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 NaiveMCTS. In addition to describing several existing planning algorithms for exponential action spaces, we show that other instantiations of this variant of NaiveMCTS 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 NaiveMCTS [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 \(C_s^* = C_i^* \cup C_{-i}^*\), for players \(i\) and \(-i\). Every component
A \in \mathcal{A}(s) \ (\text{resp. } A_{-i}(s)) \text{ is the set of legal actions player } i \ (\text{resp. } \neg i) \text{ can perform at state } s. \text{ Each action } a \in A_i(s) \text{ is denoted by a vector of } n \text{ component-actions } (m_1, \ldots, m_n), \text{ where } m_k \in A \text{ is the action of the } k\text{-th ready component of player } i. \text{ 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 \text{ and } \neg i \text{ as } C^*_i \text{ and } C^*_\neg i. \text{ For component } c, \text{ we write } a[c] \text{ to denote the action of } c \text{ in } a. \text{ We denote the set of component-actions as } M \text{ and the set of legal actions of component } c \text{ at } s \text{ as } M(s, c).

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

A decision point of player } i \text{ is a state } s \text{ in which } i \text{ has at least one ready component. A script is a function mapping a component } c \text{ and a state } s \text{ to a component-action } m. \text{ We denote as } a_\sigma \text{ the player-action defined by script } \sigma \text{ at state } s. \text{ The state } s \text{ is not included in } a_\sigma \text{ to ease the notation and because } s \text{ will be clear from the context when the notation } a_\sigma \text{ 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 \text{ to choose their action first, with player } \neg i \text{ being able to see } i\text{’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 \text{ chooses first, then at the next state player } \neg i \text{ chooses first [Churchill et al., 2012].}

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

The difference between Ontaño’s approach and the serializations studied by Kovarsky and Buro and Churchill et al. is in the time in which the action effects take place. In Ontaño’s scheme player } i\text{’s actions might finish earlier than player } \neg i\text{’s actions as they start to be executed earlier. In contrast with Ontaño’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 \text{ and } \neg i \text{ 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, \text{ PGS, POE, and SSS assume that the opponent will perform an action } a \text{ given by the model and they approximate a best response to } a \text{ 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). \text{ If one is to use Ontaño’s scheme of ignoring a player’s action, then } a \text{ is applied to } s \text{ as soon as } T(s, a) \text{ is invoked. Moreover, } s \text{ is modified as a reference parameter in the call for } T \text{ so that if } A(s) \text{ is invoked after } T(s, a), \text{ the state } s \text{ passed as parameter to } A \text{ is the modified state where action } a \text{ was executed.}

If one is to use a serialization scheme in which both actions are executed simultaneously, then action } a \text{ is stored (but not executed) in } s \text{ during the execution of } T(s, a) \text{ if } a \text{ is the action of the player who chooses first; action } a \text{ is executed jointly with the action stored in } s \text{ if } a \text{ is the action of the player who chooses second. The advantage of “hiding” the serialization logic in } T \text{ 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 abstraction for } i \text{ is a function mapping the set of legal actions } A_i \text{ to a subset } A'_i \text{ of } A_i. \text{ Action abstractions can be defined by a set of scripts } P \text{ by defining } A'_i(s) \text{ with the set of actions returned by the scripts in } P \text{ for } s.

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

Algorithms using a uniform abstraction consider only the actions in } A'_i(s) \text{ for all } s \text{ encountered during search. The idea is to let the algorithms focus their search on actions deemed as promising by the scripts in } P, \text{ as the actions in } A'_i(s) \text{ are composed of component-actions returned by the scripts in } P.
## 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 \( \epsilon \), and an evaluation function \( \Psi \). PGS returns an action vector \( a_i \) 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 \( \sigma \) from \( P \) for which \( a_o \) performs best against player \( i \), which is assumed to play the action \( a_o \) given by a default script \( \sigma \). PGS was originally presented in 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 \( a_o \) that maximizes the estimated value of game given by function \( \Psi \), this argmax operation is known as the seeding process for player \( i \).

