Action-Graph Games

Albert Xin Jiang* Kevin Leyton-Brown† Navin A.R. Bhat‡

Abstract

Representing and reasoning with games becomes difficult once they involve large numbers of actions and players, because the space requirement for utility functions can grow unmanageably. Action-Graph Games (AGGs) are a fully-expressive game representation that can compactly express utility functions with structure such as context-specific independence, anonymity, and additivity. We show that AGGs can be used to compactly represent all games that are compact when represented as graphical games, symmetric games, anonymous games, congestion games, and polymatrix games, as well as games that require exponential space under all of these existing representations. We give a polynomial-time algorithm for computing a player's expected utility under an arbitrary mixed-strategy profile, and show how to use this algorithm to achieve exponential speedups of existing methods for computing sample Nash equilibria. We present results of experiments showing that using AGGs leads to a dramatic increase in the size of games accessible to computational analysis.[1]

Keywords: game representations, graphical models, large games, computational techniques, Nash equilibria.

JEL classification codes: C63—Computational Techniques, C72—Noncooperative Games.

1 Introduction

Simultaneous-action games have received considerable study, which is reasonable as these games are in a sense the most fundamental.[2] Most of the game theory literature presumes that simultaneous-action games will be represented in normal form. This is problematic because in many domains of interest the number of players and/or the number of actions per player is large. In the normal form representation, the game's payoff function is stored 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.

Fortunately, most large games of practical interest have highly-structured payoff functions, and thus it is possible to represent them compactly. Intuitively, this helps to explain why people 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 lookup tables. One thread of recent work in the literature has explored game representations that are able to succinctly describe games of interest. In some sense, nearly every game form besides the normal form itself can be seen as such a compact representation. For example, the extensive form allows games with temporal structure to be encoded in exponentially less space than the normal form. In what follows, however, we

*Department of Computer Science, University of British Columbia. jiang@cs.ubc.ca

†Department of Computer Science, University of British Columbia. kevinlb@cs.ubc.ca

‡Department of Physics, University of Toronto (Current affiliation: BMO Capital Markets). nbhat@physics.utoronto.ca

[1]We gratefully acknowledge Moshe Tennenholtz for his co-authorship of a paper on Local Effect Games [Leyton-Brown & Tennenholtz, 2003], an action-centric graphical model for games that inspired our work on AGGs.

[2]More complex games such as those involving time or uncertainty about payoffs can always be mapped to perfect-information, simultaneous-action games by creating an action for every *policy* in the original game. This expansion is of primarily theoretical interest, however, as it tends to cause an explosion in the size of the game.

concentrate on game representations that are compact even for simultaneous-move games of perfect information.

Perhaps the most influential class of compact game representations is that which exploits strict independencies between players' utility functions. This class includes graphical games [Kearns *et al.*, 2001; Kearns, 2007], multi-agent influence diagrams [Koller & Milch, 2003], and game nets [LaMura, 2000]; we focus on the first of these. Consider a graph in which nodes correspond to agents and an edge from one node to another represents the proposition that the first agent is able to affect the second agent's payoff. If every node in the graph has a small in-degree—that is, if each agent's payoff depends only on the actions of a small number of others—then the graphical game representation is compact, by which we mean that it is exponentially smaller than its induced normal form. Of course, there are any number of ways of representing games compactly. For example, games of interest could be assigned short ID numbers. What makes graphical games important is the fact that computational questions about these games can be answered by algorithms whose running time depends on the size of the representation rather than the size of the induced normal form. (Note that this property does not hold for the naive ID number scheme.) To state one fundamental property [Daskalakis *et al.*, 2006a], it is possible to compute an agent's expected utility under an arbitrary mixed strategy profile in time polynomial in the size of the graphical game representation. This property implies that a variety of algorithms for computing game-theoretic quantities of interest, such as sample Nash [Govindan & Wilson, 2003; van der Laan *et al.*, 1987] and correlated equilibrium, can be made exponentially faster for graphical games without introducing any change in the algorithms' behavior or output [Blum *et al.*, 2006; Papadimitriou & Roughgarden, 2008]. Furthermore, graphical games are also computationally well-behaved in other ways; efficient algorithms exist for computing other quantities of interest for certain subclasses of these games such as sample Nash equilibria [Elkind *et al.*, 2006] or Nash equilibria subject to a fairness criterion [Elkind *et al.*, 2007] on path graphs, pure Nash equilibrium on bounded-treewidth graphs [Daskalakis & Papadimitriou, 2006; Gottlob *et al.*, 2005], ϵ-Nash equilibrium [Kearns *et al.*, 2001; Vickrey & Koller, 2002], and evolutionary stable strategies [Kearns & Suri, 2006].

