intuition of this strategy is to provide a finer control to units that are about to be eliminated from the game, hoping that a finer control will keep these units longer in the match.

6. **More life (HP+)**. HP+ selects the units with more hit points at a given decision-point. This strategy can be helpful in scenarios where it is important to provide a finer plan to the structures responsible for training units (base and barracks) as they the units with largest number of hit points in the game.

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. AV+ selects the N units with the largest av-values. AV+ is similar to HP- as they both provide a finer control to units with low hp-value. However, AV+ selects units with low hp-value and/or large dpf-value, while HP- only accounts for hp. Moraes and Lelis (2018) showed that AV+ yields the best results for combat scenarios that arise in RTS matches.

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.

Time complexity is an important aspect of the strategies for selecting unrestricted units, as the time used to select units reduces the time allowed for planning. Strategy R has a

GAB vs. PGS										
Strategy	Unrestricted Set Size N									
	1	2	3	4	5	6	7	8	9	10
CC	76.0	71.5	59.0	55.0	47.0	40.5	44.5	38.0	37.0	43.5
FC	63.5	60.5	60.0	44.0	38.0	43.0	41.0	37.5	46.0	38.5
CE	82.5	81.0	65.0	55.5	62.0	58.5	45.0	45.5	40.5	46.0
FE	78.0	77.5	66.0	57.0	56.0	49.5	46.5	34.0	36.0	40.5
AV-	81.0	86.0	82.0	79.0	73.0	66.0	66.0	58.5	54.0	50.0
AV+	77.5	82.0	79.5	81.0	72.0	74.5	58.5	55.5	45.0	43.5
HP-	83.0	82.0	82.5	76.5	70.0	70.0	62.0	57.0	56.0	54.0
HP+	75.0	64.0	47.0	46.5	40.5	38.0	42.5	40.0	38.0	27.5
R	70.0	59.0	47.0	49.0	44.0	36.5	46.0	47.0	37.5	45.0

SAB vs. SSS										
Strategy	Unrestricted Set Size N									
	1	2	3	4	5	6	7	8	9	10
CC	73.0	68.0	66.5	59.0	47.0	61.0	60.5	60.0	61.5	64.0
FC	72.0	61.0	72.0	62.0	66.0	63.5	57.0	54.0	60.0	49.0
CE	83.5	81.0	67.0	69.5	67.0	65.5	67.0	58.5	61.0	59.0
FE	75.0	73.0	72.0	67.0	56.0	54.0	54.0	60.0	58.0	55.0
AV-	81.0	76.0	76.0	77.0	62.0	73.0	71.0	68.0	65.0	56.5
AV+	77.0	77.5	73.0	75.0	74.0	72.5	59.5	63.5	66.0	59.5
HP-	79.0	79.0	74.0	66.0	68.0	69.0	65.0	68.0	57.0	69.0
HP+	73.0	70.0	62.0	65.0	57.0	55.0	52.0	65.0	57.0	61.5
R	73.0	68.0	65.5	56.5	62.0	52.5	58.5	56.0	61.0	60.0

A3N vs. A1N										
Strategy	Unrestricted Set Size N									
	1	2	3	4	5	6	7	8	9	10
CC	85.0	69.0	58.5	53.5	51.5	35.0	28.5	26.5	22.0	23.5
FC	89.0	86.0	80.0	67.0	55.5	50.5	38.0	32.0	30.0	23.5
CE	87.5	82.0	73.0	68.0	55.5	43.0	38.0	28.5	18.0	26.5
FE	71.0	77.5	68.0	55.5	42.5	37.5	26.0	19.5	24.5	19.0
AV-	36.0	40.0	36.0	35.0	25.5	26.5	22.5	24.0	20.0	21.0
AV+	40.0	52.0	59.0	55.5	46.5	51.5	44.0	33.5	26.5	24.0
HP-	40.0	53.5	54.0	56.0	50.0	49.5	48.5	36.5	34.0	30.0
HP+	36.5	31.0	29.5	19.5	25.5	19.0	19.0	15.0	15.5	12.0
R	75.0	76.5	65.0	62.0	49.5	37.5	36.5	24.5	29.0	27.0

Table 3: Winning rate of variants of GAB, SAB, and A3N against their respective baselines 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).

