Planning Algorithms for Zero-Sum Games with Exponential Action Spaces: A Unifying Perspective

Levi H. S. Lelis
Department of Computing Science, University of Alberta, Canada
levi.lelis@ualberta.ca

Abstract

In this paper we review several planning algorithms developed for zero-sum games with exponential action spaces, i.e., spaces that grow exponentially with the number of game components that can act simultaneously at a given game state. As an example, real-time strategy games have exponential action spaces because the number of actions available grows exponentially with the number of units controlled by the player. We also present a unifying perspective in which several existing algorithms can be described as an instantiation of a variant of NaïveMCTS. In addition to describing several existing planning algorithms for exponential action spaces, we show that other instantiations of this variant of NaïveMCTS represent novel and promising algorithms to be studied in future works.

1 Introduction

In this paper we review algorithms for planning in two-players zero-sum games with exponential action spaces. In games with exponential action spaces each player controls a set of "components" that can individually issue actions in the game. For example, in real-time strategy (RTS) games each player controls a set of units [Ontañón *et al.*, 2013]. The set of player-actions is given by the combinations of all component actions; the action space grows exponentially with the number of components controlled by the players. Other examples of exponential action spaces include collectible card games [Ward and Cowling, 2009], where one decides on the action of several cards. In addition to being excellent testbeds for planning algorithms, games with exponential action spaces are of industrial interest [Churchill and Buro, 2015]. Although we focus on planning algorithms for games, we note that other problems such as multi-agent pathfinding also have exponential action spaces and other approaches such as learning algorithms have also been successfully applied to this class of problems [Vinyals *et al.*, 2019].

Most of games with exponential action spaces impose a restrictive time limits for planning. For example, card games impose limits on the order of seconds, while RTS games impose limits on the order of milliseconds. Thus, the success of planning algorithms for this class of games depends on how effectively they are able to eliminate from consideration a large portion of the action space. Action abstractions induced by a set of scripts were developed to effectively reduce the action space and are arguably the most important development within this research area. A script is a function mapping a state and a component to a component-action. An action abstraction can be defined by considering during search only the actions returned by scripts [Churchill and Buro, 2013]. Several algorithms were developed for searching in action-abstracted spaces (e.g., [Justesen *et al.*, 2014; Barriga *et al.*, 2017]). Another form of abstraction also introduced for this class of games is asymmetric action abstractions, which allow the search to "pay more attention" to more important components in the game [Moraes and Lelis, 2018].

In addition to planning algorithms, the community has developed methods for designing action abstractions. Mariño *et al.* [2019] introduced an evolutionary approach for learning scripts for inducing action abstractions. Search algorithms using action abstractions produced by their procedure were shown to be stronger than the same algorithms using action abstractions designed by human experts in an RTS game.

In this paper we review several planning algorithms for games with exponential action spaces and show that these algorithms can be described as a special case of a variant of NaïveMCTS [Ontañón, 2017], which we call General Combinatorial Search for Exponential Spaces (GEX). The instantiation of these algorithms in GEX points to novel algorithms that can be obtained by combining characteristics of existing ones. Implementing and evaluating such algorithms is left as future work for researchers interested in this class of games.

2 Games with Exponential Action Spaces

Although some of the games we consider have simultaneous moves, we focus on two-player zero-sum games with sequential moves and exponential action spaces. This is because previous works framed simultaneous moves as sequential games. We explain below how this was done in previous works.

Let the tuple $(N, S, s_{init}, A, R, T)$ be a two-player zero-sum game with sequential moves and exponential action spaces. Here, $N = \{i, -i\}$ is the pair of players (i is the "max" player and $-i$ is our opponent, the "min" player).

S is the set of states, which can be non-terminal or terminal. Every state $s \in S$ defines a joint set of components $C^s = C^s_i \cup C^s_{-i}$, for players i and $-i$. Every component

$c \in C^s$ has properties that are specific to the game. For example, in RTS games, the units controlled by the player are the components of the states. The states include properties such a unit's position on a map. s_{init} is the game's start state.

