A Polynomial-Time Algorithm for Action-Graph Games

Albert Xin Jiang **Kevin Leyton-Brown**
Department of Computer Science
University of British Columbia
{jiang;kevinlb}@cs.ubc.ca

Abstract

Action-Graph Games (AGGs) (Bhat & Leyton-Brown 2004) are a fully expressive game representation which can compactly express strict and context-specific independence and anonymity structure in players' utility functions. We present an efficient algorithm for computing expected payoffs under mixed strategy profiles. This algorithm runs in time polynomial in the size of the AGG representation (which is itself polynomial in the number of players when the in-degree of the action graph is bounded). We also present an extension to the AGG representation which allows us to compactly represent a wider variety of structured utility functions.[1]

Introduction

Game-theoretic models have recently been very influential in the computer science community. In particular, simultaneous-action games have received considerable study, which is reasonable as these games are in a sense the most fundamental. In order to analyze these models, it is often necessary to compute game-theoretic quantities ranging from expected utility to Nash equilibria.

Most of the game theoretic literature presumes that simultaneous-action games will be represented in normal form. This is problematic because quite often games of interest have a large number of players and a large set of action choices. In the normal form representation, we store the game's payoff function as a matrix with one entry for each player's payoff under each combination of all players' actions. As a result, the size of the representation grows exponentially with the number of players. Even if we had enough space to store such games, most of the computations we'd like to perform on these exponential-sized objects take exponential time.

Fortunately, most large games of any practical interest have highly structured payoff functions, and thus it is possible to represent them compactly. (Intuitively, this is why humans are able to reason about these games in the first place: we understand the payoffs in terms of simple relationships rather than in terms of enormous look-up tables.) One influential class of representations exploit strict independencies between players' utility functions; this class include graphical games (Kearns, Littman, & Singh 2001), multi-agent influence diagrams (Koller & Milch 2001), and game nets (LaMura 2000). A second approach to compactly representing games focuses on *context-specific* independencies in agents' utility functions – that is, games in which agents' abilities to affect each other depend on the actions they choose. Since the context-specific independencies considered here are conditioned on actions and not agents, it is often natural to also exploit *anonymity* in utility functions, where each agent's utilities depend on the distribution of agents over the set of actions, but not on the identities of the agents. Examples include congestion games (Rosenthal 1973) and local effect games (LEGs) (Leyton-Brown & Tennenholtz 2003). Both of these representations make assumptions about utility functions, and as a result cannot represent arbitrary games. Bhat & Leyton-Brown (2004) introduced action graph games (AGGs). Similar to LEGs, AGGs use graphs to represent the context-specific independencies of agents' utility functions, but unlike LEGs, AGGs can represent arbitrary games. Bhat & Leyton-Brown proposed an algorithm for computing expected payoffs using the AGG representation. For AGGs with bounded in-degree, their algorithm is exponentially faster than normal-form-based algorithms, yet still exponential in the number of players.

In this paper we make several significant improvements to results in (Bhat & Leyton-Brown 2004). First, we present an improved algorithm for computing expected payoffs. Our new algorithm is able to better exploit anonymity structure in utility functions. For AGGs with bounded in-degree, our algorithm is polynomial in the number of players. We then extend the AGG representation by introducing *function nodes*. This feature allows us to compactly represent a wider range of structured utility functions. We also describe computational experiments which confirm our theoretical predictions of compactness and computational speedup.

Action Graph Games

An action-graph game (AGG) is a tuple $\langle N, \mathbf{S}, \nu, u \rangle$. Let $N = \{1, \ldots, n\}$ denote the set of agents. Denote by $\mathbf{S} = \prod_{i \in N} S_i$ the set of action profiles, where $\prod$ is the Cartesian product and S_i is agent i's set of actions. We denote by $s_i \in S_i$ one of agent i's actions, and $\mathbf{s} \in \mathbf{S}$ an action profile.

Agents may have actions in common. Let $S \equiv \bigcup_{i \in N} S_i$ denote the set of distinct action choices in the game. Let Δ denote the set of *configurations* of agents over actions. A configuration $D \in \Delta$ is an ordered tuple of $|S|$ integers $(D(s), D(s'), \ldots)$, with one integer for each action in S. For each $s \in S$, $D(s)$ specifies the number of agents that chose action $s \in S$. Let $\mathcal{D} : \mathbf{S} \mapsto \Delta$ be the func-

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

[1]We would like to acknowledge the contributions of Navin A.R. Bhat, who is one of the authors of the paper which this work extends.

tion that maps from an action profile $\mathbf{s}$ to the corresponding configuration D. These shared actions express the game's anonymity structure: agent i's utility depends only on her action s_i and the configuration $\mathcal{D}(\mathbf{s})$.