### Algorithm 1 Portfolio Greedy Search

<table>
<thead>
<tr>
<th>Input: state ( s ), set of scripts ( P ), time limit ( \epsilon ), a player-action ( a_o ) for (-i), and evaluation function ( \Psi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: action for player ( i )</td>
</tr>
<tr>
<td>1: ( T(s, a_o) )</td>
</tr>
<tr>
<td>2: ( \bar{a}<em>i \leftarrow \text{argmax}</em>{a \in P} \Psi(T(s, a)) )</td>
</tr>
<tr>
<td>3: ( a_i \leftarrow a_{\bar{a}_i} )</td>
</tr>
<tr>
<td>4: while time elapsed is not larger than ( \epsilon ) do</td>
</tr>
<tr>
<td>5: for each ( c \in C_{i,0} ) do</td>
</tr>
<tr>
<td>6: for each ( \sigma \in P ) do</td>
</tr>
<tr>
<td>7: ( a'_i \leftarrow a_i; a'_i[c] \leftarrow \sigma )</td>
</tr>
<tr>
<td>8: if ( \Psi(T(s, a'_i)) &gt; \Psi(T(s, a_i)) ) then</td>
</tr>
<tr>
<td>9: ( a_i \leftarrow a'_i )</td>
</tr>
<tr>
<td>10: if time elapsed is larger than ( \epsilon ) then</td>
</tr>
<tr>
<td>11: return ( a_i )</td>
</tr>
<tr>
<td>12: return ( a_i )</td>
</tr>
</tbody>
</table>

### Nests-Greedy Search (NGS)

Instead of assuming a fixed opponent action \( a_o \), NGS chooses for player \( i \) the action \( a_i \) with largest \( \Psi \)-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 \( \Psi(s, a_o, a'_i, \Psi) > \Psi(s, a_o, a_i, \Psi) \), where \( \Psi \) 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 \( B \) the value of the action for player \(-i\) with lowest \( \Psi \)-value encountered during search, i.e., the value of the best response approximated by the greedy search. In line 1 GS initializes \( B \) to \( \infty \) 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 \( a_i \).

### 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. \( T = \{t_1, \ldots, t_k\} \) is a type system for \( C_i \) if it is a partitioning of \( C_i \). If \( c \in C_i \) and \( t \in T \), we say that \( c \) is of type \( t \) if \( c \in t \).
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 \(\Psi\), and a type system \(T\) 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 \(|T| \times |P|\) actions in each iteration, with \(T\) 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; Onta˜nón, 2013], Minimax search with Alpha-Beta pruning [Churchill et al., 2012], and hierarchical search [Onta˜nón and Buro, 2015]. We present NaïveMCTS as a representative of such a family of algorithms [Onta˜nón, 2013; Onta˜nón, 2017].

**Naïve Monte Carlo Tree Search (NaïveMCTS)**

Onta˜nón [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 \((X, \mu)\), where,

- \(X = \{X_1, \cdots, X_n\}\), where each \(X_i\) is a variable that can assume \(K_i\) values \(X_i = \{v_1^{i}, \cdots, v_{K_i}^{i}\}\), with \(X = \{(v_1^{1}, \cdots, v_{n}^{n}) \in X_1 \times \cdots \times X_n\}\) being the possible combinations of value assignments for the variables in \(X\); a value assignment \(V \in X\) is called a macro-arm.

- \(\mu : X \rightarrow \mathbb{R}\) 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 \(V \in X\) represents a player-action and each value \(v \in V\) represents a component-action. The set \(X_i = \{v_1^{i}, \cdots, v_{K_i}^{i}\}\) 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 \(X\) is often too large in games with exponential action spaces, Onta˜nón [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 \(X_i\).
- 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 \(\pi_0\) 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 \(\pi_0\) decides to explore, then a macro-arm \(V\) is added to MAB\(_g\) by using a policy \(\pi_1\) 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 \(\pi_0\) decides to exploit, then a policy \(\pi_g\) is used to select an existing macro-arm in MAB\(_g\).