$A_i(s)$ (resp. $A_{-i}(s)$) is the set of legal actions player i (resp. $-i$) can perform at state s. Each action $a \in A_i(s)$ is denoted by a vector of n component-actions $(m_1, \cdots, m_n)$, where $m_k \in a$ is the action of the k-th ready component of player i. A component is not ready if it is already performing an action (component-actions can have different durations). We denote the set of ready components of players i and $-i$ as $C^s_{i,r}$ and $C^s_{-i,r}$. For component c, we write $a[c]$ to denote the action of c in a. We denote the set of component-actions as M and the set of legal actions of component c at s as $M(s, c)$.

$R_i(s)$ is a utility function with $R_i(s) = -R_{-i}(s)$, for any terminal state s. The transition function $T(s, a)$ deterministically determines the sucessor state for an action a applied at state s. Since actions can be durative, components might still be executing their actions in a in the state returned by $T(s, a)$.

A decision point of player i is a state s in which i has at least one ready component. A script is a function mapping a component c and a state s to a component-action m. We denote as $a_{\bar{\sigma}}$ the player-action defined by script $\bar{\sigma}$ at state s. The state s is not included in $a_{\bar{\sigma}}$ to ease the notation and because s will be clear from the context when the notation $a_{\bar{\sigma}}$ is used.

2.1 From Simultaneous to Sequential Games

Some of the exponential action spaces considered in the works reviewed in this paper are of simultaneous moves (e.g., RTS games). The works we review perform search on a "serialized" model of the simultaneous-move game. A simple serialization approach is to, during search, always allow player i to choose their action first, with player $-i$ being able to see i's choice before deciding on their action. Once both players chose their actions, they are applied to the state, thus generating the next state of the game. Kovarsky and Buro [2005] introduced an approach that randomly selects who is to choose first. This randomized approach is intended to reduce the advantage a player has if they systematically choose second. Another approach is to alternate the player who chooses first: if at a given state player i chooses first, then at the next state player $-i$ chooses first [Churchill *et al.*, 2012].

NaïveMCTS ignores the actions available to player $-i$ in a simultaneous-move state. If both players are to act simultaneously, then player i chooses an action and player $-i$ does not issue an action. Ontañón [2013] noted that other serializations produced similar results to ignoring $-i$'s actions. The explanation for Ontañón's results lies on the testbed used in his experiments: μRTS, an RTS game with durative moves. At a simultaneous-move state s of μRTS, player i issues their action, resulting in state s', while player $-i$ issues no actions. Then, since all units of player i are busy executing their actions at state s', player $-i$ is the only one who can act at s'. That is when player $-i$ issues their action. In μRTS, ignoring player $-i$'s actions at a simultaneous-move state has an effect similar to always allowing player i to choose their action first.

The difference between Ontañón's approach and the serializations studied by Kovarsky and Buro and Churchill *et al.* is the time in which the action effects take place. In Ontañón's scheme player i's actions might finish earlier than player $-i$'s actions as they start to be executed earlier. In contrast with Ontañón's scheme, in the Kovarsky and Buro and Churchill *et al.*'s schemes, the chosen actions start to be executed at the same time, only after i and $-i$ have made their decisions.

PGS [Churchill and Buro, 2013], POE [Wang *et al.*, 2016], and SSS [Lelis, 2017] bypass the serialization problem by approximating a best response to the action returned by a model of the opponent. That is, at state s, PGS, POE, and SSS assume that the opponent will perform an action a given by the model and they approximate a best response to a at s.

We unify these serialization approaches by assuming that the logic for deciding when the actions start to be executed to be encoded in the transition function $T(s, a)$. If one is to use Ontañón's scheme of ignoring a player's action, then a is applied to s as soon as $T(s, a)$ is invoked. Moreover, s is modified as a reference parameter in the call for T so that if $A(s)$ is invoked after $T(s, a)$, the state s passed as parameter to A is the modified state where action a was executed.