time complexity of $O(1)$ as it needs to simply retrieve one random unit from the list of the player's units. FC, CC, HP-, HP+, AV+, and AV- have a time complexity of $O(n)$ for n

units. This is because it requires one to iterate once over all n units (HP-, HP+, AV+, and AV-) or twice over all units for FC and CC, once to compute the centroid and another for selecting the closest or the farthest unit. Assuming that both players control n units, a naïve implementation of CE and FE is $O(n^2)$ as one computes the distance between all pairs player's i and player's $-i$ units. A k-d tree can be used to reduce this complexity to $O(n \log n)$. We use a naïve implementation of CE and FE in our experiments. Despite having a worse time complexity, as we show in our experiments, CE performs better than other strategies because it allows the search algorithm to focus its effort in battles, thus allowing the agent to better plan the actions of units being threatened by opponent units.

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 3 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 3 to help us understand the results. In our color scheme, the lowest winning rate in the table (12.0 for A3N with HP+ and $N = 10$) has the lightest color and the largest winning rate (86.0 for GAB with AV- and $N = 3$) 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 (A3N is particular). This is because for larger N the space becomes too large to allow the algorithm to encounter strong strategies under real-time constraints. Table 3 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 86.0 (AV- with $N = 2$), 83.5 (CE with $N = 1$), and 89.0 (FC with $N = 1$), respectively. We use GAB with AV- and $N = 2$, SAB with with CE with $N = 1$, and A3N with FC and $N = 1$ in the remaining experiments of this paper. Next, we explain the results presented in Table 3 using domain-dependent knowledge. The reader not interested in the problem domain should skip to Section 6.2.1.

GAB performs best with AV- and has HP- with the second best score, 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 play-out function that performs fewer steps than the one used by GAB and SAB to evaluate actions. 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.

6.2.1 Evaluating Branching Factor for Different Strategies

In this section, we present statistics of the branching factor of the strategies for selecting unrestricted units presented in Section 6.2. We consider $\{1, 2, 3, 4, 5\}$ unrestricted units in five maps (in parenthesis we present the names shown in Table 4): `basesWorkers8x8A` (8×8), `basesWorkers16x16A` (16×16-1), `TwoBasesBarracks16x16` (16×16-2), `basesWorkers24x24A` (24×24-1) and `(4)BloodBathB` (24×24-2). Table 4 presents the average and maximum branching factor of five independent runs of A3N against WR for maps of size 8×8 and 16×16 and LR for maps of size 24×24; the opponents were chosen because they perform well in these maps (see Table 9).

The branching factor grows quickly with the size of the map in unabstracted spaces. Namely, it grows from a maximum of 1,079 in the 8×8 map to a maximum larger than 3,834,000 in the 24×24-2 map. The action abstractions maintain the branching factor within the hundreds for all maps if the number of restricted units is less than or equal to two. More unrestricted units will increase the branching to the thousands in some of the maps. As a result, the search algorithms are not able to evaluate all actions available within the domain's real-time constraints and they evaluate in search an arbitrary subset of the actions available, which can hamper their performance, as observed in Table 3.