Let G be the *action graph*: a directed graph having one node for each action $s \in S$. The neighbor relation is given by $\nu : S \mapsto 2^S$. If $s' \in \nu(s)$ there is an edge from s' to s. Let $D^{(s)}$ denote a configuration over $\nu(s)$, i.e. $D^{(s)}$ is a tuple of $|\nu(s)|$ integers, one for each action in $\nu(s)$. Intuitively, agents are only counted in $D^{(s)}$ if they take an action which is an element of $\nu(s)$. $\Delta^{(s)}$ is the set of configurations over $\nu(s)$ given that some player has played s.[2] Similarly we define $\mathcal{D}^{(s)} : \mathbf{S} \mapsto \Delta^{(s)}$ which maps from an action profile to the corresponding configuration over $\nu(s)$.

The action graph expresses *context-specific independencies* of utilities of the game: $\forall i \in N$, if i chose action $s_i \in S$, then i's utility depends only on the numbers of agents who chose actions connected to s, which is the configuration $\mathcal{D}^{(s_i)}(\mathbf{s})$. In other words, the configuration of actions not in $\nu(s_i)$ does not affect i's utility.

We represent the agents' utilities using a tuple of $|S|$ functions $u \equiv (u^s, u^{s'}, \ldots)$, one for each action $s \in S$. Each u^s is a function $u^s : \Delta^{(s)} \mapsto \mathbb{R}$. So if agent i chose action s, and the configuration over $\nu(s)$ is $D^{(s)}$, then agent i's utility is $u^s(D^{(s)})$. Observe that all agents have the same utility function, i.e. conditioned on choosing the same action s, the utility each agent receives does not depend on the identity of the agent. For notational convenience, we define $u(s, D^{(s)}) \equiv u^s(D^{(s)})$ and $u_i(\mathbf{s}) \equiv u(s_i, \mathcal{D}^{(s_i)}(\mathbf{s}))$.

Bhat & Leyton-Brown (2004) provided several examples of AGGs, showing that AGGs can represent arbitrary games, graphical games and games exhibiting context-specific independence without any strict independence. Due to space limits we do not reproduce these examples here.

Size of an AGG Representation

We have claimed that action graph games provide a way of representing games compactly. But what exactly is the size of an AGG representation? And how does this size grow as the number of agents n grows? Let $\mathcal{I} = \max_s |\nu(s)|$, i.e. the maximum in-degree of the action graph. The size of an AGG representation is dominated by the size of its utility functions.[3] For each action s, we need to specify a utility value for each distinct configuration $D^{(s)} \in \Delta^{(s)}$. The set of configurations $\Delta^{(s)}$ can be derived from the action graph, and can be sorted in lexicographical order. So we do not need to explicitly specify $\Delta^{(s)}$; we can just specify a list of $|\Delta^{(s)}|$ utility values that correspond to the (ordered) set of configurations.[4] $|\Delta^{(s)}|$, the number of distinct configurations over $\nu(s)$, in general does not have a closed-form expression. Instead, we consider the operation of extending all agents' action sets via $\forall i : S_i \mapsto S$. Now the number of configurations over $\nu(s)$ is an upper bound on $|\Delta^{(s)}|$. The bound is the number of (ordered) combinatorial compositions of $n-1$ (since one player has already chosen s) into $|\nu(s)|+1$ nonnegative integers, which is $\frac{(n-1+|\nu(s)|)!}{(n-1)!|\nu(s)|!}$. Then the total space required for the utilities is bounded from above by $|S|\frac{(n-1+\mathcal{I})!}{(n-1)!\mathcal{I}!}$. If $\mathcal{I}$ is bounded by a constant as n grows, the representation size grows like $O(|S|n^{\mathcal{I}})$, i.e. polynomially with respect to n.

For each AGG, there exists a unique *induced normal form* representation with the same set of players and $|S_i|$ actions for each i; its utility function is a matrix that specifies each player i's payoff for each possible action profile $\mathbf{s} \in \mathbf{S}$. This implies a space complexity of $n\prod_{i=1}^{n}|S_i|$. When $S_i \equiv S$ for all i, this becomes $n|S|^n$, which grows exponentially with respect to n. The number of payoff values stored in an AGG representation is always less than or equal to the number of payoff values in the induced normal form representation. For each entry in the induced normal form which represents i's utility under action profile $\mathbf{s}$, there exists a unique action profile $\mathbf{s}$ in the AGG with the corresponding action for each player. This $\mathbf{s}$ induces a unique configuration $\mathcal{D}(\mathbf{s})$ over the AGG's action nodes. By construction of the AGG utility functions, $\mathcal{D}(\mathbf{s})$ together with s_i determines a unique utility $u^{s_i}(\mathcal{D}^{(s_i)}(\mathbf{s}))$ in the AGG. Furthermore, there are no entries in the AGG utility functions that do not correspond to any action profile (s_i, s_{-i}) in the normal form. This means that there exists a many-to-one mapping from entries of normal form to utilities in the AGG. Of course, the AGG representation has the extra overhead of representing the action graph, which is bounded by $|S|\mathcal{I}$. But asymptotically, AGG's space complexity is never worse than the equivalent normal form.

