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

Draft of April 18, 2006

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.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.

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 *et al.*, 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 and 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]. In Section 3, 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. In Section 4, we 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 in Section 6 which confirm our theoretical predictions of compactness and computational speedup.

2 Action Graph Games

2.1 Definition

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 actions 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 function 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.[1] 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 $u_i(\mathbf{s}) \equiv u(s_i, \mathcal{D}^{(s_i)}(\mathbf{s}))$.

2.2 Examples

Any arbitrary game can be encoded as an AGG as follows. Create a unique node s_i for each action available to each agent i. Thus $\forall s \in S,\ D(s) \in \{0, 1\}$, and $\forall i,\ \sum_{s \in S_i} D(s)$ must equal 1. The distribution simply indicates each agent's action choice, and the representation is no more or less compact than the normal form (see Section 2.3 for a detailed analysis).

Example 1. *Figure 1 shows an arbitrary 3-player, 3-action game encoded as an AGG. As always, nodes represent actions and directed edges represent membership in a node's neighborhood. The dotted boxes represent the players' action sets: player 1 has actions 1, 2 and 3; etc. Observe that there is always an edge between pairs of nodes belonging to different action sets, and that there is never an edge between nodes in the same action set.*

In a graphical game [Kearns *et al.*, 2001] nodes denote agents and there is an edge connecting each agent i to each other agent whose actions can affect

[1] 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.

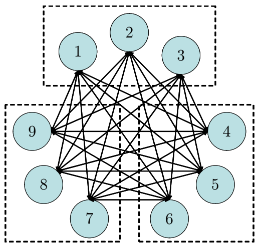

Figure 1: AGG representation of an arbitrary 3-player, 3-action game

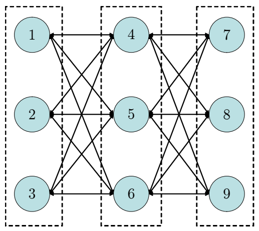

Figure 2: AGG representation of a 3-player, 3-action graphical game

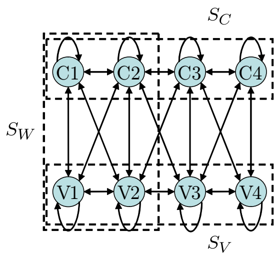

Figure 3: AGG representation of the ice cream vendor game

i's utility. Each agent then has a payoff matrix representing his local game with neighboring agents; this representation is more compact than normal form whenever the graph is not a clique. Graphical games can be represented as AGGs by replacing each node i in the graphical game by a distinct cluster of nodes S_i representing the action set of agent i. If the graphical game has an edge from i to j, create edges so that $\forall s_i \in S_i, \forall s_j \in S_j,\ s_i \in \nu(s_j)$. The resulting AGG representations are as compact as the original graphical game representations.

Example 2. *Figure 2 shows the AGG representation of a graphical game having three nodes and two edges between them (i.e., player 1 and player 3 do not directly affect each others' payoffs). The AGG may appear more complex than the graphical game; in fact, this is only because players' actions are made explicit.*

The AGG representation becomes even more compact when agents have actions in common, with utility functions depending only on the *number* of agents taking these actions rather than on the *identities* of the agents.

Example 3. *The action graph in Figure 3 represents a setting in which n vendors sell chocolate or vanilla ice creams, and must choose one of four locations along a beach. There are three kinds of vendors: n_C chocolate (C) vendors, n_V vanilla vendors, and n_W vendors that can sell both chocolate and vanilla, but only on the west side. Chocolate (vanilla) vendors are negatively affected by the presence of other chocolate (vanilla) vendors in the same or neighboring locations, and are simultaneously positively affected by the presence of nearby vanilla (chocolate) vendors. Note that this game exhibits context-specific independence without any strict independence, and that the graph structure is independent of n.*

Other examples of compact AGGs that cannot be compactly represented as graphical games include: location games, role formation games, traffic routing games, product placement games and party affiliation games.

2.3 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? From the definition of AGG in Section 2.1, we observe that we need the following to completely specify an AGG:

- The set of agents $N = \{1, \ldots, n\}$. This can be specified by the integer n.
- The set of actions S.
- Each agent's action set $S_i \subseteq S$.
- The action graph G. The set of nodes is S, which is already specified. The neighbor relation ν can be straightforwardly represented as neighbor lists: for each node $s \in S$ we specify its list of neighbors $\nu(s) \subseteq S$. The space required is $\sum_{s \in S} |\nu(s)|$, which is bounded by $|S|\mathcal{I}$, where $\mathcal{I} = \max_s |\nu(s)|$, i.e. the maximum in-degree of the action graph.
- For each action s, the utility function $u^s : \Delta^{(s)} \mapsto \mathbb{R}$. 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.[2] $|\Delta^{(s)}|$, 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 $|S|\frac{(n-1+\mathcal{I})!}{(n-1)!\mathcal{I}!}$.

Therefore the size of an AGG representation is dominated by the size of its utility functions, which is bounded 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.

The AGG representation achieves compactness by exploiting two types of structure in the utilities:

1. **Anonymity:** agent i's utility depends only on her action s_i and the configuration (i.e. number of players that play each action), but not on

[2]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 $O(\mathcal{I}|\Delta^{(s)}|)$.

the identities of the players. Since the number of configurations $|\Delta|$ is usually less than the number of action profiles $|\mathbf{S}| = \prod_i |S_i|$ and is never greater, we need fewer numbers to represent the utilities in AGG compared to the normal form.

2. **Context-specific independence:** for each node $s \in S$, the utility function u^s only needs to be defined over $\Delta^{(s)}$. Since $|\Delta^{(s)}|$ is usually less than $|\Delta|$ and is never greater, this further reduces the numbers we need to specify.