Abstraction	Map									
	8×8		16×16-1		16×16-2		24×24-1		24×24-2	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
Unabstracted	30.6	1,079	101.9	9,808	4,526.7	1,264,000	11,283.7	2,737,863	12,241.2	3,834,400
AV+(1)	2.8	12	2.6	12	2.2	15	2.0	15	4.3	30
AV+(2)	6.8	51	6.8	46	6.0	60	3.8	60	20.0	186
AV+(3)	13.9	225	13.7	234	17.0	300	8.1	248	77.4	905
AV+(4)	23.9	689	27.4	1,149	47.7	1,921	20.4	1,560	225.2	2,625
AV+(5)	28.3	1,079	47.5	1,636	91.5	6,000	64.2	10,845	805.3	14,865
AV-(1)	1.5	10	2.3	9	2.2	15	1.4	15	2.4	9
AV-(2)	4.0	35	5.2	41	3.6	75	2.0	50	6.1	27
AV-(3)	8.8	158	15.5	196	6.4	325	3.3	190	13.8	133
AV-(4)	21.8	475	27.6	427	16.6	1,921	8.8	910	56.6	720
AV-(5)	28.8	1,079	51.3	1,636	20.4	1,921	22.4	2,900	242.0	3,060
CC(1)	3.0	7	2.2	12	2.3	21	34.6	1.8	2.6	21
CC(2)	4.5	20	3.8	37	5.6	108	34.7	4.1	7.9	66
CC(3)	7.5	158	6.8	111	17.4	540	34.8	8.6	27.3	279
CC(4)	11.4	475	13.3	683	43.3	2,340	35.0	21.5	113.3	2,196
CC(5)	28.3	1,079	19.5	1,000	39.5	11,700	36.5	48.5	492.9	14,292
CE(1)	2.8	12	2.6	15	2.4	18	2.2	21	2.5	24
CE(2)	5.5	51	5.1	65	4.0	65	3.9	69	7.3	100
CE(3)	11.7	158	10.8	245	8.1	269	7.9	387	15.1	165
CE(4)	22.3	475	23.2	440	22.9	1,921	14.8	782	47.3	615
CE(5)	27.6	1,079	51.0	1,636	36.2	3,000	32.8	3,473	188.4	3,060
FC(1)	2.0	9	2.6	9	2.5	18	2.0	30	3.1	12
FC(2)	5.3	36	5.9	42	4.1	60	3.9	150	12.8	110
FC(3)	11.4	158	13.0	196	10.6	399	9.4	483	50.7	730
FC(4)	23.1	475	30.8	1,149	30.6	1,921	25.8	1,560	173.5	5,509
FC(5)	29.9	1,079	53.9	1,636	50.4	1,921	78.5	6,570	478.5	10,392
FE(1)	2.3	12	2.6	12	1.9	15	1.7	30	4.7	30
FE(2)	5.4	37	6.0	46	5.3	75	3.8	123	19.4	183
FE(3)	9.5	158	16.5	196	18.2	399	9.2	354	76.3	1,668
FE(4)	22.0	475	29.6	400	38.8	1,921	23.2	1,352	262.3	2,298
FE(5)	29.2	1,079	53.2	1,636	54.9	1,921	65.1	11,024	983.5	10,977
HP+(1)	1.5	10	2.3	9	2.2	15	1.4	15	2.5	9
HP+(2)	4.4	40	5.1	35	3.6	75	2.6	50	6.1	27
HP+(3)	10.8	158	10.7	196	7.8	399	4.8	180	24.2	124
HP+(4)	22.3	475	24.2	539	19.4	1,921	9.6	820	86.9	615
HP+(5)	30.2	1,079	48.9	1,636	23.7	1,921	23.2	4,025	214.8	3,060
HP-(1)	2.6	12	2.6	12	2.1	13	2.0	15	4.3	30
HP-(2)	6.5	51	6.7	46	6.2	60	3.9	60	20.0	186
HP-(3)	14.0	225	13.8	234	17.3	300	8.3	248	77.4	905
HP-(4)	24.5	689	27.3	1,149	48.8	1,921	20.9	1,560	225.2	2,625
HP-(5)	30.1	1,079	47.8	1,636	104.7	6,000	68.3	10,845	805.3	14,865
R(1)	1.9	9	2.6	9	2.4	15	1.8	15	3.3	11
R(2)	4.8	36	5.7	42	4.4	65	3.6	120	11.9	100
R(3)	10.6	158	12.9	196	10.0	399	7.8	876	41.6	348
R(4)	22.1	475	28.2	539	27.7	1,921	20.4	5,520	121.4	1,137
R(5)	28.3	1,079	58.5	1,636	43.1	2,275	41.5	5,520	376.1	3,727

Table 4: Statistics of the branching factor of different abstraction schemes in five maps.

6.3 Comparison with Baselines

In this section, we evaluate GAB against its baselines PGS, GAS, and GAB$_{\mathcal{P}}$, SAB against SSS, SAS and SAB$_{\mathcal{P}}$, and A3N against A1N and A2N. Tables 5, 6, and 7 show the results for GAB, SAB, and A3N against their respective baselines.

	PGS	GAS	$GAB_{\mathcal{P}}$	GAB	Avg.
PGS	50.0	78.5	84.0	19.0	57.9
GAS	21.5	51.5	52.0	8.0	33.2
$GAB_{\mathcal{P}}$	16.0	48.0	54.0	20.0	34.5
GAB	81.0	92.0	80.0	51.5	76.1

Table 5: Comparison of GAB with its baselines. Winning rate of the row player against the column player.