Computing with AGGs

One of the main motivations of compactly representing games is to do efficient computation on the games. We focus on the computational task of computing expected payoffs under a mixed strategy profile. Besides being important in itself, this task is an essential component of many game-theoretic applications, e.g. computing best responses, Govindan and Wilson's continuation methods for finding Nash equilibria (2003; 2004), the simplicial subdivision algorithm for finding Nash equilibria (van der Laan, Talman, & van der Heyden 1987), and finding correlated equilibria using Papadimitriou's algorithm (2005).

Let $\varphi(X)$ denote the set of all probability distributions over a set X. Define the set of mixed strategies for i as $\Sigma_i \equiv \varphi(S_i)$, and the set of all mixed strategy profiles as $\Sigma \equiv \prod_{i \in N}\Sigma_i$. We denote an element of Σ_i by σ_i, an element of Σ by σ, and the probability that i plays action s as $\sigma_i(s)$.

[2] If action s is in multiple players' action sets (say players i, j), and these action sets do not completely overlap, then it is possible that the set of configurations given that i played s (denoted $\Delta^{(s,i)}$) is different from the set of configurations given that j played s. $\Delta^{(s)}$ is the union of these sets of configurations.

[3] The action graph can be represented as neighbor lists, with space complexity $O(|S|\mathcal{I})$.

[4] This is the most compact way of representing the utility functions, but does not provide easy random access of the utilities. Therefore, when we want to do computation using AGG, we may convert each utility function u^s to a data structure that efficiently implements a mapping from sequences of integers to (floating-point) numbers, (e.g. tries, hash tables or Red-Black trees), with space complexity in the order of $O(\mathcal{I}|\Delta^{(s)}|)$.

Define the expected utility to agent i for playing pure strategy s_i, given that all other agents play the mixed strategy profile σ_{-i}, as

$$V^i_{s_i}(\sigma_{-i}) \equiv \sum_{\mathbf{s}_{-i} \in \mathbf{S}_{-i}} u_i(s_i, \mathbf{s}_{-i}) \Pr(\mathbf{s}_{-i}|\sigma_{-i}). \quad (1)$$

where $\Pr(\mathbf{s}_{-i}|\sigma_{-i}) = \prod_{j \neq i} \sigma_j(s_j)$ is the probability of s_{-i} under the mixed strategy σ_{-i}.

Equation (1) is a sum over the set S_{-i} of action profiles of players other than i. The number of terms is $\prod_{j \neq i} |S_j|$, which grows exponentially in n. Thus (1) is an exponential time algorithm for computing $V^i_{s_i}(\sigma_{-i})$. If we were using the normal form representation, there really would be $|S_{-i}|$ different outcomes to consider, each with potentially distinct payoff values, so evaluation Equation (1) is the best we could do.

Can we do better using the AGG representation? Since AGGs are fully expressive, representing a game without any structure as an AGG would not give us any computational savings compared to the normal form. Instead, we are interested in structured games that have a compact AGG representation. In this section we present an algorithm that given any i, s_i and σ_{-i}, computes the expected payoff $V^i_{s_i}(\sigma_{-i})$ in time polynomial with respect to the size of the AGG representation. In other words, our algorithm is efficient if the AGG is compact, and requires time exponential in n if it is not. In particular, recall that for classes of AGGs whose in-degrees are bounded by a constant, their sizes are polynomial in n. As a result our algorithm will be polynomial in n for such games.

First we consider how to take advantage of the context-specific independence structure of the AGG, i.e. the fact that i's payoff when playing s_i only depends on the configurations in the neighborhood of i. This allows us to *project* the other players' strategies into smaller action spaces that are relevant given s_i. Intuitively we construct a graph from the point of view of an agent who took a particular action, expressing his indifference between actions that do not affect his chosen action. This can be thought of as inducing a context-specific graphical game. Formally, for every action $s \in S$ define a reduced graph $G^{(s)}$ by including only the nodes $\nu(s)$ and a new node denoted $\emptyset$. The only edges included in $G^{(s)}$ are the directed edges from each of the nodes $\nu(s)$ to the node s. Player j's action s_j is projected to a node $s_j^{(s)}$ in the reduced graph $G^{(s)}$ by the following mapping: $s_j^{(s)} \equiv \begin{cases} s_j & s_j \in \nu(s) \\ \emptyset & s_j \notin \nu(s) \end{cases}$. In other words, actions that are not in $\nu(s)$ (and therefore do not affect the payoffs of agents playing s) are projected to $\emptyset$. The resulting *projected* action set $S_j^{(s)}$ has cardinality at most $\min(|S_j|, |\nu(s)| + 1)$.