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.

Theorem 1. *The number of payoff values stored in an AGG representation is always less or equal to the number of payoff values in the induced normal form representation.*

Proof. 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. $\square$

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.

3 Computing with AGGs

One of the main motivations of compactly representing games is to do efficient computation on the games. We have introduced AGG as a compact representation of games; now we would like to exploit the compactness of the AGG representation when we do computation. 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 [Govindan & Wilson, 2003; Govindan & Wilson, 2004], the simplicial subdivision algorithm for finding Nash equilibria [van der Laan *et al.*, 1987], and finding correlated equilibria using Papadimitriou's algorithm [Papadimitriou, 2005].

Besides exploiting the compactness of the representation, we would also like to be able to exploit the fact that quite often the mixed strategy profile given will have small *support*. The support of a mixed strategy σ_i is the set of pure strategies played with positive probability (i.e. $\sigma_i(s_i) > 0$). Quite often games have Nash equilibria with small support. Porter *et al.* [2004] proposed algorithms that explicitly search for Nash equilibria with small support. In other algorithms for computing Nash equilibria such as Govindan-Wilson and simplicial subdivision, quite often we will also be computing expected payoffs for mixed strategy profiles with small support. Our algorithm appropriately exploits strategy profiles with small supports.

3.1 Notation

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 $\sigma_i(s)$.

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

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

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}.

The set of i's pure strategy best responses to a mixed strategy profile σ_{-i} is $\arg\max_s V^i_s(\sigma_{-i})$, and hence the full set of i's pure and mixed strategy best responses to σ_{-i} is

$$BR_i(\sigma_{-i}) \equiv \varphi(\arg\max_s V^i_s(\sigma_{-i})). \tag{2}$$

A strategy profile σ is a Nash equilibrium iff

$$\forall i \in N,\ \sigma_i \in BR_i(\sigma_{-i}). \tag{3}$$

3.2 Computing $V^i_{s_i}(\sigma_{-i})$

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 Equation (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 for computing $V^i_{s_i}$.

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. This is illustrated in Figure 4, using the ice cream vendor game (Figure 3). 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:

$$s_j^{(s)} \equiv \begin{cases} s_j & s_j \in \nu(s) \\ \emptyset & s_j \notin \nu(s) \end{cases} . \tag{4}$$

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:

$$\sigma_j^{(s)}(s_j^{(s)}) \equiv \begin{cases} \sigma_j(s_j) & s_j \in \nu(s) \\ \sum_{s' \in S_i \setminus \nu(s)} \sigma_j(s') & s_j^{(s)} = \emptyset \end{cases} . \tag{5}$$

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

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

where $\Pr(s_{-i}^{(s_i)} | \sigma_{-i}^{(s_i)}) = \prod_{j \neq i} \sigma_j^{(s_i)}(s_j^{(s_i)})$. The summation is over $S_{-i}^{(s_i)}$, 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

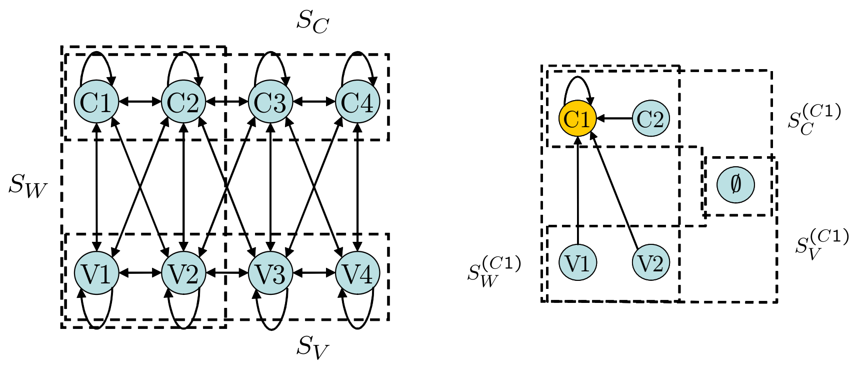

Figure 4: Projection of the action graph. Left: action graph of the ice cream vendor game. Right: projected action graph and action sets with respect to the action C1.

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:

$$V^i_{s_i}(\sigma_{-i}) = \sum_{D^{(s_i)} \in \Delta^{(s_i,i)}} u_i(s_i, D^{(s_i)}) Pr(D^{(s_i)}|\sigma^{(s_i)}) \tag{6}$$

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

$$\Pr(D^{(s_i)}|\sigma^{(s_i)}) = \sum_{\mathbf{s}:\mathcal{D}^{(s_i)}(\mathbf{s})=D^{(s_i)}} \prod_{j=1}^{N} \sigma_j(s_j) \tag{7}$$

which is the probability of $D^{(s_i)}$ given the mixed strategy profile $\sigma^{(s_i)}$. Equation (6) 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 (7) 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 (7) 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