A drawback of the graphical games representation is that it only helps when there exist agents who *never* affect some other agents' utilities. Unfortunately, many games of interest lack any structure of this kind. For example, nontrivial symmetric games are cliques when represented as graphical games. Another useful form of structure not generally captured by graphical games is dubbed *anonymity*; it holds when each agent's utility depends only on the number of agents who took each action, rather than on these agents' identities.[3] Recently, researchers such as Papadimitriou and Roughgarden [2008], Kalai [2005], Daskalakis and Papadimitriou [2007], Brandt *et al.* [2010] and Ryan *et al.* [2010] have explored the representational, computational and strategic benefits that can be derived from symmetry and anonymity assumptions.

A weaker form of utility independence can usefully be combined with symmetry and anonymity. Specifically, utility functions exhibit *context-specific* independencies when the question of whether two agents are able to affect each other's utilities depends on the actions both agents choose. Congestion games [Rosenthal, 1973] are a prominent game representation that can express context-specific payoff independencies, anonymity, *and* symmetry. This representation has many advantages. First and most importantly, many realistic interactions—even involving very large numbers of players and actions—have compact representations as congestion games (see, e.g., [Roughgarden & Tardos, 2002]). Second, congestion games have attractive theoretical properties.

[3]Note that our definition of anonymity presumes that it makes sense to speak about two different agents having at least some of the same action choices. There are various ways of achieving this formally; for now, one can simply assume that anonymous games are also symmetric.

Most notably, they always have pure-strategy equilibria, and indeed always admit an exact potential function [Monderer & Shapley, 1996]. As a consequence, simple best-response dynamics are guaranteed to converge to a pure-strategy equilibrium. Finally, congestion games have attractive computational properties. For example, correlated equilibrium can be efficiently computed for congestion games [Papadimitriou, 2005; Papadimitriou & Roughgarden, 2008], and pure-strategy Nash equilibrium can be efficiently computed for restricted subclasses of congestion games (see, e.g., [Ieong *et al.*, 2005]).

Unfortunately, congestion games too have a catch. Unlike graphical games, congestion games are not a universal game representation: not every normal-form game can be encoded as a congestion game. Indeed, this problem should be obvious from the fact that congestion games always have pure-strategy equilibria. Congestion games require that agents' utility functions must be expressible as a *sum* of arbitrary functions of the numbers of agents who chose each of a set of resources, where each action is interpreted as the choice of one or more resources. This linearity assumption is restrictive. Thus, while congestion games constitute a useful model for reasoning about certain game-theoretic domains, they cannot serve as the basis for a set of general tools for representing and reasoning about games.

Action-graph games (AGGs) are a general game representation that can be understood as offering the advantages of—and, indeed, unifying—both graphical games and congestion games. Like graphical games, AGGs can represent any game, and important game-theoretic computations can be performed efficiently when the AGG representation is compact. Hence, AGGs offer a general representational framework for game-theoretic computation. Like congestion games, AGGs compactly represent context-specific independence, anonymity, and additivity, though unlike congestion games they do not require any of these. Finally, AGGs can also compactly represent many games that are not compact as either graphical games or as congestion games.

We begin this paper in Section 2 by defining action-graph games, including the basic representation and extensions with function nodes and additive utility functions, and characterizing their representation sizes. In Section 3 we provide several more examples of structured games which can be compactly represented as AGGs. Then we turn from representational to computational issues. In Section 4 we present a dynamic programming algorithm for computing an agent's expected utility under an arbitrary mixed-strategy profile, prove its complexity, and explore several elaborations. In Section 5 we show that (as a corollary of the polynomial complexity of our expected utility algorithm) the problem of finding an ϵ-Nash equilibrium of an AGG is in PPAD: a positive result, as AGGs can be exponentially smaller than normal-form games. We also show how to use our dynamic programming algorithm to speed up existing methods for computing sample ϵ-Nash and ϵ-correlated equilibria. Finally, in Section 6 we present the results of extensive experiments with some of these algorithms, demonstrating that AGGs can feasibly be used to reason about interesting games that were inaccessible to any previous techniques. The largest game that we tackled in our experiments had 20 agents and 13 actions per agent; we found its Nash equilibrium in 14.3 minutes. A normal form representation of this game would have involved 9.4×10^{134} numbers, requiring an outrageous 7.5×10^{126} gigabytes even to store.