If one is to use a serialization scheme in which both actions are executed simultaneously, then action a is stored (but not executed) in s during the execution of $T(s, a)$ if a is the action of the player who chooses first; action a is executed jointly with the action stored in s if a is the action of the player who chooses second. The advantage of "hiding" the serialization logic in T is that the algorithms discussed in this paper can be applied to a larger class of games with exponential action spaces, including games with sequential or simultaneous moves and games with durative or non-durative actions.

3 Planning Algorithms

We divide the planning algorithms into three categories, according to the kind of action space used in search: uniformly-abstracted action spaces (PGS, NGS, POE, and SSS), asymmetrically-abstracted action spaces (A3N), and regular action spaces (NaïveMCTS and GNS). For ease of presentation we review the algorithms in the following ordering: PGS, NGS, SSS, NaïveMCTS, and A3N. GNS, A1N, A2N, and POE are described later, directly as instantiations of GEX, a general version of NaïveMCTS.

3.1 Uniform Action Abstractions

A uniform action abstraction for i is a function mapping the set of legal actions A_i to a subset A'_i of A_i. Action abstractions can be defined by a set of scripts P by defining $A'_i(s)$ with the set of actions returned by the scripts in P for s.

Definition 1. *A uniform abstraction Φ is a function receiving a state s, a player i, and a set of scripts P. Φ returns a subset of $A_i(s)$ denoted $A'_i(s)$. $A'_i(s)$ is defined by the Cartesian product of actions in $M(s, c, P) = \{\bar{\sigma}(s, c) | \bar{\sigma} \in P\}$ for all c in $C^s_{i,r}$, where $C^s_{i,r}$ is the set of ready components of i in s.*

Algorithms using a uniform abstraction consider only the actions in $A'_i(s)$ for all s encountered during search. The idea is to let the algorithms focus their search on actions deemed as promising by the scripts in P, as the actions in $A'_i(s)$ are composed of component-actions returned by the scripts in P.

Algorithm 1 Portfolio Greedy Search

Input: state s, set of scripts P, time limit e, a player-action a_o for $-i$, and evaluation function Ψ
Output: action for player i

```
1: T(s, a_o)
2: ā_i ← argmax_{ā∈P} Ψ(T(s, a_ā))
3: a_i ← a_{ā_i}
4: while time elapsed is not larger than e do
5:    for each c ∈ C^s_{i,r} do
6:       for each ā ∈ P do
7:          a'_i ← a_i; a'_i[c] ← ā(s, c)
8:          if Ψ(T(s, a'_i)) > Ψ(T(s, a_i)) then
9:             a_i ← a'_i
10:      if time elapsed is larger than e then
11:         return a_i
12: return a_i
```

Portfolio Greedy Search (PGS)

Churchill and Buro [2013] introduced Portfolio Greedy Search (PGS), a greedy search procedure for uniformly-abstracted action spaces. Algorithm 1 shows the pseudocode of PGS, which receives as input a state s, a set of scripts P, an opponent player-action a_o, a time limit e, and an evaluation function Ψ. PGS returns an action vector a for player i.

The opponent player-action a_o is defined by a model of opponent. For example, action a_o for player $-i$ can be defined by a default script (i.e., a script that is known to perform well in the task) or by a seeding procedure for player $-i$ that chooses a script $\bar{\sigma}$ from P for such that $a_{\bar{\sigma}}$ performs best against player i, which is assumed to play the action $a_{\bar{\sigma}_d}$ given by a default script $\bar{\sigma}_d$. PGS was originally presented with this seeding approach [Churchill and Buro, 2013].

In line 1 of PGS we invoke the transition function so that the action a_o is stored in s, but is not executed, if PGS is playing a simultaneous-move game. If the game is sequential, then a_o is a void action and $T(s, a_o)$ has no effect on state s. Then, in line 2 the transition function is invoked for the action $a_{\bar{\sigma}}$ that maximizes the estimated value of game given by function Ψ, this argmax operation is known as the seeding process for player i. This call of function T generates a new state s', which is generated by the application of a_o and $a_{\bar{\sigma}}$ to s; state s' is then evaluated by Ψ. Player-action a_i is initialized with $a_{\bar{\sigma}_i}$ (see line 3). PGS then performs a greedy search to improve the assignment of component-actions to a_i.