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.
$D_0^{(s_i)} = (0, \dots, 0)$
$P_0[D_0^{(s_i)}] = 1.0$ // 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 $P_n$
```

agent's mixed strategy at a time. In other words, we add one agent at a time to the action graph. Let $\sigma_{1\dots k}^{(s_i)}$ denote the projected strategy profile of agents $\{1, \dots, k\}$. Denote by $\Delta_k^{(s_i)}$ the set of configurations induced by actions of agents $\{1, \dots, k\}$. Similarly denote $D_k^{(s_i)} \in \Delta_k^{(s_i)}$. Denote by P_k the probability distribution on $\Delta_k^{(s_i)}$ induced by $\sigma_{1\dots k}^{(s_i)}$, 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 $\sigma_k^{(s_i)}$. After iteration n, the algorithm stops and returns P_n. The pseudocode of our DP algorithm is shown as Algorithm 1.

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.

3.3 Proof of correctness

It is straightforward to see that Algorithm 1 is computing the following recurrence in iteration k:

$$\forall D_k \in \Delta_k^{(s_i)}, \quad P_k[D_k] = \sum_{D_{k-1}, s_k^{(s_i)} : \mathcal{D}^{(s_i)}(D_{k-1}, s_k^{(s_i)}) = D_k} P_{k-1}[D_{k-1}] \times \sigma_k^{(s_i)}(s_k^{(s_i)}) \tag{8}$$

where $\mathcal{D}^{(s_i)}(D_{k-1}, s_k^{(s_i)})$ denotes the configuration resulting from applying k's projected action $s_k^{(s_i)}$ to the configuration $D_{k-1} \in \Delta_k^{(s_i)}$.

On the other hand, the probability distribution on $\Delta_k^{(s_i)}$ induced by $\sigma_{1\ldots k}$ is by definition

$$\Pr(D_k | \sigma_{1\ldots k}) = \sum_{s_{1\ldots k} : \mathcal{D}^{(s_i)}(s_{1\ldots k}) = D_k} \prod_{j=1}^{k} \sigma_j(s_j) \tag{9}$$

Now we want to prove that our DP algorithm is indeed computing the correct probability distribution, i.e. $P_k[D_k]$ as defined by Equation 8 is equal to $\Pr(D_k|\sigma_{1\ldots k})$.

Theorem 2. *For all k, and for all $D_k \in \Delta_k^{(s_i)}$, $P_k[D_k] = \Pr(D_k|\sigma_{1\ldots k})$.*

Proof by induction on k. **Base case**: Applying Equation (8) for $k = 1$, it is straightforward to verify that $P_1[D_1] = \Pr(D_1|\sigma_1)$ for all $D_1 \in \Delta_1^{(s_i)}$.

Inductive case: Now assume $P_{k-1}[D_{k-1}] = \Pr(D_{k-1}|\sigma_{1\ldots k-1})$ for all $D_{k-1} \in \Delta_{k-1}^{(s_i)}$.

$$
\begin{aligned}
P_k[D_k] &= \sum_{\substack{D_{k-1}, s_k : \\ \mathcal{D}(D_{k-1}, s_k) = D_k}} P_{k-1}[D_{k-1}] \times \sigma_k(s_k) && (10) \\
&= \sum_{\substack{D_{k-1}, s_k : \\ \mathcal{D}(D_{k-1}, s_k) = D_k}} \sigma_k(s_k) \times \left(\sum_{s_{1\ldots k-1} : \mathcal{D}(s_{1\ldots k-1}) = D_{k-1}} \prod_{j=1}^{k-1} \sigma_j(s_j) \right) && (11) \\
&= \sum_{D_{k-1}, s_k : \mathcal{D}(D_{k-1}, s_k) = D_k} \left(\sum_{s_{1\ldots k-1} : \mathcal{D}(s_{1\ldots k-1}) = D_{k-1}} \prod_{j=1}^{k} \sigma_j(s_j) \right) && (12) \\
&= \sum_{s_{1\ldots k-1}} \sum_{s_k} \sum_{D_{k-1}} \mathbf{1}_{[\mathcal{D}(D_{k-1}, s_k) = D_k]} \cdot \mathbf{1}_{[\mathcal{D}(s_{1\ldots k-1}) = D_{k-1}]} \cdot \prod_{j=1}^{k} \sigma_j(s_j) && (13) \\
&= \sum_{s_{1\ldots k}} \left(\sum_{D_{k-1}} \mathbf{1}_{[\mathcal{D}(D_{k-1}, s_k) = D_k]} \cdot \mathbf{1}_{[\mathcal{D}(s_{1\ldots k-1}) = D_{k-1}]} \right) \cdot \prod_{j=1}^{k} \sigma_j(s_j) && (14) \\
&= \sum_{s_{1\ldots k}} \mathbf{1}_{[\mathcal{D}(s_{1\ldots k}) = D_k]} \prod_{j=1}^{k} \sigma_j(s_j) && (15) \\
&= \sum_{s_{1\ldots k} : \mathcal{D}(s_{1\ldots k}) = D_k} \prod_{j=1}^{k} \sigma_j(s_j) && (16) \\
&= \Pr(D_k | \sigma_{1\ldots k}) && (17)
\end{aligned}
$$

Note that from (13) to (14) we use the fact that given an action profile $s_{1\ldots k-1}$, there is a unique configuration $D_{k-1} \in \Delta_{k-1}^{(s_i)}$ such that $D_{k-1} = \mathcal{D}^{(s_i)}(s_{1\ldots k-1})$.
□

3.4 Complexity

Our algorithm for computing $V_{s_i}^i(\sigma_{-i})$ consists of first computing the projected strategies using (5), then following Algorithm 1, and finally doing the weighted sum given in Equation (6). Let $\Delta^{(s_i,i)}(\sigma_{-i})$ denote the set of configurations over $\nu(s_i)$ that have positive probability of occurring under the mixed strategy (s_i, σ_{-i}). In other words, this is the number of terms we need to add together when doing the weighted sum in Equation (6). When σ_{-i} has full support, $\Delta^{(s_i,i)}(\sigma_{-i}) = \Delta^{(s_i,i)}$. Since looking up an entry in a trie takes time linear in the size of the key, which is $|\nu(s_i)|$ in our case, the complexity of doing the weighted sum in Equation (6) is $O(|\nu(s_i)||\Delta^{(s_i,i)}(\sigma_{-i})|)$.

Algorithm 1 requires n iterations; in iteration k, we look at all possible combinations of $D_{k-1}^{(s_i)}$ and $s_k^{(s_i)}$, and in each case do a trie look-up which costs

$O(|\nu(s_i)|)$. Since $|S_k^{(s_i)}| \leq |\nu(s_i)| + 1$, and $|\Delta_{k-1}^{(s_i)}| \leq |\Delta^{(s_i,i)}|$, the complexity of Algorithm 1 is $O(n|\nu(s_i)|^2|\Delta^{(s_i,i)}(\sigma_{-i})|)$. This dominates the complexity of summing up (6). Adding the cost of computing $\sigma_{-i}^{(s)}$, we get the overall complexity of expected payoff computation $O(n|S| + n|\nu(s_i)|^2|\Delta^{(s_i,i)}(\sigma_{-i})|)$.

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 3. *Given an AGG representation of a game, i's expected payoff $V_{s_i}^i(\sigma_{-i})$ can be computed in polynomial time with respect to the representation size, and if the in-degree of the action graph is bounded by a constant, the complexity is polynomial in n.*