Finally, let us describe the relationship between this paper and past work, mostly our own, on AGGs. Leyton-Brown and Tennenholtz [2003] introduced local-effect games, which can be understood as symmetric AGGs in which utility functions are required to satisfy a particular linearity property. Bhat and Leyton-Brown [2004] introduced the basic AGG representation and some of the computational ideas for reasoning with them. The dynamic programming algorithm was first proposed in Jiang and Leyton-Brown [2006], as was the idea of function nodes. The current paper substantially elaborates upon and extends the representations and methods from these two papers. Other new material includes the additive structure model and the encoding of

congestion games, several of the examples, our computational methods for k-symmetric games and for additive structure, and our speedup of the simplicial subdivision algorithm. Furthermore, all experiments in this paper (Section 6) are new. Going beyond the work described here, in Jiang and Leyton-Brown [2007] we gave a message-passing algorithm for computing pure-strategy equilibria of symmetric AGGs, in Thompson *et al.* [2007] we explored the use of AGGs to model network congestion problems that cannot be captured as congestion games, in Thompson and Leyton-Brown [2009] we used AGGs to compute the Nash equilibria of perfect-information advertising auction problems, and in Jiang *et al.* [2009] and Jiang and Leyton-Brown [2010] we extend our AGG framework to represent dynamic games and Bayesian games, respectively. Daskalakis *et al.* [2009] (a separate group of researchers) recently considered the computation of ϵ-Nash equilibrium of AGGs, providing a fully polynomial time approximation scheme (FPTAS) for one family of AGGs and proving computational hardness results for other families.

2 Action Graph Games

This section has three parts, each of which defines a different AGG variant. In Section 2.1 we define the basic AGG representation (which we dub AGG-$\emptyset$), characterize its representation size, and show how it can be used to represent normal-form, graphical, and symmetric games. In Section 2.2 we introduce the idea of *function nodes*, show how AGGs with function nodes (AGG-FNs) can capture additional structure in several example games, and show how to represent anonymous games as AGG-FNs. In Section 2.3 we introduce AGG-FNs with additive structure (AGG-FNA), which compactly represent additive structure in the utility functions of AGGs, and show how congestion games can be succinctly written as AGG-FNAs.

2.1 Basic Action Graph Games

We begin with an intuitive description of basic action-graph games. Consider a directed graph with nodes $\mathcal{A}$ and edges E, and a set of agents $N = \{1, \ldots, n\}$. Identical tokens are given to each agent $i \in N$. To play the game, each agent i simultaneously places her token on a node $a_i \in A_i$, where $A_i \subseteq \mathcal{A}$. Each node in the graph thus corresponds to an action choice that is available to one or more of the agents; this is where action-graph games get their name. Each agent's utility is calculated according to an arbitrary function of the node she chose and the *numbers* of tokens placed on the nodes that neighbor that chosen node in the graph. We will argue below that any simultaneous-move game can be represented in this way, and that action-graph games are often much more compact than games represented in other ways.

We now turn to a formal definition of basic action-graph games. Let $N = \{1, \ldots, n\}$ be the set of agents. Central to our model is the *action graph*.

Definition 2.1 (Action graph) *An* action graph $G = (\mathcal{A}, E)$ *is a directed graph where:*

- *$\mathcal{A}$ is the set of nodes. We call each node $\alpha \in \mathcal{A}$ an* action*, and $\mathcal{A}$ the* set of distinct actions*. For each agent $i \in N$, let A_i be the set of actions available to i, with $\mathcal{A} = \bigcup_{i \in N} A_i$.*[4] *We denote by $a_i \in A_i$ one of agent i's actions. An* action profile *(or* pure strategy profile*) is a tuple $a = (a_1, \ldots, a_n)$. Denote by A the set of action profiles. Then $A = \prod_{i \in N} A_i$ where $\prod$ is the Cartesian product.*

[4]Different agents' action sets A_i, A_j may (partially or completely) overlap. The implications of this will become clear once we define the utility functions.

- *E is a set of directed edges, where self edges are allowed. We say α' is a* neighbor *of α if there is an edge from α' to α, i.e., $(\alpha', \alpha) \in E$. Let the* neighborhood *of α, denoted $\nu(\alpha)$, be the set of neighbors of α, i.e., $\nu(\alpha) \equiv \{\alpha' \in \mathcal{A} | (\alpha', \alpha) \in E\}$.*

Given an action graph and a set of agents, we can further define a *configuration*, which is a feasible arrangement of agents across nodes in an action graph.