We define the set of mixed strategies on the projected action set $S_j^{(s)}$ by $\Sigma_j^{(s)} \equiv \varphi(S_j^{(s)})$. A mixed strategy σ_j on the original action set S_j is projected to $\sigma_j^{(s)} \in \Sigma_j^{(s)}$ by the following mapping:

$$\sigma_j^{(s)}(s_j^{(s)}) \equiv \begin{cases} \sigma_j(s_j) & s_j \in \nu(s) \\ \sum_{s' \in S_i \setminus \nu(s)} \sigma_j(s') & s_j^{(s)} = \emptyset \end{cases}. \quad (2)$$

So given s_i and σ_{-i}, we can compute $\sigma_{-i}^{(s_i)}$ in $O(n|S|)$ time in the worst case. Now we can operate entirely on the projected space, and write the expected payoff as

$$V^i_{s_i}(\sigma_{-i}) = \sum_{s^{(s_i)}_{-i} \in S^{(s_i)}_{-i}} u(s_i, \mathcal{D}^{(s_i)}(s_i, s_{-i})) \Pr(s^{(s_i)}_{-i}|\sigma^{(s_i)}_{-i})$$

where $\Pr(s^{(s_i)}_{-i}|\sigma^{(s_i)}_{-i}) = \prod_{j \neq i} \sigma^{(s_i)}_j(s^{(s_i)}_j)$. The summation is over $S^{(s_i)}_{-i}$, which in the worst case has $(|\nu(s_i)| + 1)^{(n-1)}$ terms. So for AGGs with strict or context-specific independence structure, computing $V^i_{s_i}(\sigma_{-i})$ this way is much faster than doing the summation in (1) directly. However, the time complexity of this approach is still exponential in n.

Next we want to take advantage of the anonymity structure of the AGG. Recall from our discussion of representation size that the number of distinct configurations is usually smaller than the number of distinct pure action profiles. So ideally, we want to compute the expected payoff $V^i_{s_i}(\sigma_{-i})$ as a sum over the possible configurations, weighted by their probabilities:

$$V^i_{s_i}(\sigma_{-i}) = \sum_{D^{(s_i)} \in \Delta^{(s_i,i)}} u_i(s_i, D^{(s_i)}) Pr(D^{(s_i)}|\sigma^{(s_i)}) \quad (3)$$

where $\sigma^{(s_i)} \equiv (s_i, \sigma^{(s_i)}_{-i})$ and

$$\Pr(D^{(s_i)}|\sigma^{(s_i)}) = \sum_{\mathbf{s}: \mathcal{D}^{(s_i)}(\mathbf{s}) = D^{(s_i)}} \prod_{j=1}^{N} \sigma_j(s_j) \quad (4)$$

which is the probability of $D^{(s_i)}$ given the mixed strategy profile $\sigma^{(s_i)}$. Equation (3) is a summation of size $|\Delta^{(s_i,i)}|$, the number of configurations given that i played s_i, which is polynomial in n if $\mathcal{I}$ is bounded. The difficult task is to compute $\Pr(D^{(s_i)}|\sigma^{(s_i)})$ for all $D^{(s_i)} \in \Delta^{(s_i,i)}$, i.e. the probability distribution over $\Delta^{(s_i,i)}$ induced by $\sigma^{(s_i)}$. We observe that the sum in Equation (4) is over the set of all action profiles corresponding to the configuration $D^{(s_i)}$. The size of this set is exponential in the number of players. Therefore directly computing the probability distribution using Equation (4) would take exponential time in n. Indeed this is the approach proposed in (Bhat & Leyton-Brown 2004).

Can we do better? We observe that the players' mixed strategies are independent, i.e. σ is a product probability distribution $\sigma(\mathbf{s}) = \prod_i \sigma_i(s_i)$. Also, each player affects the configuration D independently. This structure allows us to use dynamic programming (DP) to efficiently compute the probability distribution $\Pr(D^{(s_i)}|\sigma^{(s_i)})$. The intuition behind our algorithm is to apply one agent's mixed strategy at a time. Let $\sigma^{(s_i)}_{1\ldots k}$ denote the projected strategy profile of agents $\{1, \ldots, k\}$. Denote by $\Delta^{(s_i)}_k$ the set of configurations induced by actions of agents $\{1, \ldots, k\}$. Similarly denote $D^{(s_i)}_k \in \Delta^{(s_i)}_k$. Denote by P_k the probability distribution on $\Delta^{(s_i)}_k$ induced by $\sigma^{(s_i)}_{1\ldots k}$, and by $P_k[D]$ the probability of configuration D. At iteration k of the algorithm, we compute P_k from P_{k-1} and $\sigma^{(s_i)}_k$. After iteration n, the algorithm stops and returns P_n. The pseudocode of our DP algorithm is shown as Algorithm 1. Due to space limits we omit the proof of correctness of our algorithm.