3.5 Discussion

Of course it is not necessary to apply the agents' mixed strategies in the order $1 \ldots n$. In fact, we can apply the strategies in any order. Although the number of configurations $|\Delta^{(s_i,i)}(\sigma_{-i})|$ remains the same, the ordering does affect the intermediate configurations $\Delta_k^{(s_i)}$. We can use the following heuristic to try to minimize the number of intermediate configurations: sort the players by the sizes of their projected action sets, in ascending order. This would reduce the amount of work we do in earlier iterations of Algorithm 1, but does not change the overall complexity of our algorithm.

In fact, we do not even have to apply *one* agent's strategy at a time. We could partition the set of players into sub-groups, compute the distributions induced by each of these sub-groups, then combine these distributions together. Algorithm 1 can be straightforwardly extended to deal with such distributions instead of mixed strategies of single agents. In Section 5.1 we apply this approach to compute Jacobians efficiently.

3.5.1 Relation to Polynomial Multiplication

We observe that the problem of computing $Pr(D|\sigma^{(s_i)})$ can be expressed as one of multiplication of multivariate polynomials. For each action node $s \in \nu(s_i)$, let x_s be a variable corresponding to s. Then consider the following expression:

$$\prod_{k=1}^{n} \left(\sigma_k^{(s_i)}(\emptyset) + \sum_{s_k \in S_k \cap \nu(s_i)} \sigma_k(s_k) x_{s_k} \right) \tag{18}$$

This is a multiplication of n multivariate polynomials, each corresponding to one player's projected mixed strategy. This expression expands to a sum of $|\Delta^{(s_i,i)}|$

terms. Each term can be identified by the tuple of exponents of the x variables, $(D(s), D(s'), \ldots)$. In other words, the set of terms corresponds to the set of configurations $\Delta^{(s_i,i)}$. The coefficient of the term with exponents $D \in \Delta^{(s_i,i)}$ is

$$\sum_{\mathbf{s}^{(s_i)}:\mathcal{D}^{(s_i)}(\mathbf{s}^{(s_i)})=D} \left(\prod_{k=1}^{n} \sigma^{(s_i)}(s_k^{(s_i)}) \right)$$

which is exactly $\Pr(D|\sigma^{(s_i)})$ by Equation (7)! So the whole expression (18) evaluates to

$$\sum_{D\in\Delta^{(s_i,i)}} \Pr(D|\sigma^{(s_i)}) \prod_{s\in\nu(s_i)} x_s^{D(s)}$$

Thus the problem of computing $\Pr(D|\sigma^{(s_i)})$ is equivalent to the problem of computing the coefficients in (18). Our DP algorithm corresponds to the strategy of multiplying one polynomial at a time, i.e. at iteration k we multiply the polynomial corresponding to player k's strategy with the expanded polynomial of $1 \ldots (k-1)$ that we computed in the previous iteration.

4 AGG with Function Nodes

There are games with certain kinds of context-specific independence structures that AGGs are not able to exploit. In Example 4 we show a class of games with one such kind of structure. Our solution is to extend the AGG representation by introducing function nodes, which allows us to exploit a much wider variety of structures.

4.1 Motivating Example: Coffee Shop

Example 4. *In the Coffee Shop Game there are n players; each player is planning to open a new coffee shop in an 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 $n|S|^n = n(B+1)^n$. Since there are no strict independencies in the utility function, the size of the graphical game representation would be similar. 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 and the graphical game 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 and graphical game representations.

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 number of players who chose surrounding blocks, and the number of players who chose 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.

4.2 Function Nodes

In the above example we showed a kind of context-specific independence structure that AGGs cannot exploit. It is easy to think of 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. $S \cap P = \{\}$, so the total number of nodes in G is $|S| + |P|$. 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.

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).

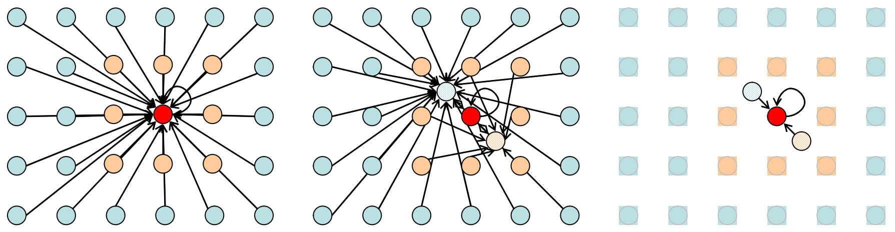

Figure 5: A 5×6 Coffee Shop Game: Left: the AGG representation without function nodes (looking at only the neighborhood of the a node s). Middle: we introduce two function nodes. Right: s now has only 3 incoming edges.

4.3 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 could construct examples with huge number of function nodes, such that the space complexity of representing $\{f_p\}_{p \in P}$ would be greater than that of the utility functions. In other words, blindly adding function nodes will not make the representation more compact. 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 5. 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 nonnegative 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. This is significantly more compact than the AGG representation without function nodes, which had a representation size of $O(B\frac{(n-1+B)!}{(n-1)!B!})$.