Definition 2.2 (Configuration) *Given an action graph $(\mathcal{A}, E)$ and a set of action profiles A, a* configuration *c is a tuple of $|\mathcal{A}|$ non-negative integers $(c(\alpha))_{\alpha \in \mathcal{A}}$, where $c(\alpha)$ is interpreted as the number of agents who chose action $\alpha \in \mathcal{A}$, and where there exists some $a \in A$ that would give rise to c. Denote the set of all configurations as C. Let $\mathcal{C} : A \to C$ be the function that maps from an action profile a to the corresponding configuration c. Formally, if $c = \mathcal{C}(a)$ then $c(\alpha) = |\{i \in N : a_i = \alpha\}|$ for all $\alpha \in \mathcal{A}$.*

We can also restrict a configuration to a given node's neighborhood.

Definition 2.3 (Configuration over a neighborhood) *Given a configuration $c \in C$ and a node $\alpha \in \mathcal{A}$, let the* configuration over the neighborhood *of α, denoted $c^{(\alpha)}$, be the restriction of c to $\nu(\alpha)$, i.e., $c^{(\alpha)} = (c(\alpha'))_{\alpha' \in \nu(\alpha)}$. Similarly, let $C^{(\alpha)}$ denote the set of configurations over $\nu(\alpha)$ in which at least one player plays α.*[5] *Let $\mathcal{C}^{(\alpha)} : A \to C^{(\alpha)}$ be the function which maps from an action profile to the corresponding configuration over $\nu(\alpha)$.*

Now we can state the formal definition of basic action-graph games as follows.

Definition 2.4 (Basic action-graph game) *A basic action-graph game (AGG-$\emptyset$) is a tuple (N, A, G, u) where*

- *N is the set of agents;*
- *$A = \prod_{i \in N} A_i$ is the set of action profiles;*
- *$G = (\mathcal{A}, E)$ is an action graph, where $\mathcal{A} = \bigcup_{i \in N} A_i$ is the set of distinct actions;*
- *u is a tuple $(u^\alpha)_{\alpha \in \mathcal{A}}$, where each $u^\alpha : C^{(\alpha)} \to \mathbb{R}$ is the* utility function *for action α. Semantically, $u^\alpha(c^{(\alpha)})$ is the utility of an agent who chose α, when the configuration over $\nu(\alpha)$ is $c^{(\alpha)}$.*

For notational convenience, we define $u(\alpha, c^{(\alpha)}) \equiv u^\alpha(c^{(\alpha)})$ and $u_i(a) \equiv u(a_i, \mathcal{C}^{(a_i)}(a))$. We also define $A_{-i} \equiv \prod_{j \neq i} A_j$ as the set of action profiles of agents other than i, and denote an element of A_{-i} by a_{-i}.

2.1.1 Example: Ice Cream Vendors

The following example helps to illustrate the elements of the AGG-$\emptyset$ representation, and also exhibits context-specificity and anonymity in utility functions. This example would not be compact under the existing game representations discussed in the introduction. It was inspired by Hotelling [1929], and elaborates an example used in Leyton-Brown and Tennenholtz [2003].

[5] If action α 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 α (denoted $C^{(s,i)}$) is different from the set of configurations given that j played α. $C^{(\alpha)}$ is the union of these sets of configurations.

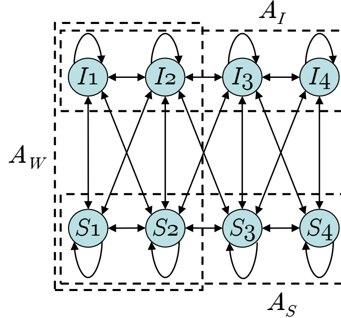

Figure 1: AGG-∅ representation of the Ice Cream Vendor game.

Example 2.5 (Ice Cream Vendor game) *Consider a setting in which n vendors sell ice cream or strawberries, and must choose one of four locations along a beach. There are three kinds of vendors: n_I ice cream vendors, n_S strawberry vendors, and n_W vendors who can sell both ice cream and strawberry, but only on the west side. Ice cream (strawberry) vendors are negatively affected by the presence of other ice cream (strawberry) vendors in the same or neighboring locations, and are simultaneously positively affected by the presence of nearby strawberry (ice cream) vendors.*

The AGG-∅ representation of this game is illustrated in Figure 1. As always, nodes represent actions and directed edges represent membership in a node's neighborhood. The dotted boxes represent the action sets for each group of players; for example, the ice cream vendors have action set A_I. Note that this game exhibits context-specific independence without any strict independence, and that the graph structure is independent of n.