In its greedy search PGS iterates through all components c in $C^s_{i,r}$ and tries to greedily improve the component-action assigned to c in a_i, denoted by $a_i[c]$. Note that PGS only considers the component-actions in the uniform abstraction, i.e., those in $M(s, c, P)$. PGS evaluates a_i while replacing $a_i[c]$ by each of the possible component-actions m for c. 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; it then returns a_i (see lines 11 and 12).

Algorithm 1 differs in an important way from the original PGS. The opponent action a_o is fixed throughout search. Although in its original formulation PGS alternates between improving player i's and player $-i$'s action vectors, in their experiments, Churchill and Buro allowed PGS to improve only player i's action vector while player $-i$'s is fixed. Moraes *et al.* [2018b] showed that, if set to improve both a_i and a_o, PGS can suffer from a pathological issue that can cause PGS to find worse strategies than PGS with a_o fixed, even if the former is granted more computation time than the latter. In addition to identifying the pathology, Moraes *et al.* introduced an algorithm called Nested-Greedy Search (NGS), which we discuss next, that fixes the pathology identified.

Algorithm 2 GREEDY SEARCH (GS)

Input: state s, set of scripts P, action a_o for player $-i$, action a_i for player i, and evaluation function Ψ.
Output: value of action by player $-i$ in response for a_i.

```
1: B ← ∞; T(s, a_i)
2: for each c ∈ C^s_{−i,r} do
3:    for each ā ∈ P do
4:       a_{−i} ← a_o; a_{−i}[c] ← ā(s, c)
5:       if Ψ(T(s, a_{−i})) < B then
6:          a_o ← a_{−i} ; B ← Ψ(T(s, a_o))
7: return B
```

Nested-Greedy Search (NGS)

Instead of assuming a fixed opponent action a_o, NGS chooses for player i the action a_i with largest Ψ-value assuming that the opponent $-i$ will play an approximated best response to a_i. This is achieved by replacing the Boolean expression in line 8 in Algorithm 1, $\Psi(T(s, a'_i)) > \Psi(T(s, a_i))$, by the expression $\mathrm{GS}(s, a_o, a'_i, \Psi) > \mathrm{GS}(s, a_o, a_i, \Psi)$, where GS is a greedy search procedure similar to PGS's procedure for approximating a best response for player $-i$ for a fixed action for player i. The GS procedure is presented in Algorithm 2.

GS stores in variable $\mathcal{B}$ the value of the action for player $-i$ with lowest Ψ-value encountered during search, i.e., the value of the best response approximated by the greedy search. In line 1 GS initializes $\mathcal{B}$ to ∞ and invokes $T(s, a_i)$ to store a_i in s so that the subsequent calls to T with actions a_{-i} for $-i$ returns a state resulting from the application of both a_i and a_{-i} to s. GS then performs the same greedy search performed by PGS to approximate a best response to a_i. Moraes *et al.* [2018b] showed empirically in μRTS that NGS does not suffer from PGS's pathology. They also suggested that, if a good opponent model is known, then one should use PGS as we present in this paper; otherwise, one should use NGS.

Stratified Strategy Selection (SSS)

Similarly to PGS, SSS performs a greedy search [Lelis, 2017]. However, in contrast with PGS, SSS searches in the space of script assignments induced by a type system, which is a partition of a player's components. SSS always assigns the same script to components of the same type. For example, in RTS games, all units with low hit points (type) move away from the battle (strategy of a script). Formally, we have

Definition 2 (Type System)**.** *Let C_i be the set of player i's components. $\mathcal{T} = \{t_1, \ldots, t_k\}$ is a type system for C_i if it is a partitioning of C_i. If $c \in C_i$ and $t \in \mathcal{T}$, we say that c is of type t if $c \in t$.*