Algorithm 1 Computing the induced probability distribution $\Pr(D^{(s_i)}|\sigma^{(s_i)})$.

Algorithm ComputeP
Input: s_i, $\sigma^{(s_i)}$
Output: P_n, which is the distribution $Pr(D^{(s_i)}|\sigma^{(s_i)})$ represented as a trie.
$D_0^{(s_i)} = (0, \ldots, 0)$
$P_0[D_0^{(s_i)}] = 1.0$ // Initialization: $\Delta_0^{(s_i)} = \{D_0^{(s_i)}\}$
for $k = 1$ to n **do**
  Initialize P_k to be an empty trie
  for all $D_{k-1}^{(s_i)}$ from P_{k-1} **do**
    for all $s_k^{(s_i)} \in S_k^{(s_i)}$ such that $\sigma_k^{(s_i)}(s_k^{(s_i)}) > 0$ **do**
      $D_k^{(s_i)} = D_{k-1}^{(s_i)}$
      if $s_k^{(s_i)} \neq \emptyset$ **then**
        $D_k^{(s_i)}(s_k^{(s_i)})$ += 1 // Apply action $s_k^{(s_i)}$
      end if
      if $P_k[D_k^{(s_i)}]$ does not exist yet **then**
        $P_k[D_k^{(s_i)}] = 0.0$
      end if
      $P_k[D_k^{(s_i)}]$ += $P_{k-1}[D_{k-1}^{(s_i)}] \times \sigma_k^{(s_i)}(s_k^{(s_i)})$
    end for
  end for
end for
return P_n

Each $D_k^{(s_i)}$ is represented as a sequence of integers, so P_k is a mapping from sequences of integers to real numbers. We need a data structure to manipulate such probability distributions over configurations (sequences of integers) which permits quick lookup, insertion and enumeration. An efficient data structure for this purpose is a *trie* (Fredkin 1962). Tries are commonly used in text processing to store strings of characters, e.g. as dictionaries for spell checkers. Here we use tries to store strings of integers rather than characters. Both lookup and insertion complexity is linear in $|\nu(s_i)|$. To achieve efficient enumeration of all elements of a trie, we store the elements in a list, in the order of their insertions.

Our algorithm for computing $V_{s_i}^i(\sigma_{-i})$ consists of first computing the projected strategies using (2), then following Algorithm 1, and finally doing the weighted sum given in (3). The overall complexity is $O(n|S| + n|\nu(s_i)|^2|\Delta^{(s_i,i)}(\sigma_{-i})|)$, where $\Delta^{(s_i,i)}(\sigma_{-i})$ denotes the set of configurations over $\nu(s_i)$ that have positive probability of occurring under the mixed strategy (s_i, σ_{-i}). Due to space limits we omit the derivation of this complexity result. Since $|\Delta^{(s_i,i)}(\sigma_{-i})| \leq |\Delta^{(s_i,i)}| \leq |\Delta^{(s_i)}|$, and $|\Delta^{(s_i)}|$ is the number of payoff values stored in payoff function u^{s_i}, this means that expected payoffs can be computed in polynomial time with respect to the size of the AGG. Furthermore, our algorithm is able to exploit strategies with small supports which lead to a small $|\Delta^{(s_i,i)}(\sigma_{-i})|$. Since $|\Delta^{(s_i)}|$ is bounded by $\frac{(n-1+|\nu(s_i)|)!}{(n-1)!|\nu(s_i)|!}$, this implies that if the in-degree of the graph is bounded by a constant, then the complexity of computing expected payoffs is $O(n|S| + n^{\mathcal{I}+1})$.

Theorem 1. *Given an AGG representation of a game, i's expected payoff $V_{s_i}^i(\sigma_{-i})$ can be computed in time polynomial in the size of the representation. If $\mathcal{I}$, the in-degree of the action graph, is bounded by a constant, $V_{s_i}^i(\sigma_{-i})$ can be computed in time polynomial in n.*

AGG with Function Nodes

There are games with certain kinds of context-specific independence structures that AGGs are not able to exploit.

Example 1. *In the Coffee Shop Game there are n players; each player is planning to open a new coffee shop in a downtown area, but has to decide on the location. The downtown area is represented by a $r \times c$ grid. Each player can choose to open the shop at any of the $B \equiv rc$ blocks, or decide not to enter the market. Conditioned on player i choosing some location s, her utility depends on the number of players that chose the same block, the number of players that chose any of the surrounding blocks, and the number of players that chose any other location.*

The normal form representation of this game has size $n|S|^n = n(B+1)^n$. Let us now represent the game as an AGG. We observe that if agent i chooses an action s corresponding to one of the B locations, then her payoff is affected by the configuration over all B locations. Hence, $\nu(s)$ would consist of B action nodes corresponding to the B locations. The action graph has in-degree $\mathcal{I} = B$. Since the action sets completely overlap, the representation size is $O(|S||\Delta^{(s)}|) = O(B\frac{(n-1+B)!}{(n-1)!B!})$. If we hold B constant, this becomes $O(Bn^B)$, which is exponentially more compact than the normal form representation. If we instead hold n constant, the size of the representation is $O(B^n)$, which is only slightly better than the normal form.