2.1.2 Size of an AGG-∅ Representation

Intuitively, AGG-∅s capture two types of structure in games:

1. Shared actions capture the game's *anonymity* structure: agent i's utility depends only on her action a_i and the configuration. Thus, agent i cares about the *number* of players that play each action, but not the identities of those players.
2. The (lack of) edges between nodes in the action graph expresses *context-specific independencies* of utilities of the game: for all $i \in N$, if i chose action $\alpha \in \mathcal{A}$, then i's utility depends only on the configuration over the neighborhood of α. In other words, the configuration over actions not in $\nu(\alpha)$ does not affect i's utility.

We have claimed informally that action graph games provide a way of representing games compactly. But what exactly is the size of an AGG-∅ representation, and how does it grow with the number of agents n? In this subsection we give a bound on the size of an AGG-∅, and show that asymptotically it is never worse than the size of the equivalent normal form.

From Definition 2.4 we observe that to completely specify an AGG-∅ we need to specify (1) the set of agents, (2) each agent's set of actions, (3) the action graph, and (4) the utility functions. The first three can easily be compactly represented:

1. The set of agents $N = \{1, \ldots, n\}$ can be specified by the integer n.
2. The set of actions $\mathcal{A}$ can be specified by the integer $|\mathcal{A}|$. Each agent's action set $A_i \subseteq \mathcal{A}$ can be specified in $O(|\mathcal{A}|)$ space.

3. The action graph $G = (\mathcal{A}, E)$ can be straightforwardly represented as neighbor lists: for each node $\alpha \in \mathcal{A}$ we specify its list of neighbors $\nu(\alpha) \subseteq \mathcal{A}$. The space required is $\sum_{\alpha \in \mathcal{A}} |\nu(\alpha)|$, which is bounded by $|\mathcal{A}|\mathcal{I}$, where $\mathcal{I} = \max_\alpha |\nu(\alpha)|$, i.e., the maximum in-degree of G.

We observe that whereas the first three components of an AGG-$\emptyset$ (N, A, G, u) can always be represented in space polynomial in n and $|A_i|$, the size of the utility functions is worst-case exponential. So the size of the utility functions determines whether an AGG-$\emptyset$ can be tractably represented. Indeed, for the rest of the paper we will refer to the number of payoff values stored as the representation size of the AGG-$\emptyset$. The following proposition gives an upper bound on the number of payoff values stored.

Proposition 2.6 *Given an AGG-$\emptyset$, the number of payoff values stored by its utility functions is at most $|\mathcal{A}|\frac{(n-1+\mathcal{I})!}{(n-1)!\mathcal{I}!}$. If $\mathcal{I}$ is bounded by a constant as n grows, the number of payoff values is $O(|\mathcal{A}|n^{\mathcal{I}})$, i.e. polynomial with respect to n.*

Proof. For each utility function $u^\alpha : C^{(\alpha)} \to \mathbb{R}$, we need to specify a utility value for each distinct configuration $c^{(\alpha)} \in C^{(\alpha)}$. The set of configurations $C^{(\alpha)}$ can be derived from the action graph, and can be sorted in lexicographical order. Thus, we can just specify a list of $|C^{(\alpha)}|$ utility values that correspond to the (ordered) set of configurations.[6] In general there is no closed form expression for $|C^{(\alpha)}|$, the number of distinct configurations over $\nu(\alpha)$. Instead, we consider the operation of extending all agents' action sets via $\forall i : A_i \mapsto \mathcal{A}$. The number of configurations over $\nu(\alpha)$ under the new action sets is an upper bound on $|C^{(\alpha)}|$. This is the number of (ordered) combinatorial compositions of $n-1$ (since one player has already chosen α) into $|\nu(\alpha)|+1$ nonnegative integers, which is $\binom{n-1+|\nu(\alpha)|}{|\nu(\alpha)|} = \frac{(n-1+|\nu(\alpha)|)!}{(n-1)!|\nu(\alpha)|!}$. Then the total space required for the utilities is bounded from above by $|\mathcal{A}|\frac{(n-1+\mathcal{I})!}{(n-1)!\mathcal{I}!}$. If $\mathcal{I}$ is bounded by a constant as n grows, this grows like $O(|\mathcal{A}|n^{\mathcal{I}})$. ■