Remark 1. One property of the AGG representation as defined in Section 2.1 is that utility function u^s is shared by all players that have s in their action sets. What if we want to represent games with agent-specific utility functions, where utilities depend not only on s and $D^{(s)}$, but also on the identity of the player playing s? We could split s into individual player's actions s_i, s_j etc., so that each action node has its own utility function, however the resulting AGG would not be able to take advantage of the fact that the actions s_i, s_j affect the other players' utilities in the same way. Using function nodes, we are able to compactly represent this kind of structure. We again split s into separate action nodes s_i, s_j, but also introduce a function node p with s_i, s_j as its neighbors, and define f_p to be the summation operator $f_p(D^{(p)}) = \sum_{m \in \nu p} D(m)$. This way the function node p with its configuration $D(p)$ acts as if s_i and s_j had been merged into one node. Action nodes could then include p instead of both s_i and s_j as a neighbor. This way agents can have different utility functions, without sacrificing representational compactness.

4.4 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: $D_k^{(s_i)}(m) = C_{s_k}^{(s_i)}(m) + D_{k-1}^{(s_i)}(m)$ if $m \in S$, and $D_k^{(s_i)}(m) = C_{s_k}^{(s_i)}(m) * D_{k-1}^{(s_i)}(m)$ if $m \in P$. Following similar complexity analysis, if an AGGFN is contribution-independent, 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 $|S|$.

Remark 2. We note that similar ideas are emloyed in the variable elimination algorithms that exploit *causal independence* in Bayes nets [Zhang & Poole, 1996]. Bayes nets are compact representations of probability distributions that graphically represent independencies between random variables. A Bayes net is a DAG where nodes represent random variables and edges represent direct probabilistic dependence between random variables. Efficient algorithms have been developed to compute conditional probabilities in Bayes nets, such as clique tree propagation and variable elimination. Causal independence refers to the situation where a node's parents (which may represent causes) affect the node independently. The conditional probabilities of the node can be defined using a binary operator that can be applied to values from each of the parent variables. Zhang and Poole [1996] proposed a variable elimination algorithm that exploits causal independence by *factoring* the conditional probability distribution into factors corresponding to the causes. The way factors are combined together is similar in spirit to our DP algorithm that combines the independent contributions of the players' strategies to the configuration $D^{(s_i)}$.

This parallel between Bayes nets and action graphs are not surprising. In AGGFNs, we are trying to compute the probability distribution over configurations $\Pr(D^{(s_i)}|\sigma^{(s_i)})$. If we see each node m in the action graph as a random variable $D(m)$, this is the joint distribution of variables $\nu(s_i)$. However, whereas edges in Bayes nets represent probabilistic dependence, edges in the action graph have different semantics depending on the target. Incoming edges of action nodes specifies the neighborhood $\nu(s)$ that we are interested in computing the probabilities of. Incoming edges of a function node represents the deterministic dependence between the random variable of the function node $D(p)$ and its parents. The only probabilistic components of action graphs are the players' mixed strategies. These are probability distributions of random variables associated with players, but are not explicitly represented in the action graph. Whereas AGGFNs in general are not DAGs, given an action s, we can construct an induced Bayes net consisting of $\nu(s)$, the neighbors of function nodes in $\nu(s)$, and the neighbors of any new function nodes included, and so on until no more function nodes are included, and finally augmented with n nodes representing the players' mixed strategies. Whereas for CI AGGFNs, the Bayes net formulation has a simple structure and does not yield a more efficient algorithm compared to Algorithm 1, this formulation could be useful for non-CI AGGFNs with a complex network of function nodes, as standard Bayes net algorithms can be used to exploit the independencies in the induced Bayes net.

5 Applications

5.1 Application: Computing Payoff Jacobian

A game's payoff Jacobian under a mixed strategy σ is defined as a $\sum_i |S_i|$ by $\sum_i |S_i|$ matrix with entries defined as follows:

$$\frac{\partial V^i_{s_i}(\sigma_{-i})}{\partial \sigma_{i'}(s_{i'})} \equiv \nabla V^{i,i'}_{s_i,s_{i'}}(\overline{\sigma}) \tag{19}$$

$$= \sum_{\overline{\mathbf{s}} \in \overline{\mathbf{S}}} u\left(s_i, \mathcal{D}(s_i, s_{i'}, \overline{\mathbf{s}})\right) Pr(\overline{\mathbf{s}}|\overline{\sigma}) \tag{20}$$

Here whenever we use an overbar in our notation, it is shorthand for the subscript $-\{i,i'\}$. For example, $\overline{\mathbf{s}} \equiv \mathbf{s}_{-\{i,i'\}}$. The rows of the matrix are indexed by i and s_i while the columns are indexed by i' and $s_{i'}$. Given entry $\nabla V^{i,i'}_{s_i,s_{i'}}(\overline{\sigma})$, we call s_i its *primary action node*, and $s_{i'}$ its *secondary action node.*