	SSS	SAS	$SAB_{\mathcal{P}}$	SAB	Avg.
SSS	50.0	85.0	73.0	26.0	58.5
SAS	15.0	48.5	27.5	8.0	24.7
$SAB_{\mathcal{P}}$	27.0	72.5	58.0	26.0	45.8
SAB	74.0	92.0	74.0	52.5	73.1

Table 6: Comparison of SAB with its baselines. Winning rate of the row player against the column player.

	A1N	A2N	A3N	Avg.
A1N	50.0	24.0	5.5	26.5
A2N	76.0	51.5	27.0	51.5
A3N	94.5	73.0	50.5	72.7

Table 7: Comparison of A3N with its baselines. 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 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.

	HR	RR	AHT	NS	A1N	WR	SSS	PGS	PS	LR	GNS	STT	SAB	GAB	A3N	Avg.
HR	50.0	85.0	11.0	54.5	26.5	15.0	21.5	15.0	18.0	12.5	9.0	3.0	17.0	9.0	2.0	23.3
RR	15.0	50.0	57.0	78.5	28.5	60.0	15.0	7.0	0.0	0.0	9.0	15.0	6.0	12.0	2.0	23.7
AHT	89.0	43.0	50.5	21.0	9.5	18.0	29.0	19.0	31.5	44.5	25.0	9.0	17.5	18.0	14.5	29.3
NS	45.5	21.5	79.0	49.5	29.5	36.5	20.5	20.5	26.5	20.0	20.0	13.5	27.5	30.0	20.0	30.7
A1N	73.5	71.5	90.5	70.5	50.0	65.5	36.0	32.5	37.0	28.0	26.0	28.0	28.0	30.0	22.0	45.9
WR	85.0	40.0	82.0	63.5	34.5	50.0	46.0	43.0	35.0	35.0	35.0	36.5	35.0	33.0	39.0	46.2
SSS	78.5	85.0	71.0	79.5	64.0	54.0	50.0	47.0	42.5	36.0	34.0	24.0	15.0	20.0	12.0	47.5
PGS	85.0	93.0	81.0	79.5	67.5	57.0	53.0	50.0	49.0	40.0	36.0	35.5	27.0	29.0	19.0	53.4
PS	82.0	100.0	68.5	73.5	63.0	65.0	57.5	51.0	50.0	52.0	48.0	37.0	30.0	29.0	18.5	55.0
LR	87.5	100.0	55.5	80.0	72.0	65.0	64.0	60.0	48.0	50.0	53.0	52.0	34.0	36.0	20.0	58.5
GNS	91.0	91.0	75.0	80.0	74.0	65.0	66.0	64.0	52.0	47.0	49.5	48.0	44.0	44.0	13.5	60.3
STT	97.0	85.0	91.0	86.5	72.0	63.5	76.0	64.5	63.0	48.0	52.0	50.5	44.0	36.5	31.0	64.0
SAB	83.0	94.0	82.5	72.5	72.0	65.0	85.0	73.0	70.0	66.0	56.0	56.0	52.5	41.5	35.5	67.0
GAB	91.0	88.0	82.0	70.0	70.0	67.0	80.0	71.0	71.0	64.0	56.0	63.5	58.5	51.5	40.0	68.2
A3N	98.0	98.0	85.5	80.0	78.0	61.0	88.0	81.0	81.5	80.0	86.5	69.0	64.5	60.0	50.5	77.4

Table 8: Comparison of GAB, SAB, and A3N with current state-of-the-art search-based methods. The table shows the winning rate of the row player against the column player.

6.4 Frequency in Which GAB and SAB Return a_1 or a_2

We also evaluate the frequency in which GAB and SAB return action a_1 or a_2, the action from their first and second searches, respectively. GAB returns a_1 in 45% of the decision points in matches played on the `FourBasesWorkers8x8` map, while SAB returns a_1 in 42% of the decision points on the same map. GAB and SAB return a_1 in 47% and 49% of the decision points, respectively, in matches played on the `TwoBasesBarracks16x16` map. These results highlight the importance of evaluating a_1 and a_2 with the same evaluation function Ψ to only then return the action with higher Ψ-value.