For each AGG-$\emptyset$, there exists a unique *induced normal form* representation with the same set of players and $|A_i|$ actions for each i; its utility function is a matrix that specifies each player i's payoff for each possible action profile $a \in A$. This implies a space complexity of $n\prod_{i=1}^n |A_i|$. When $A_i \geq 2$ for all i, the size of the induced normal form representation grows exponentially with respect to n. On the other hand, we observe that the number of payoff values stored in an AGG-$\emptyset$ representation is always less than or equal to the number of payoff values in the induced normal form representation. Of course, the AGG-$\emptyset$ representation has the extra overhead of representing the action graph, which is bounded by $|\mathcal{A}|\mathcal{I}$. But this overhead is dominated by the size of the induced normal form, $n\prod_j |A_j|$. Thus, an AGG-$\emptyset$'s asymptotic space complexity is never worse than that of its induced normal form game.

It is also possible to describe a reverse transformation that encodes any arbitrary game in normal form as an AGG-$\emptyset$. Specifically, a unique node a_i must be created for each action available to each agent i. Thus $\forall \alpha \in \mathcal{A},\ c(\alpha) \in \{0, 1\}$, and $\forall i,\ \sum_{\alpha \in A_i} c(\alpha)$ must equal 1. The configuration simply indicates each agent's action choice, and expresses no anonymity or context-specific independence structure.

This representation is no more or less compact than the normal form. More precisely, the number of distinct configurations over $\nu(a_i)$ is the number of action profiles of the other players,

[6]This is the most compact way of representing the utility functions, but does not provide easy random access to the utilities. Therefore, when we want to do computation using AGGs, we may convert each utility function u^α 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 $O(\mathcal{I}|C^{(\alpha)}|)$.

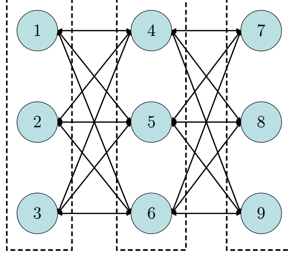

Figure 2: AGG-$\emptyset$ representation of a 3-player, 3-action graphical game.

which is $\prod_{j \neq i} |A_j|$. Since i has $|A_i|$ actions, $\prod_j |A_j|$ payoff values are needed to represent i's payoffs. So in total $n \prod_j |A_j|$ payoff values are stored, exactly the number in the normal form.

One might ask whether AGG-$\emptyset$s can compactly represent known classes of structured games. Consider the graphical game representation [Kearns *et al.*, 2001]. In a graphical game nodes denote agents, and there is an (undirected) edge connecting each agent i to each other agent whose actions can affect i's utility. Each agent then has a payoff matrix representing his local game with neighboring agents. Graphical games can be represented as AGG-$\emptyset$s by replacing each node i in the graphical game by a distinct cluster of nodes A_i representing the action set of agent i. If the graphical game has an edge from i to j, edges must be created in the AGG-$\emptyset$ so that $\forall a_i \in A_i, \forall a_j \in A_j,\ a_i \in \nu(a_j)$. The resulting AGG-$\emptyset$s are as compact as the original graphical games. Figure 2 shows the AGG-$\emptyset$ representation of a graphical game having three nodes and two edges (i.e., player 1 and player 3 do not directly affect each others' payoffs).

Another important class of structured games are symmetric games. A symmetric game is one in which all players are identical and indistinguishable. Symmetric games exhibit anonymity structure: the utility of a player who chose a certain action depends only on the numbers of players who played each of the actions. An arbitrary symmetric game can be encoded as an AGG-$\emptyset$ without an increase in asymptotic size. Specifically, let $A_i = \mathcal{A}$ for all $i \in N$. The resulting action graph is a clique, i.e. $\nu(\alpha) = \mathcal{A}$ for all $\alpha \in \mathcal{A}$.

2.2 AGGs with Function Nodes

There are games with certain kinds of context-specific independence structures that AGG-$\emptyset$s are not able to exploit (see, e.g., Example 2.7 below). In this section we extend the AGG-$\emptyset$ representation by introducing *function nodes*, allowing us to exploit a much wider variety of utility structures. Of course, as always, compact representation is not interesting as an end in itself. In Section 4.2 we identify broad subclasses of AGG-FNs—indeed, rich enough to encompass all AGG-FN examples presented in this paper—which are amenable to efficient computation.