One of the main reasons we are interested in computing Jacobians is that it is the computational bottleneck in Govindan and Wilson's continuation method for finding mixed-strategy Nash equilibria in multi-player games [Govindan & Wilson, 2003]. The Govindan-Wilson algorithm starts by perturbing the payoffs to obtain a game with a known equilibrium. It then follows a path that is guaranteed to give us one or more equilibria of the unperturbed game. In each step, we need to compute the payoff Jacobian under the current mixed strategy in order to get the direction of the path; we then take a small step along the path and repeat.

Efficient computation of the payoff Jacobian is important for more than this continuation method. For example, the iterated polymatrix approximation (IPA) method [Govindan & Wilson, 2004] has the same computational problem at its core. At each step the IPA method constructs a polymatrix game that is a linearization of the current game with respect to the mixed strategy profile, the

Lemke-Howson algorithm is used to solve this game, and the result updates the mixed strategy profile used in the next iteration. Though theoretically it offers no convergence guarantee, IPA is typically much faster than the continuation method. Also, it is often used to give the continuation method a quick start. The payoff Jacobian may also be useful to multiagent reinforcement learning algorithms that perform policy search.

Equation (20) shows that the $\nabla V^{i,i'}_{s_i,s_{i'}}(\overline{\sigma})$ element of the Jacobian can be interpreted as the expected utility of agent i when she takes action s_i, agent i' takes action $s_{i'}$, and all other agents use mixed strategies according to $\overline{\sigma}$. So a straightforward approach is to use our DP algorithm to compute each entry of the Jacobian. However, the Jacobian matrix has certain extra structure that allows us to achieve further speedup.

First, we observe that some entries of the Jacobian are identical. If two entries have same primary action node s, then they are expected payoffs on the same utility function u^s, i.e. they have the same value if their induced probability distributions over $\Delta^{(s)}$ are the same. We need to consider two cases:

1. Suppose the two entries come from the same row of the Jacobian, say player i's action s_i. There are two sub-cases to consider:

   (a) Suppose the columns of the two entries belong to the same player j, but different actions s_j and s'_j. If $s_j^{(s_i)} = s'^{(s_i)}_j$, i.e. s_j and s'_j both project to the same projected action in s_i's projected action graph, then $\nabla V^{i,j}_{s_i,s_j} = \nabla V^{i,j}_{s_i,s'_j}$.

   (b) Suppose the columns of the entries correspond to actions of different players. We observe that for all j and s_j such that $\sigma^{(s_i)}(s_j^{(s_i)}) = 1$, $\nabla V^{i,j}_{s_i,s_j}(\overline{\sigma}) = V^i_{s_i}(\sigma_{-i})$. As a special case, if $S_j^{(s_i)} = \{\emptyset\}$, i.e. agent j does not affect i's payoff when i plays s_i, then for all $s_j \in S_j$, $\nabla V^{i,j}_{s_i,s_j}(\overline{\sigma}) = V^i_{s_i}(\sigma_{-i})$.

2. If s_i and s_j correspond to the same action node s (but owned by agents i and j respectively), thus sharing the same payoff function u^s, then $\nabla V^{i,j}_{s_i,s_j} = \nabla V^{j,i}_{s_j,s_i}$. Furthermore, if there exist $s'_i \in S_i, s'_j \in S_j$ such that $s'^{(s)}_i = s'^{(s)}_j$, then $\nabla V^{i,j}_{s_i,s'_j} = \nabla V^{j,i}_{s_j,s'_i}$.

Even if the entries are not equal, we can exploit the similarity of the projected strategy profiles (and thus the similarity of the induced distributions) between entries, and re-use intermediate results when computing the induced distributions of different entries. Since computing the induced probability distributions is the bottleneck of our expected payoff algorithm, this provides significant speedup.

First we observe that if we fix the row (i, s_i) and the column's player j, then $\overline{\sigma}$ is the same for all secondary actions $s_j \in S_j$. We can compute the probability distribution $\Pr(D_{n-1}|s_i, \overline{\sigma}^{(s_i)})$, then for all $s_j \in S_j$, we just need to apply the action s_j to get the induced probability distribution for the entry $\nabla V^{i,j}_{s_i,s_j}$.

Now suppose we fix the row (i, s_i). For two column players j and j', their corresponding strategy profiles $\sigma_{-\{i,j\}}$ and $\sigma_{-\{i,j'\}}$ are very similar, in fact they are identical in $n-3$ of the $n-2$ components. For AGGs without function nodes, we can exploit this similarity by computing the distribution $\Pr(D_{n-1}|\sigma_{-i}^{(s_i)})$, then for each $j \neq i$, we "undo" j's mixed strategy to get the distribution induced by $\sigma_{-\{i,j\}}$. Recall from Section 3.5.1 that the distributions are coefficients of the multiplication of certain polynomials. So we can undo j's strategy by computing the long division of the polynomial for σ_{-i} by the polynomial for σ_j.