Algorithm 3 Stratified Strategy Selection

Input: state s, set of scripts P, player-action a_o for player $-i$, time limit e, evaluation function Ψ, and a type system $\mathcal{T}$ for the set of components $\mathcal{C}_i$ in s.
Output: action for player.

```
1: T(s, a_o)
2: σ̄_i ← argmax_{σ̄∈P} Ψ(T(s, a_σ̄))
3: a_i ← a_{σ̄_i}
4: while time elapsed is not larger than e do
5:    for each t ∈ T do
6:       for each σ̄ ∈ P do
7:          a'_i ← a_i with the actions of all components c of
            type t replaced by σ̄(c)
8:          if Ψ(T(s, a'_i)) > Ψ(T(s, a_i)) then
9:             a_i ← a'_i
10:         if time elapsed is larger than e then
11:            return a_i
12: return a_i
```

Algorithm 3 presents SSS, which receives as input a state s, a set of scripts P, an opponent player-action a_o, a time limit e, an evaluation function Ψ, and a type system $\mathcal{T}$ for components C_i at state s. SSS returns a player-action for i. The main difference between PGS and SSS is that the former tries different script assignments to all components, while the latter tries different script assignments to all types of components (see line 5 of Algorithm 3). The type system, which can be constructed with domain knowledge (e.g., all units with the same hit points and attack range have the same type), reduces the action space. That is, instead of evaluating $|C_i^s| \times |P|$ different actions like PGS, SSS evaluates $|\mathcal{T}| \times |P|$ actions in each iteration, with $\mathcal{T}$ being designed to be smaller than C_i^s.

3.2 Regular Action Spaces

Several algorithms were developed to search on the regular action spaces of games with exponential action spaces. Examples include Monte-Carlo methods [Chung *et al.*, 2005; Balla and Fern, 2009; Ontañón, 2013], Minimax search with Alpha-Beta pruning [Churchill *et al.*, 2012], and hierarchical search [Ontañón and Buro, 2015]. We present NaïveMCTS as a representative of such a family of algorithms [Ontañón, 2013; Ontañón, 2017].

Naïve Monte Carlo Tree Search (NaïveMCTS)
Ontañón [2017] modeled the search problem of deriving strategies in games with exponential action spaces 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 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 games with exponential action spaces, each decision point s can be cast as a CMAB problem in which X contains one variable for each ready component 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 component-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 component. 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 games with exponential action spaces, 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.

- n local MABs, one for each variable $X_i \in X$. For variable X_i representing the i-th component, the arms of the MAB are the K_i values (component-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 π_0 decides to explore, 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. 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 π_0 decides to exploit, 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) [Browne *et al.*, 2012] by introducing an algorithm named NaïveMCTS, for which we present the pseudocode when introducing GEX. NaïveMCTS differs from other MCTS algorithms in that it uses NS to decide which player-actions should be evaluated at a state. By contrast, a vanilla MCTS algorithm iteratively selects all player-actions to be evaluated at a given state.

Recently, Yang and Ontañón [2019] introduced Guided Naïve Sampling (GNS), a variant of NaïveMCTS whose exploration step is biased by a set of scripts, as we explain in Section 4.1.

3.3 Asymmetric Action Abstractions

Moraes and Lelis [2018] introduced the concept of asymmetric action abstractions for exponential action spaces. Uniform abstractions are restrictive in the sense that all components have their legal actions reduced to those specified by scripts. By contrast, asymmetric abstractions reduce the number of legal actions of only a subset of the player's components; the sets of legal actions of the other components remain unchanged. The subset of components that do not have their set

of legal actions reduced are known as the unrestricted components; the complement of the unrestricted components are known as the restricted components.