6.5 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) (Churchill & Buro, 2013), Stratified Strategy Selection (SSS) (Lelis, 2017), Adversarial Hierarchical Task Network (AHT) (Ontañón & Buro, 2015), an algorithm that uses Monte Carlo tree search and HTN planning; NaïveMCTS (Ontañón, 2017) (henceforth referred as NS), Guided-NaïveMCTS (GNS) (Yang & Ontañón, 2019); the MCTS version of Puppet Search (PS) (Barriga et al., 2018), Strategy Tactics (STT) (Barriga et al., 2017), and four hard-coded scripts LR, RR, HR, and WR (Stanescu et al., 2016). The tables we present in this section use a cell-coloring scheme similar to the one used in Table 3. Cells with the lowest winning rate (0.0) have the lightest color and cells with the largest winning rate (100.0) have the darkest color; the remaining cell colors are chosen as a linear interpolation of the colors of the two extremes.

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 one might expect that algorithms searching in unabstracted spaces 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.

We start by presenting the winning rate of the algorithms tested in all 10 maps used in our experiments. Table 8 shows the winning rate of the row player against the column player. For example, A3N has a winning rate of 64.5 against SAB in all 10 maps. Overall, A3N wins more matches than any approach tested, with an average winning rate of 77.4. GAB ranks second with an average winning rate of 68.2, which is close to SAB's average winning rate of 67.0. The three best-performing methods use asymmetric action abstractions.

Next, we present the winning rate of the algorithms separated by map size. Table 9 shows the winning rate of search algorithms in small maps (top) and large maps (bottom). The small maps consist of the two maps of size 8×8 and the two of size 16×16. The large maps consist of the two maps of size 24×24, two of size 32×32, and two of size 64×64. We observe that NS, an algorithm that searches in unabstracted spaces, is superior to algorithms using uniform abstractions such as PGS and SSS on small maps. NS has an average winning rate of 56.8 while PGS and SSS have average winning rates of 48.9 and 41.6, respectively. However, on large maps, NS is consistently defeated by the same algorithms. The results shown in Table 9 also support our hypothesis that algorithms searching in asymmetrically abstracted spaces can perform well in both small and large maps as A3N is competitive with search-based methods in small maps and far superior to all methods tested in larger maps. GAB and SAB perform better than PGS and SSS on both small and large maps. The results on smaller maps show that the hard-coded script WR still outperforms all search-based algorithms tested: WR and A3N have average winning rates of 69.6 and 62.0, respectively. However, hard-coded strategies such as WR do not seem to generalize to larger maps as WR obtains an average winning rate of only 30.6 on them.

7. Conclusions

In this paper we introduced asymmetric action abstractions for multi-unit zero-sum extensive-form games. We also introduced A3N, GAB, and SAB, three search algorithms for searching in asymmetrically abstracted action 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 strategies derived by search algorithms are focused on the unrestricted units, as the algorithms are able to derive finer