This method does not work for contribution-independent AGGFNs, because in general a player's contribution to the configurations are not reversible, i.e. given $Pr(D_{n-1}|\sigma_{-i}^{(s_i)})$ and σ_j, it is not always possible to undo the contributions of σ_j. Instead, we can efficiently compute the distributions by recursively bisecting the set of players in to sub-groups, computing probability distributions induced by the strategies of these sub-groups and combining them. For example, suppose $n = 9$ and $i = 9$, so $\sigma_{-i} = \sigma_{1\ldots8}$. We need to compute the distributions induced by $\sigma_{-\{1,9\}}, \ldots, \sigma_{-\{8,9\}}$, respectively. Now we bisect σ_{-i} into $\sigma_{1\ldots4}$ and $\sigma_{5\ldots8}$. Suppose we have computed the distributions induced by $\sigma_{1\ldots4}$ as well as $\sigma_{234}, \sigma_{134}, \sigma_{124}, \sigma_{123}$, and similarly for the other group of $5\ldots8$. Then we can compute $\Pr(\cdot|\sigma_{-\{1,9\}}^{(s_i)})$ by combining $\Pr(\cdot|\sigma_{234}^{(s_i)})$ and $\Pr(\cdot|\sigma_{5678}^{(s_i)})$, compute $\Pr(\cdot|\sigma_{-\{2,9\}}^{(s_i)})$ by combining $\Pr(\cdot|\sigma_{134}^{(s_i)})$ and $\Pr(\cdot|\sigma_{5678}^{(s_i)})$, etc. We have reduced the problem into two smaller problems over the sub-groups $1\ldots4$ and $5\ldots8$, which can then be solved recursively by further bisecting the sub-groups. This method saves the re-computation of sub-groups of strategies when computing the induced distributions for each row of the Jacobian, and it works with any contribution-independent AGGFNs because it does not use long division to undo strategies.

6 Experiments

We implemented the AGG representation and our algorithm for computing expected payoffs and payoff Jacobians in C++. We ran several experiments to compare the performance of our implementation against the (heavily optimized) GameTracer implementation [Blum *et al.*, 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.

6.1 Representation Size

For each game instance we counted the number of payoff values that need to be stored in each representation. Since for both normal form and AGG, the size of the representation is dominated by the number of payoff values stored, the number of payoff values is a good indication of the size of the representation.

We first looked at Coffee Shop games with 5×5 blocks, with varying number

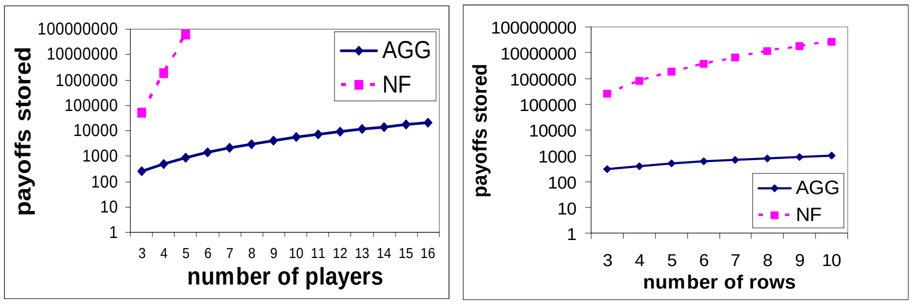

Figure 6: Comparing Representation Sizes of the Coffee Shop Game (log-scale). Left: 5×5 grid with 3 to 16 players. Right: 4-player $r \times 5$ grid with r varying from 3 to 10.

of players. Figure 6 has a log-scale plot of the number of payoff values in each representation versus the number of players. The normal form representation grew exponentially with respect to the number of players, and quickly becomes impractical for large number of players. The size of the AGG representation grew polynomially with respect to n.

We then fixed the number of players at 4, and varied the number of blocks. For ease of comparison we fixed the number of colums at 5, and only changed the number of rows. Figure 6 has a log-scale plot of the number of payoff values versus the number of rows. The size of the AGG representation grew linearly with the number of rows, whereas the size of the normal form representation grew like a higher-order polynomial. This was consistent with our theoretical prediction that AGGFNs store $O(|S|n^3)$ payoff values for Coffee Shop games while normal form representations store $n|S|^n$ payoff values.

6.2 Expected Payoff Computation

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 7 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-

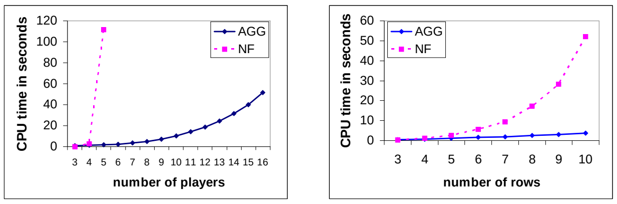

Figure 7: Running times for payoff computation in the Coffee Shop Game. Left: 5×5 grid with 3 to 16 players. Right: 4-player $r \times 5$ grid with r varying from 3 to 10.

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. 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.

6.3 Computing Payoff Jacobians

We have also run similar experiments on computing payoff Jacobians. As discussed in Section 5.1, the entries of a Jacobian can be formulated as expected payoffs, so a Jacobian can be computed by doing an expected payoff computation for each of its entry. In Section 5.1 we discussed methods that exploits the structure of the Jacobian to further speedup the computation. GameTracer's normal-form based implementation also exploits the structure of the Jacobian by re-using partial results of expected-payoff computations. When comparing our AGG-based Jacobian algorithm as described in Section 5.1 against GameTracer's implementation, the results are very similar to the above results for computing expected payoffs, i.e. our implementation scales polynomially in n while GameTracer scales exponentially in n. We instead focus on the question of how much speedup does the methods in Section 5.1 provide, by comparing our algorithm in Section 5.1 against the algorithm that computes expected payoffs (using our AGG-based algorithm described in Section 3) for each of the Jacobian's entries. The results are shown in Figure 8. Our algorithm is about 50 times faster. This confirms that the methods discussed in Seciton 5.1 provide significant speedup for computing payoff Jacobians.

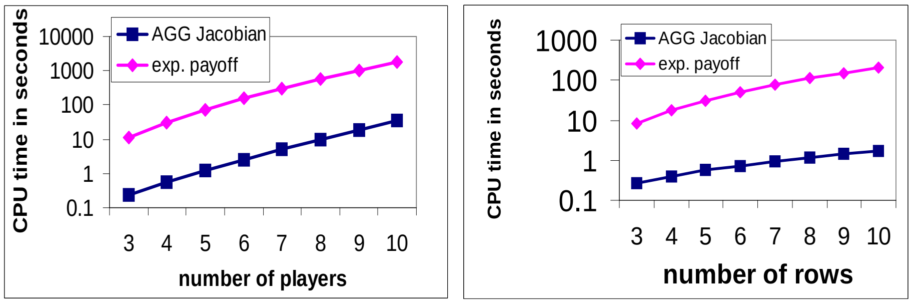