Onta˜nón [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 Onta˜nón [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 \( \Omega \) is a function receiving as input a state \( s \), a player \( i \), a set of unrestricted components \( C^u_i \subseteq C^u \), and a set of scripts \( P \). \( \Omega \) returns a subset of actions of \( A_i(s) \), denoted \( \Omega^u_i(s) \), defined by the Cartesian product of the component-actions in \( M(s, c, P) \) for all \( c \) in \( C^u_i \) and of component-actions \( M(s, c') \) for all \( c' \) in \( C^u_i \).

Algorithms using an asymmetric abstraction \( \Omega \) consider only the actions in \( \Omega^u_i(s) \) 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 \( \Omega \) and a modified version of \( \pi_1 \). A3N’s \( \pi_1 \) 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 macr-arms representing actions in \( \Omega^u_i(s) \) are added to MABs.

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 \( \mu \)RTS 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 \( \pi_0, \pi_1, \pi_g, \) and \( \pi \) (all parameter inputs to Algorithm 4) are defined and how the functions PROPAGATE-EVALUATION, GETBESTACTION, and MACROARM_SAMPLER 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 \( \Psi \) and the value \( v \) thus obtained is propagated upward the tree.

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 MACROARM_SAMPLER with each of the policies \( \pi_o, \pi_i, \) and \( \pi_g \) implementing an \( \epsilon \)-greedy policy.

Serialization Strategy. The policy \( \pi \) 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; \( \pi \) 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 MACROARM_SAMPLER contains the state resulting from the application of a player \( j \)’s action to \( n_o.state \).

Evaluation Function. NaïveMCTS uses a random play-out as the \( \Psi \), 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 PROPAGATE-EVALUATION propagates the value \( v \) returned by function \( \Psi \) 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 \( \pi_i \) that samples
macro-arms to be added to MAB$_g$. GNS prefers macro-arms returned by a script. That is, GNS’s $\pi_l$ 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 pair. In subsequent iterations, with probability $\epsilon$, GNS’s $\pi_l$ 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 $\pi_l$ 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 $\pi_l$ 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 $\pi_o$ always returns “explore” and adds an unseen macro-arm to MAB$_g$. Since PGS always chooses to explore, then policy $\pi_o$ can be implemented arbitrarily, as it will never be invoked. Policy $\pi_l$ stores information in the tree node so that it remembers which macro-arm should be evaluated next time $\pi_l$ is invoked. Namely, $\pi_l$ retrieves from the tree node the best incumbent action (this information is stored and updated by function PROPAGATE-EVALUATION as explained below) and the indexes of the last component $p$ and the last script $q$ evaluated in the state. This way $\pi_l$ is able to modify $p$ and/or $q$ so that $\pi_l$ returns the macro-arms in the same order as PGS (see for loops in Algorithm 1). The first time $\pi_l$ is invoked it performs the seeding step described in Section 3.1 by iteratively returning the actions $a_\sigma$ for each of the scripts $\sigma$ in $P$ before returning the actions according to PGS’s greedy search.

Serialization Strategy. The policy $\pi$ always returns $-i$ in PGS. The first time $\pi_l$ is invoked it models the value 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 $\pi_l$ invokes $T$ again with an action for $i$.

Evaluation Function. The function $\Psi$ 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 $\Psi$ 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 PROPAGATE-EVALUATION verifies if $v$ is larger than the $\Psi$-value of the best incumbent action stored in the tree node. If it is, then PROPAGATE-EVALUATION updates the best incumbent action to the leaf returned by SELECT-AND-EXPAND with $\Psi$-value of $v$.

Returning Best Action. Function GET-BEST-ACTION 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 $\pi_l$ iterating over all components, $\pi_l$ for SSS should iterate over all types in a type system $T$. A version of POE[Wang et al., 2016] can also be obtained by implementing $\pi_l$ 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, $\pi_l$ returns the actions from the population that needs to be evaluated. The procedure PROPAGATE-EVALUATION 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 $\pi_l$ initially chooses ‘explore’ to add the first macro-arm $a_i$ and its corresponding node $n$ into the tree. Then, at the root, $\pi_l$ 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, $\pi_l$ 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, $\pi_l$ 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 $\pi_l$ 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 MACRO-ARM-SAMPLER 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 $\pi_l$, suggesting that other local search methods can be used as policies $\pi_l$.

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