2.2.1 Examples: Coffee Shops and Parity

Example 2.7 (Coffee Shop game) *Consider a game involving n players; each player plans to open a coffee shop in a downtown area, represented by a $r \times k$ grid. Each player can choose to open a shop located within any of the $B \equiv rk$ blocks or decide not to enter the market. Conditioned on player i choosing some location α, her utility depends on the numbers of players who chose (i) the same block; (ii) any of the surrounding blocks; and (iii) any other location.*

The normal form representation of this game has size $n|\mathcal{A}|^n = n(B+1)^n$. Since there are no strict independencies in the utility function, the asymptotic size of the graphical game

representation is the same. Let us now represent the game as an AGG-$\emptyset$. We observe that if agent i chooses an action α corresponding to one of the B locations, then her payoff is affected by the configuration over all B locations. Hence, $\nu(\alpha)$ must consist of B action nodes corresponding to the B locations, and so the action graph has in-degree $\mathcal{I} = B$. Since the action sets completely overlap, the representation size is $\Theta(|\mathcal{A}||C^{(\alpha)}|) = \Theta\left(B\frac{(n-1+B)!}{(n-1)!B!}\right)$. If we hold B constant, this becomes $\Theta(Bn^B)$, 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 $\Theta(B^n)$, which is only slightly better than the normal form and graphical game representations.

Intuitively, the AGG-$\emptyset$ representation is able to exploit anonymity structure in this game. However, this game's payoff function also has context-specific structure that the AGG-$\emptyset$ does not capture. Observe that u^α depends only on three quantities: the number of players who chose the same block, the number of players who chose an adjacent block, and the number of players who chose another location. In other words, u^α can be written as a function g of only three integers: $u^\alpha(c^{(\alpha)}) = g(c(\alpha), \sum_{\alpha' \in \mathcal{A}'} c(\alpha'), \sum_{\alpha'' \in \mathcal{A}''} c(\alpha''))$ where $\mathcal{A}'$ is the set of actions surrounding α and $\mathcal{A}''$ the set of actions corresponding to other locations. The AGG-$\emptyset$ representation is not able to exploit this context-specific information, and so duplicates some utility values.

There exist many similar examples in which the utility functions u^α can be expressed as functions of a small number of intermediate parameters. Here we give one more.

Example 2.8 (Parity game) *In a "parity game", each u^α depends only on whether the number of agents at neighboring nodes is even or odd, as follows:*

$$u^\alpha = \begin{cases} 1 & \textit{if } \sum_{\alpha' \in \nu(\alpha)} c(\alpha') \mod 2 = 0; \\ 0 & \textit{otherwise.} \end{cases}$$

Observe that in the Parity game u^α can take just two distinct values; however, the AGG-$\emptyset$ representation must specify a value for every configuration $c^{(\alpha)}$.

2.2.2 Definition of AGG-FNs

Structure such as that in Examples 2.7 and 2.8 can be exploited within the AGG framework by introducing *function nodes* to the action graph G; intuitively, we use them to describe intermediate parameters upon which players' utilities depend. Now G's vertices consist of both the set of action nodes $\mathcal{A}$ and the set of function nodes $\mathcal{P}$, i.e. $G = (\mathcal{A} \cup \mathcal{P}, E)$. We require that no function node $p \in \mathcal{P}$ can be in any player's action set: $\mathcal{A} \cap \mathcal{P} = \{\}$. Thus, the total number of nodes in G is $|\mathcal{A}| + |\mathcal{P}|$. Each node in G can have action nodes and/or function nodes as neighbors. We associate a function $f^p : C^{(p)} \to \mathbb{R}$ with each $p \in \mathcal{P}$, where $c^{(p)} \in C^{(p)}$ denotes configurations over p's neighbors. The configurations c are extended to include the function nodes by the definition $c(p) \equiv f^p(c^{(p)})$. If $p \in \mathcal{P}$ has no neighbors, f^p is a constant function. To ensure that the AGG is meaningful, the graph G restricted to nodes in $\mathcal{P}$ is required to be a directed acyclic graph (DAG). This condition ensures that for all α and p, $c(\alpha)$ and $c(p)$ are well defined. To ensure that every $p \in \mathcal{P}$ is "useful", we also require that p has at least one outgoing edge. As before, for each action node α we define a utility function $u^\alpha : C^{(\alpha)} \to \mathbb{R}$. We call this extended representation an Action Graph Game with Function Nodes (AGG-FN), and define it formally as follows.