Figure 8: Running times for Jacobian computation in the Coffee Shop Game. Left: 5×5 grid with 3 to 10 players. Right: 4-player $r \times 5$ grid with r varying from 3 to 10.

6.4 Finding Nash Equilibria using the Govindan-Wilson algorithm

Govindan and Wilson's algorithm [Govindan & Wilson, 2003] is one of the most competitive algorithms for finding Nash equilibria for multi-player games. The computational bottleneck of the algorithm is repeated computation of payoff Jacobians as defined in Section 5.1. Now we show experimentally that the speedup we achieved for computing Jacobians using the AGG representation leads to a speedup in the Govidan-Wilson algorithm.

We compared two versions of the Govindan-Wilson algorithm: one is the implementation in GameTracer, where the Jacobian computation is based on the normal form representation; the other is identical to the GameTracer implementation, except that the Jacobians are computed using our algorithm for the AGG representation. Both techniques compute the Jacobians exactly. As a result, given an initial perturbation to the original game, these two implementations would follow the same path and return exactly the same answers. So the difference in their running times would be due to the different speeds of computing Jacobians.

Again, we tested the two algorithms on Coffee Shop games of varying sizes: first we fixed the size of blocks at 4×4 and varied the number of players; then we fixed the number of players at 4 and number of columns at 4, and varied the number of rows. For each game instance, we randomly generated 10 initial perturbation vectors, and for each initial perturbation we run the two versions of the Govindan-Wilson algorithm. Since the running time of the Govindan-Wilson algorithm highly depends on the initial perturbation, it is not meaningful to compare the running times with different initial perturbations. Instead, we look at the *ratio* of running times between the normal form implementation and the AGG implementation. Thus a ratio greater than 1 means the AGG implementation spent less time than the normal form implementation. We plotted the results in Figure 9. The results confirmed our theoretical prediction that as the size of the games grows (either in the number of players or in the

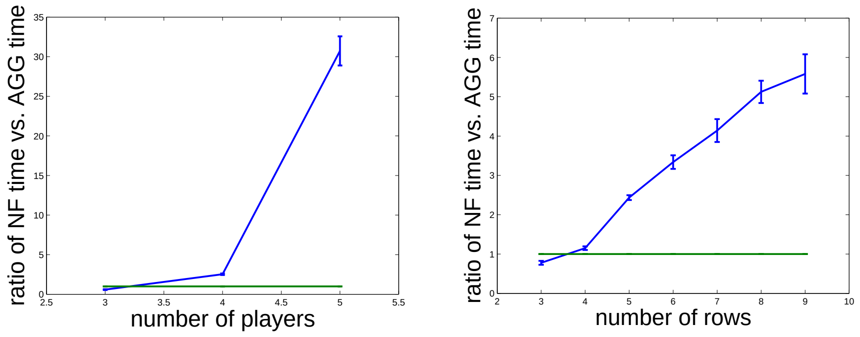

Figure 9: Ratios of Running times for the Govindan-Wilson algorithm in the Coffee Shop Game. Left: 4×4 grid with 3 to 5 players. Right: 4-player $r \times 4$ grid with r varying from 3 to 9. The error bars indicate standard deviation over 10 random initial perturbations. The constant lines at 1.0 indicating equal running times are also shown.

number of actions), the speedup of the AGG implementation compared to the normal from implementation increases.

7 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 and Leyton-Brown [2004]'s algorithm. 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.

Our current and future research includes two directions: Computationally, we plan to apply our expected payoff algorithm to speed up other game-theoretic computations, such as computing best responses and the simplicial subdivision algorithm for finding Nash equilibria. Also, as a direct corollary of our Theorem 3 and Papadimitriou [2005]'s result, correlated equilibria can be computed in time polynomial in the size of the AGG.

Representationally, we plan to extend the AGG framework to represent more types of structure such as additivity of payoffs. In particular, we intend to study is Bayesian games. In a Bayesian game, players are uncertain about which game (i.e. payoff function) they are playing, and each receives certain private information about the underlying game. Bayesian games are heavily used in economics for modeling competitive scenarios involving information asymmetries, e.g. for modeling auctions and other kinds of markets. A Bayesian game can be seen as a compact representation, since it is much more compact than its induced normal form. We plan to use the AGG framework to represent not only the structure inherent in Bayesian games, but also context-specific independence structures such as the ones we have considered here.

References

Bhat, N., & Leyton-Brown, K. (2004). Computing Nash equilibria of action-graph games. *UAI*.

Blum, B., Shelton, C., & Koller, D. (2002). Gametracer. http://dags.stanford.edu/Games/gametracer.html.

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

Govindan, S., & Wilson, R. (2003). A global Newton method to compute Nash equilibria. *Journal of Economic Theory*.

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

Kearns, M., Littman, M., & Singh, S. (2001). Graphical models for game theory. *UAI*.

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

LaMura, P. (2000). Game networks. *UAI*.

Leyton-Brown, K., & Tennenholtz, M. (2003). Local-effect games. *IJCAI*.

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

Porter, R., Nudelman, E., & Shoham, Y. (2004). Simple search methods for finding a Nash equilibrium. *Proc. AAAI* (pp. 664–669).

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., & 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.

Zhang, N., & Poole, D. (1996). Exploiting causal independence in Bayesian network inference. *JAIR*, *5*, 301–328.