Definition 3. *An asymmetric abstraction Ω is a function receiving as input a state s, a player i, a set of unrestricted components $C'_i \subseteq C^s_{i,r}$, and a set of scripts P. Ω returns a subset of actions of $A_i(s)$, denoted $A''_i(s)$, defined by the Cartesian product of the component-actions in $M(s, c, P)$ for all c in $C^s_{i,r} \setminus C'_i$ and of component-actions $M(s, c')$ for all c' in C'_i.*

Algorithms using an asymmetric abstraction Ω consider only the actions in $A''_i(s)$ in search. If the set of unrestricted units is equal to the set of units controlled by the player, then the asymmetrically-abstracted space equals the regular space, and if the set of unrestricted units is empty, the asymmetrically-abstracted space is the same as the uniformly-abstracted space induced by the same set of scripts.

Asymmetric abstractions allow search algorithms to divide their "attention" differently among components at a given state of the game. That is, depending on the state, some components might be more important than others (e.g., in RTS games, units with low hit points can be more important), and asymmetric abstractions allow for finer strategies for these components by accounting for a larger set of actions for them.

Asymmetrically Abstracted NaïveMCTS (A3N)

Moraes *et al.* [2018a] introduced a version of NaïveMCTS, named A3N, that searches in asymmetrically-abstracted spaces by using an abstraction Ω and a modified version of π_l. A3N's π_l is able to select any component-action for the unrestricted components but only component-actions returned by scripts in P for the restricted components. Thus only macro-arms representing actions in $A''_i(s)$ are added to MAB_g.

An asymmetric abstraction is defined for a set of unrestricted components. Moraes *et al.* [2018a] evaluated several policies for choosing the unrestricted set and showed empirically in μRTS that A3N performs well even with a policy that randomly selects a subset of units as the unrestricted units.

4 GEX: A Unifying Algorithm

In this section we introduce General Combinatorial Search for Exponential Action Spaces (GEX) and show how all algorithms described in this paper can be seen as a special case of GEX. Algorithms 4 and 5 show the pseudocode for GEX. These two procedures are identical to NaïveMCTS. The difference between GEX and NaïveMCTS is how the policies π_0, π_l, π_g, and π (all parameter inputs to Algorithm 4) are defined and how the functions PROPAGATEEVALUATION, GETBESTACTION, and MACROARMSAMPLER are implemented. Next, we explain how the algorithms reviewed in this paper can be instantiated from Algorithms 4 and 5.

4.1 NaïveMCTS, GNS, A1N, A2N, and A3N

NaïveMCTS builds a search tree by adding a new node to the tree in each of its iterations. Algorithm 5 defines which node is added to the tree. Here, a node contains a state and other information, which depends on the algorithm GEX instantiates. The node to be added (leaf in Algorithm 4) is evaluated with Ψ and the value v thus obtained is propagated upward the tree.

Algorithm 4 GEX

Input: State s, sampling strategies π_0, π_l and π_g, serialization strategy π, opponent model O, and evaluation function Ψ, transition function T.
Output: Action a

```
1: root ← node(s)
2: while hasTime() do
3:     leaf ← SELECTANDEXPAND(root, π0, πl, πg, π, T)
4:     v ← Ψ(leaf.state)
5:     PROPAGATEEVALUATION(leaf, v)
6: return GETBESTACTION(root)
```

Algorithm 5 SELECTANDEXPAND

Input: A game tree node n_0 and sampling policies π_0, π_l and π_g, serialization policy π, and transition function T.
Output: A node in the tree

```
1: j ← π(n0.state)
2: n ← MACROARMSAMPLER(n0.state, π0, πl, πg, j)
3: if n ∈ n0.children then
4:     return SELECTANDEXPAND(n0.child(α))
5: else
6:     n0.addChild(n)
7:     return return n
```

GEX repeats this procedure while there is time available for planning. The algorithm then returns the best action available from the state in the root of the tree (line 6 of Algorithm 4).

We now describe the implementations of the policies and functions that allow GEX to instantiate NaïveMCTS.

Macro-Arm Sampling. NaïveMCTS uses NS as the function MACROARMSAMPLER with each of the policies π_0, π_l, and π_g implementing an ϵ-greedy policy.