Intuitively, the AGG representation is only able to exploit the anonymity structure in this game. However, this game's payoff function does have context-specific structure. Observe that u^s depends only on three quantities: the number of players that chose the same block, the surrounding blocks, and other locations. In other words, u^s can be written as a function g of only 3 integers: $u^s(D^{(s)}) = g(D(s), \sum_{s' \in S'} D(s'), \sum_{s'' \in S''} D(s''))$ where S' is the set of actions that surrounds s and S'' the set of actions corresponding to the other locations. Because the AGG representation is not able to exploit this context-specific information, utility values are duplicated in the representation.

We can find similar examples where u^s could be written as a function of a small number of intermediate parameters. One example is a "parity game" where u^s depends only on whether $\sum_{s' \in \nu(s)} D(s')$ is even or odd. Thus u^s would have just two distinct values, but the AGG representation would have to specify a value for every configuration $D^{(s)}$.

This kind of structure can be exploited within the AGG framework by introducing *function nodes* to the action graph G. Now G's vertices consist of both the set of action nodes S and the set of function nodes P. We require that no function node $p \in P$ can be in any player's action set, i.e. $S \cap P = \{\}$. Each node in G can have action nodes and/or function nodes as neighbors. For each $p \in P$, we introduce a function $f_p : \Delta^{(p)} \mapsto \mathbb{N}$, where $D^{(p)} \in \Delta^{(p)}$ denotes configurations over p's neighbors. The configurations D are extended over the entire set of nodes, by defining $D(p) \equiv f_p(D^{(p)})$. Intuitively, $D(p)$ are the intermediate parameters that players' utilities depend on.

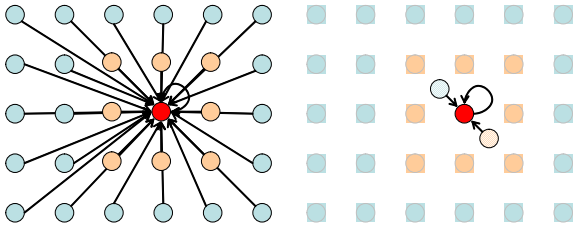

Figure 1: A 5×6 Coffee Shop Game: Left: the AGG representation without function nodes (looking at only the neighborhood of the a node s). Right: after introducing two function nodes, s now has only 3 incoming edges.

To ensure that the AGG is meaningful, the graph G restricted to nodes in P is required to be a directed acyclic graph (DAG). Furthermore it is required that every $p \in P$ has at least one neighbor (i.e. incoming edge). These conditions ensure that $D(s)$ for all s and $D(p)$ for all p are well-defined. To ensure that every $p \in P$ is "useful", we also require that p has at least one out-going edge. As before, for each action node s we define a utility function $u^s : \Delta^{(s)} \mapsto \mathbb{R}$. We call this extended representation $(N, \mathbf{S}, P, \nu, \{f_p\}_{p \in P}, u)$ an Action Graph Game with Function Nodes (AGGFN).

Representation Size

Given an AGGFN, we can construct an equivalent AGG with the same players N and actions S and equivalent utility functions, but represented without any function nodes. We put an edge from s' to s in the AGG if either there is an edge from s' to s in the AGGFN, or there is a path from s' to s through a chain of function nodes. The number of utilities stored in an AGGFN is no greater than the number of utilities in the equivalent AGG without function nodes. We can show this by following similar arguments as before, establishing a many-to-one mapping from utilities in the AGG representation to utilities in the AGGFN. On the other hand, AGGFNs have to represent the functions f_p, which can either be implemented using elementary operations, or represented as mappings similar to u^s. We want to add function nodes only when they represent meaningful intermediate parameters and hence reduce the number of incoming edges on action nodes.

Consider our coffee shop example. For each action node s corresponding to a location, we introduce function nodes p'_s and p''_s. Let $\nu(p'_s)$ consist of actions surrounding s, and $\nu(p''_s)$ consist of actions for the other locations. Then we modify $\nu(s)$ so that it has 3 nodes: $\nu(s) = \{s, p'_s, p''_s\}$, as shown in Figure 1. For all function nodes $p \in P$, we define $f_p(D^{(p)}) = \sum_{m \in \nu(p)} D(m)$. Now each $D^{(s)}$ is a configuration over only 3 nodes. Since f_p is a summation operator, $|\Delta^{(s)}|$ is the number of compositions of $n-1$ into 4 non-negative integers, $\frac{(n+2)!}{(n-1)!3!} = n(n+1)(n+2)/6 = O(n^3)$. We must therefore store $O(Bn^3)$ utility values.