Small Maps

	RR	HR	PS	SSS	AHT	GNS	PGS	LR	NS	A1N	STT	WR	GAB	SAB	A3N	Avg.
RR	50.0	25.0	0.0	32.5	35.0	15.0	17.5	0.0	47.5	15.0	10.0	25.0	30.0	15.0	5.0	21.5
HR	75.0	50.0	20.0	30.0	10.0	20.0	17.5	6.3	36.3	22.5	2.5	25.0	17.5	20.0	5.0	23.8
PS	100.0	80.0	50.0	43.8	36.3	57.5	35.0	52.5	33.8	23.8	20.0	12.5	12.5	17.5	20.0	39.7
SSS	67.5	70.0	56.3	50.0	37.5	50.0	47.5	45.0	50.0	32.5	15.0	22.5	30.0	27.5	22.5	41.6
AHT	65.0	90.0	63.8	62.5	50.0	57.5	47.5	75.0	5.0	22.5	22.5	27.5	42.5	27.5	32.5	46.1
GNS	85.0	80.0	42.5	50.0	42.5	48.8	55.0	35.0	50.0	50.0	45.0	30.0	37.5	47.5	21.3	48.0
PGS	82.5	82.5	65.0	52.5	52.5	45.0	50.0	47.5	48.8	35.0	32.5	25.0	42.5	35.0	37.5	48.9
LIR	100.0	93.8	47.5	55.0	25.0	65.0	52.5	50.0	50.0	45.0	35.0	25.0	42.5	32.5	30.0	49.9
NS	52.5	63.8	66.3	50.0	95.0	50.0	51.3	50.0	48.8	50.0	33.8	46.3	75.0	68.8	50.0	56.8
A1N	85.0	77.5	76.3	67.5	77.5	50.0	65.0	55.0	50.0	50.0	47.5	30.0	62.5	67.5	52.5	60.9
STT	90.0	97.5	80.0	85.0	77.5	55.0	67.5	65.0	66.3	52.5	48.8	45.0	60.0	67.5	60.0	67.8
WR	75.0	75.0	87.5	77.5	72.5	70.0	75.0	75.0	53.8	70.0	55.0	50.0	62.5	72.5	72.5	69.6
GAB	70.0	82.5	87.5	70.0	57.5	62.5	57.5	57.5	25.0	37.5	40.0	37.5	48.8	57.5	55.0	56.4
SAB	85.0	80.0	82.5	72.5	72.5	52.5	65.0	67.5	31.3	32.5	32.5	27.5	42.5	50.0	56.3	56.7
A3N	95.0	95.0	80.0	77.5	67.5	78.8	62.5	70.0	50.0	47.5	40.0	27.5	45.0	43.8	50.0	62.0

Large Maps

	NS	AHT	HR	RR	WR	A1N	SSS	PGS	STT	LR	PS	GNS	SAB	GAB	A3N	Avg.
NS	50.0	68.3	33.3	0.8	30.0	15.8	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	13.3
AHT	31.7	50.8	88.3	28.3	11.7	0.8	6.7	0.0	0.0	24.2	10.0	3.3	10.8	1.7	2.5	18.1
HR	66.7	11.7	50.0	91.7	8.3	29.2	15.8	13.3	3.3	16.7	16.7	1.7	15.0	3.3	0.0	22.9
RR	99.2	71.7	8.3	50.0	83.3	37.5	3.3	0.0	18.3	0.0	0.0	5.0	0.0	0.0	0.0	25.1
WR	70.0	88.3	91.7	16.7	50.0	10.8	25.0	21.7	24.2	8.3	0.0	11.7	10.0	13.3	16.7	30.6
A1N	84.2	99.2	70.8	62.5	89.2	50.0	15.0	10.8	15.0	10.0	10.8	10.0	1.7	8.3	1.7	35.9
SSS	99.2	93.3	84.2	96.7	75.0	85.0	50.0	46.7	30.0	30.0	33.3	23.3	6.7	13.3	5.0	51.4
PGS	100.0	100.0	86.7	100.0	78.3	89.2	53.3	50.0	37.5	35.0	38.3	30.0	21.7	20.0	6.7	56.4
STT	100.0	100.0	96.7	81.7	75.8	85.0	70.0	62.5	51.7	36.7	51.7	50.0	28.3	20.8	11.7	61.5
LR	100.0	75.8	83.3	100.0	91.7	90.0	70.0	65.0	63.3	50.0	48.3	45.0	35.0	31.7	13.3	64.2
PS	100.0	90.0	83.3	100.0	100.0	89.2	66.7	61.7	48.3	51.7	50.0	41.7	38.3	40.0	17.5	65.2
GNS	100.0	96.7	98.3	95.0	88.3	90.0	76.7	70.0	50.0	55.0	58.3	50.0	41.7	48.3	8.3	68.4
SAB	100.0	89.2	85.0	100.0	90.0	98.3	93.3	78.3	71.7	65.0	61.7	58.3	54.2	40.8	21.7	73.8
GAB	100.0	98.3	96.7	100.0	86.7	91.7	86.7	80.0	79.2	68.3	60.0	51.7	59.2	53.3	30.0	76.1
A3N	100.0	97.5	100.0	100.0	83.3	98.3	95.0	93.3	88.3	86.7	82.5	91.7	78.3	70.0	50.8	87.7

Table 9: Winning rate of the row player against the column player, divided into small maps (8×8 and 16×6) and large maps (24×24, 32×32 and 64×64).