Serialization Strategy. The policy π for deciding the player j that acts first (see line 1 of Algorithm 5) returns player i if i has ready components in the state; π returns $-i$ otherwise. The function $T(s, a)$ passed as parameter to GEX applies action a to s as soon as T is invoked. This way, the node n returned by MACROARMSAMPLER contains the state resulting from the application of a player j's action to $n_0.state$.

Evaluation Function. NaïveMCTS uses a random play-out as Ψ, i.e., a state is evaluated by randomly choosing player-actions for both players. If a terminal state is encountered after m steps, then the value of that state is returned; otherwise a heuristic value of the state reached is returned.

Propagation of Evaluation. The function PROPAGATEEVALUATION propagates the value v returned by function Ψ to all nodes on the path from the leaf to the root of the MCTS tree. In each node on the path NaïveMCTS updates the estimated average reward value $\mu'(v_i)$ for each component v_i.

Returning Best Action. Function GETBESTACTION returns the player-action (macro-arm) most visited at the root of the tree, i.e., the macro-arm that NS chose to be exploited the largest number of times at the root of the tree.

GNS, A1N, A2N, and A3N also use the same policies and functions described above, except for policy π_l that samples

macro-arms to be added to MAB_g. GNS prefers macro-arms returned by a script. That is, GNS's π_l returns the component actions returned by a script selected at random from a pool of options if the macro-arm to be added is the first of a node. In subsequent iterations, with probability ϵ, GNS's π_l returns a component action from a randomly selected script and, with probability $1 - \epsilon$, a component action returned by NS.

A1N searches in uniformly-abstracted spaces and can be implemented by having π_l sample component-actions returned by scripts in P. A2N is similar to A3N in that it also searches in asymmetrically-abstracted spaces, but it defines its abstraction according to the "type" of the player's components. In A2N each component type has access to a possibly different set of scripts P, thus making the abstraction asymmetric as some components can have access to sets P of different sizes. Finally, A3N can be defined by a policy π_l that samples actions from an asymmetric abstraction.

4.2 PGS, SSS, POE, and NGS

Next, we discuss the implementations of the policies and functions that allow GEX to instantiate PGS. We will then discuss the functions to instantiate SSS, POE, and NGS.

Macro-Arm Sampling. PGS evaluates each macro-arm at most once. This is achieved by GEX if policy π_0 always returns "explore" and adds an unseen macro-arm to MAB_g. Since PGS always chooses to explore, then policy π_g can be implemented arbitrarily, as it will never be invoked. Policy π_l stores information in the tree node so that it remembers which macro-arm should be evaluated next time π_l is invoked. Namely, π_l retrieves from the tree node the best incumbent action (this information is stored and updated by function PROPAGATEEVALUATION as explained below) and the indexes of the last component p and the last script q evaluated in the state. This way π_l is able to modify p and/or q so that π_l returns the macro-arms in the same order as PGS (see for loops in Algorithm 1). The first time π_l is invoked it performs the seeding step described in Section 3.1 by iteratively returning the actions $a_{\bar{\sigma}}$ for each of the scripts $\bar{\sigma}$ in P before returning the actions according to PGS's greedy search.

Serialization Strategy. The policy π always returns $-i$ in PGS. The first time π_l is invoked it uses the model of the opponent O, which for PGS is the seeding procedure we described in Section 3.1, to return an action a_o for $-i$. Action a_o is then passed to the transition function T, so that the value of a_o is stored within the function. The transition is only executed when policy π_l invokes T again with an action for i.

Evaluation Function. The function Ψ of PGS is deterministic, as described in PGS's original paper [Churchill and Buro, 2013]. It simulates the game from the leaf that is passed as parameter to Ψ by using a deterministic script for both players. If a terminal state is encountered after a fixed number m of steps, then the value of that state is returned; otherwise a heuristic value of the state reached after m steps is returned.

Propagation of Evaluation. The function PROPAGATEEVALUATION verifies if v is larger than the Ψ-value of the best incumbent action stored in the tree node. If it is, then PROPAGATEEVALUATION updates the best incumbent action to the leaf returned by SELECTANDEXPAND with Ψ-value of v.