Computing with AGGFNs

Our expected-payoff algorithm cannot be directly applied to AGGFNs with arbitrary f_p. First of all, projection of strategies does not work directly, because a player j playing an action $s_j \notin \nu(s)$ could still affect $D^{(s)}$ via function nodes. Furthermore, our DP algorithm for computing the probabilities does not work because for an arbitrary function node $p \in \nu(s)$, each player would not be guaranteed to affect $D(p)$ independently. Therefore in the worst case we need to convert the AGGFN to an AGG without function nodes in order to apply our algorithm. This means that we are not always able to translate the extra compactness of AGGFNs over AGGs into more efficient computation.

Definition 1. *An AGGFN is* contribution-independent (CI) *if*

- *For all $p \in P$, $\nu(p) \subseteq S$, i.e. the neighbors of function nodes are action nodes.*
- *There exists a commutative and associative operator $*$, and for each node $s \in S$ an integer w_s, such that given an action profile* $\mathbf{s}$*, for all $p \in P$, $D(p) = *_{i \in N : s_i \in \nu(p)} w_{s_i}$.*

Note that this definition entails that $D(p)$ can be written as a function of $D^{(p)}$ by collecting terms: $D(p) \equiv f_p(D^{(p)}) = *_{s \in \nu(p)}(*_{k=1}^{D(s)} w_s)$.

The coffee shop game is an example of a contribution-independent AGGFN, with the summation operator serving as $*$, and $w_s = 1$ for all s. For the parity game mentioned earlier, $*$ is instead addition mod 2. If we are modeling an auction, and want $D(p)$ to represent the amount of the winning bid, we would let w_s be the bid amount corresponding to action s, and $*$ be the max operator.

For contribution-independent AGGFNs, it is the case that for all function nodes p, each player's strategy affects $D(p)$ independently. This fact allows us to adapt our algorithm to efficiently compute the expected payoff $V^i_{s_i}(\sigma_{-i})$. For simplicity we present the algorithm for the case where we have one operator $*$ for all $p \in P$, but our approach can be directly applied to games with different operators associated with different function nodes, and likewise with a different set of w_s for each operator.

We define the *contribution* of action s to node $m \in S \cup P$, denoted $C_s(m)$, as 1 if $m = s$, 0 if $m \in S \setminus \{s\}$, and $*_{m' \in \nu(m)}(*_{k=1}^{C_s(m')} w_s)$ if $m \in P$. Then it is easy to verify that given an action profile $\mathbf{s}$, $D(s) = \sum_{j=1}^{n} C_{s_j}(s)$ for all $s \in S$ and $D(p) = *_{j=1}^{n} C_{s_j}(p)$ for all $p \in P$.

Given that player i played s_i, we define the projected contribution of action s, denoted $C_s^{(s_i)}$, as the tuple $(C_s(m))_{m \in \nu(s_i)}$. Note that different actions may have identical projected contributions. Player j's mixed strategy σ_j induces a probability distribution over j's projected contributions, $\Pr(C^{(s_i)} | \sigma_j) = \sum_{s_j : C_{s_j}^{(s_i)} = C^{(s_i)}} \sigma_j(s_j)$. Now we can operate entirely using the probabilities on projected contributions instead of the mixed strategy probabilities. This is analogous to the projection of σ_j to $\sigma_j^{(s_i)}$ in our algorithm for AGGs without function nodes.

Algorithm 1 for computing the distribution $Pr(D^{(s_i)} | \sigma)$ can be straightforwardly adopted to work with contribution-independent AGGFNs: whenever we apply player k's contribution $C_{s_k}^{(s_i)}$ to $D_{k-1}^{(s_i)}$, the resulting configuration $D_k^{(s_i)}$ is computed componentwise as follows: $D_k^{(s_i)}(m) =$

$C^{(s_i)}_{s_k}(m) + D^{(s_i)}_{k-1}(m)$ if $m \in S$, and $D^{(s_i)}_{k}(m) = C^{(s_i)}_{s_k}(m) * D^{(s_i)}_{k-1}(m)$ if $m \in P$. Following similar complexity analysis, if an AGGFN is CI, expected payoffs can be computed in polynomial time with respect to the representation size. Applied to the coffee shop example, since $|\Delta^{(s)}| = O(n^3)$, our algorithm takes $O(n|S| + n^4)$ time, which grows *linearly* in $|S|$.

Experiments

We implemented the AGG representation and our algorithm for computing expected payoffs in C++. We ran several experiments to compare the performance of our implementation against the (heavily optimized) GameTracer implementation (Blum, Shelton, & Koller 2002) which performs the same computation for a normal form representation. We used the Coffee Shop game (with randomly-chosen payoff values) as a benchmark. We varied both the number of players and the number of actions.