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. If one considers large maps such as those used in commercial games, then A3N presented the strongest results. In smaller maps, a hard-coded script still outperformed all search algorithms tested. Although we performed our experiments on μRTS, the ideas of this paper are general and could be applied to other games. For example, card games such as Prismata (Churchill & Buro, 2015), 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 to 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 as some actuators might require a finer control than the others, similarly to the different levels of control required over units in games.

8. Acknowledgements

The authors gratefully thank Brazil's FAPEMIG, CNPq, CAPES and Canada's CIFAR AI Chairs program and NSERC for financial support. The authors also thank the anonymous reviewers for several great suggestions. We also thank Rob Holte for fruitful discussions and suggestions on an earlier draft of this paper. Rubens O. Moraes performed part of the work presented in this paper while visiting the University of Alberta. Levi Lelis is on leave from the Departamento de Informática, Universidade Federal de Viçosa.

References

Atkin, L., & Slate, D. (1988). Computer chess compendium.. chap. Chess 4.5-The Northwestern University Chess Program, pp. 80–103. Springer-Verlag, Berlin, Heidelberg.

Balla, R.-K., & Fern, A. (2009). UCT for tactical assault planning in real-time strategy games. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 40–45.

Barriga, N. A., Stanescu, M., & Buro, M. (2017). Combining strategic learning with tactical search in real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 9–15. AAAI Press.

Barriga, N. A., Stanescu, M., & Buro, M. (2018). Game tree search based on nondeterministic action scripts in real-time strategy games. *IEEE Transactions on Games*, *10*, 69–77.

Billings, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T., & Szafron, D. (2003). Approximating game-theoretic optimal strategies for full-scale poker. In *Proceedings of the International Joint Conference on Artificial Intelligence*, p. 661–668, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Bosanský, B., Lisý, V., Lanctot, M., Cermák, J., & Winands, M. H. M. (2016). Algorithms for computing strategies in two-player simultaneous move games. *Artificial Intelligence*, *237*, 1–40.

Brown, N., & Sandholm, T. (2018). Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, *359*(6374), 418–424.

Campbell, M., Hoane, A., & hsiung Hsu, F. (2002). Deep blue. *Artificial Intelligence*, *134*(1), 57 – 83.

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

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

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

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

Hawkin, J. A., Holte, R., & Szafron, D. (2011). Automated action abstraction of imperfect information extensive-form games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 681–687.

Hawkin, J. A., Holte, R., & Szafron, D. (2012). Using sliding windows to generate action abstractions in extensive-form games. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press.

Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, *26*(1), 191–246.

Hoffmann, J., & Nebel, B. (2001). The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*(1), 253–302.

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

Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, *6*(4), 293–326.

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

Kovarsky, A., & 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*, pp. 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*, pp. 3735–3741.

Lelis, L. H. S. (2020). Planning algorithms for zero-sum games with exponential action spaces: A unifying perspective. In *International Joint Conference on Artificial Intelligence*.

Liu, S., Louis, S. J., & 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.

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

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

Moraes, R. O., Mariño, J. R. H., & Lelis, L. H. S. (2018b). Nested-greedy search for adversarial real-time games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 67–73.

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*, pp. 58–64.

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

Ontañón, S. (2017). Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, *58*, 665–702.

Ontañón, S., Barriga, N. A., Silva, C. R., Moraes, R. O., & Lelis, L. H. (2018). The first microrts artificial intelligence competition.. *AI Magazine*, *39*(1).

Richter, S., & Helmert, M. (2009). Preferred operators and deferred evaluation in satisficing planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, *19*(1), 273–280.

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

Sandholm, T., & Singh, S. (2012). Lossy stochastic game abstraction with bounds. In *Proceedings of the ACM Conference on Electronic Commerce*, p. 880–897. Association for Computing Machinery.

Silva, C. R., Moraes, R. O., Lelis, L. H. S., & Gal, K. (2019). Strategy generation for multiunit real-time games via voting. *IEEE Transactions on Games*, *11*(4), 426–435.

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

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

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., ..., & Silver, D. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, *575*(7782), 350–354.

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

Yang, Z., & Ontañón, S. (2019). Guiding Monte Carlo Tree Search by Scripts in Real-Time Strategy Games. In *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 100–107.

Zobrist, A. L. (1990). A new hashing method with application for game playing. *ICGA Journal*, *13*, 69–73.