Returning Best Action. Function GETBESTACTION returns the best incumbent action at the root of the tree.

SSS can be instantiated with the functions described above, but with a small change. Instead of having π_l iterating over all components, π_l for SSS should iterate over all types in a type system $\mathcal{T}$. A version of POE [Wang *et al.*, 2016] can also be obtained by implementing π_l as a genetic algorithm where the population is formed by a set of player-actions uniformly sampled at random from the uniform action abstraction. In POE, π_l returns the actions from the population that needs to be evaluated. The procedure PROPAGATEEVALUATION assigns v as the fitness value of the macro-arm evaluated.

To instantiate NGS, we change the serialization scheme, since in PGS and SSS player $-i$ chooses first while in NGS player i chooses first. In NGS, the policy π_l initially chooses 'explore' to add the first macro-arm a_i and its corresponding node n into the tree. Then, at the root, π_l chooses to 'exploit' a_i a number of times equal to the number of components controlled by $-i$ times the number of scripts in P, to allow for the evaluation of all actions of $-i$ that are considered by GS as responses to a_i. During this procedure, π_l always chooses to 'explore' at node n so that the algorithm evaluates all possible responses of the player $-i$. After all responses have been evaluated, π_l chooses to explore another macro-arm for player i at the root, thus repeating the procedure.

4.3 GEX Instantiating Novel Algorithms

The study of existing algorithms as instantiations of GEX points to the discovery of algorithms by combining the characteristics of existing methods. For example, a new algorithm is given by an instantiation similar to NaïveMCTS in which the policy π_l only samples macro-arms where components of the same type are assigned component-actions returned by the same script, as in SSS. Another novel algorithm is given by an instantiation of NaïveMCTS in which the function MACROARMSAMPLER implements the NGS's greedy search instead of NS's policy. Such an algorithm would iteratively perform a greedy search for player i and greedily approximate a best response for player $-i$. However, instead of performing a 2-step lookahead as in NGS, the tree could grow deeper. The instantiations of SSS, PGS, and POE only differ on the local search used within π_l, suggesting that other local search methods can be used as policies π_l.

5 Conclusions

In this paper we reviewed several algorithms for planning in games with exponential action spaces. We then presented a unifying perspective where a variant of NaïveMCTS, named GEX, can be used to describe several existing approaches. By instantiating existing algorithms in GEX one can easily see how combinations of features of existing approaches can lead to novel algorithms for this challenging class of games. We leave as future work for the research community the task of implementing and evaluating these novel search approaches.

Acknowledgements

The author thanks Rubens Moraes for discussions about this research and the reviewers for their suggestions.

References

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

[Barriga *et al.*, 2017] N. A Barriga, M. Stanescu, and M. Buro. Combining strategic learning and tactical search in real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 9–15. AAAI, 2017.

[Browne *et al.*, 2012] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Comp. Int. and AI in Games*, 4(1):1–43, 2012.

[Chung *et al.*, 2005] M. Chung, M. Buro, and J. Schaeffer. Monte Carlo planning in RTS games. In *Proceedings of the IEEE Symposium on Comp. Int. and Games*, 2005.

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

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

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

[Justesen *et al.*, 2014] N. Justesen, B. Tillman, J. Togelius, and S. Risi. Script- and cluster-based UCT for StarCraft. In *IEEE Conference on Comp. Int. and Games*, pages 1–8, 2014.

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

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

[Mariño *et al.*, 2019] J. R. H. Mariño, R. O. Moraes, C. Toledo, and L. H. S. Lelis. Evolving action abstractions for real-time planning in extensive-form games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 2330–2337, 2019.

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

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

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

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

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

[Ontañón, 2013] S. Ontañón. 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*, pages 58–64. AAAI, 2013.

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

[Vinyals *et al.*, 2019] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

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

[Ward and Cowling, 2009] C. D. Ward and P. I. Cowling. Monte carlo search applied to card selection in magic: The gathering. In *Proceedings of the International Conference on Comp. Int. and Games*, pages 9–16, 2009.

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