First, we compared the AGGFNs' representation size to that of the normal form. The results confirmed our theoretical predictions that the AGGFN representation grows polynomially with n while the normal form representation grows exponentially with n. (The graph is omitted because of space constraints.)

Second, we tested the performance of our dynamic programming algorithm against GameTracer's normal form based algorithm for computing expected payoffs, on Coffee Shop games of different sizes. For each game instance, we generated 1000 random strategy profiles with full support, and measured the CPU (user) time spent computing the expected payoffs under these strategy profiles. We fixed the size of blocks at 5×5 and varied the number of players. Figure 2 shows plots of the results. For very small games the normal form based algorithm is faster due to its smaller bookkeeping overhead; as the number of players grows larger, our AGGFN-based algorithm's running time grows polynomially, while the normal form based algorithm scales exponentially. For more than five players, we were not able to store the normal form representation in memory.

Next, we fixed the number of players at 4 and number of columns at 5, and varied the number of rows. Our algorithm's running time grew roughly linearly in the number of rows, while the normal form based algorithm grew like a higher-order polynomial. This was consistent with our theoretical prediction that our algorithm take $O(n|S| + n^4)$ time for this class of games while normal-form based algorithms take $O(|S|^{n-1})$ time.

Last, we considered strategy profiles having partial support (though space prevents showing the figure). While ensuring that each player's support included at least one action, we generated strategy profiles with each action included in the support with probability 0.4. GameTracer took about 60% of its full-support running times to compute expected payoffs in this domain, while our algorithm required about 20% of its full-support running times.

Conclusions

We presented a polynomial-time algorithm for computing expected payoffs in action-graph games. For AGGs with bounded in-degree, our algorithm achieves an exponential speed-up compared to normal-form based algorithms and Bhat & Leyton-Brown's algorithm (2004). We also extended the AGG representation by introducing function nodes, which allows us to compactly represent a wider range of structured utility functions. We showed that if an AGGFN is contribution-independent, expected payoffs can be computed in polynomial time.

In the full version of this paper we will also discuss speeding up the computation of Nash and correlated equilibria. We have combined our expected-payoff algorithm with GameTracer's implementation of Govindan & Wilson's algorithm (2003) for computing Nash equilibria, and achieved exponential speedup compared to the normal form. Also, as a direct corollary of our Theorem 1 and Papadimitriou's result (2005), correlated equilibria can be computed in time polynomial in the size of the AGG.

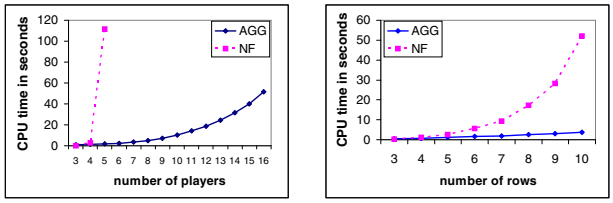

Figure 2: Running times for payoff computation in the Coffee Shop Game. Left: 5×5 grid with 3 to 16 players. Right: 4-player $r \times 5$ grid with r varying from 3 to 10.

References

Bhat, N., and Leyton-Brown, K. 2004. Computing Nash equilibria of action-graph games. In *UAI*.

Blum, B.; Shelton, C.; and Koller, D. 2002. Gametracer. http://dags.stanford.edu/Games/gametracer.html.

Fredkin, E. 1962. Trie memory. *Comm. ACM* 3:490–499.

Govindan, S., and Wilson, R. 2003. A global Newton method to compute Nash equilibria. *Journal of Economic Theory* 110:65–86.

Govindan, S., and Wilson, R. 2004. Computing Nash equilibria by iterated polymatrix approximation. *Journal of Economic Dynamics and Control* 28:1229–1241.

Kearns, M.; Littman, M.; and Singh, S. 2001. Graphical models for game theory. In *UAI*.

Koller, D., and Milch, B. 2001. Multi-agent influence diagrams for representing and solving games. In *IJCAI*.

LaMura, P. 2000. Game networks. In *UAI*.

Leyton-Brown, K., and Tennenholtz, M. 2003. Local-effect games. In *IJCAI*.

Papadimitriou, C. 2005. Computing correlated equilibria in multiplayer games. In *STOC*. Available at http://www.cs.berkeley.edu/˜christos/papers/cor.ps.

Rosenthal, R. 1973. A class of games possessing pure-strategy Nash equilibria. *Int. J. Game Theory* 2:65–67.

van der Laan, G.; Talman, A.; and van der Heyden, L. 1987. Simplicial variable dimension algorithms for solving the nonlinear complementarity problem on a product of unit simplices using a general labelling. *Mathematics of OR* 12(3):377–397.