Definition 2.9 (AGG-FN) *An Action Graph Game with Function Nodes (AGG-FN) is a tuple $(N, A, \mathcal{P}, G, f, u)$, where:*

- *N is the set of agents;*

- $A = \prod_{i \in N} A_i$ *is the set of action profiles;*
- $\mathcal{P}$ *is a finite set of function nodes;*
- $G = (\mathcal{A} \cup \mathcal{P}, E)$ *is an action graph, where* $\mathcal{A} = \bigcup_{i \in N} A_i$ *is the set of distinct actions. We require that the restriction of* G *to the nodes* $\mathcal{P}$ *is acyclic and that for every* $p \in \mathcal{P}$ *there exists an* $m \in \mathcal{A} \cup \mathcal{P}$ *such that* $(p, m) \in E$*;*
- f *is a tuple* $(f^p)_{p \in \mathcal{P}}$*, where each* $f^p : C^{(p)} \to \mathbb{R}$ *is an arbitrary mapping from neighbors of* p *to real numbers;*
- u *is a tuple* $(u^\alpha)_{\alpha \in \mathcal{A}}$*, where each* $u^\alpha : C^{(\alpha)} \to \mathbb{R}$ *is the* utility function *for action* α*.*

Given an AGG-FN, we can construct an equivalent AGG-$\emptyset$ with the same players N and actions $\mathcal{A}$ and equivalent utility functions, but without any function nodes. We call this the *induced AGG-*$\emptyset$ of the AGG-FN. There is an edge from α' to α in the induced AGG-$\emptyset$ either if there is an edge from α' to α in the AGG-FN, or if there is a path from α' to α through a chain consisting entirely of function nodes. From the definition of AGG-FNs, the utility of playing action α is uniquely determined by the configuration $c^{(\alpha)}$, which is uniquely determined by the configuration over the actions that are neighbors of α in the induced AGG-$\emptyset$. As a result, the utility tables of the induced AGG-$\emptyset$ can be filled in unambiguously. We observe that the number of utility values stored in an AGG-FN is no greater than the number of utility values in the induced AGG-$\emptyset$. On the other hand, AGG-FNs have to represent the functions f^p for each $p \in \mathcal{P}$. In the worst case, these functions can be represented as explicit mappings similar to the utility functions u^α. However, it is often possible to define these functions algebraically by combining elementary operations, as we do in most of the examples given in this paper. In this case the functions' representations require a negligible amount of space.

2.2.3 Representation Size

What is the size of an AGG-FN $(N, A, \mathcal{P}, G, f, u)$? The following proposition gives a sufficient condition for the representation size to be polynomial. Here we speak about a *class* of AGG-FNs because our statement is about the asymptotic behavior of the representation size. This is in contrast to Proposition 2.6, where we gave an exact bound on the size of an individual AGG-$\emptyset$.

Proposition 2.10 *A class of AGG-FNs has representation size bounded by a function polynomial in* n*,* $|\mathcal{A}|$ *and* $|\mathcal{P}|$ *if the following conditions hold:*

1. *for all function nodes* $p \in \mathcal{P}$*, the size of* p*'s range* $|\mathcal{R}(f^p)|$ *is bounded by a function polynomial in* n*,* $|\mathcal{A}|$ *and* $|\mathcal{P}|$*; and*
2. $\max_{m \in A \cup \mathcal{P}} \nu(m)$ *(the maximum in-degree in the action graph) is bounded by a constant.*

Proof. Given an AGG-FN $(N, A, \mathcal{P}, G, f, u)$, it is straightforward to check that all components except u and f are polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$.

First, consider an action node $\alpha \in \mathcal{A}$. Recall that the size of the utility function u^α is $C^{(\alpha)}$. Partition $\nu(\alpha)$, the set of α's neighbors, into $\nu_{\mathcal{A}}(\alpha) = \nu(\alpha) \cap \mathcal{A}$ and $\nu_{\mathcal{P}}(\alpha) = \nu(\alpha) \cap \mathcal{P}$ (neighboring action nodes and function nodes respectively). Since for each action $\alpha' \in \nu_{\mathcal{A}}(\alpha)$, $c(\alpha') \in \{0, \ldots, n\}$, and for each $p' \in \nu_{\mathcal{P}}(\alpha)$, $c(p) \in \mathcal{R}(f^p)$, then $C^{(\alpha)} \le (n+1)^{|\nu_{\mathcal{A}}(\alpha)|} \prod_{p \in \nu_{\mathcal{P}}(\alpha)} |\mathcal{R}(f^p)|$. This is polynomial because all action node in-degrees are bounded by a constant.

Now consider a function node $p \in \mathcal{P}$. Without loss of generality, assume that its function f^p is represented explicitly as a mapping. (Any other representation of f^p can be transformed

into this explicit representation.) The representation size of f^p is then $C^{(p)}$. Using the same reasoning as above, we have $C^{(p)} \leq (n+1)^{|\nu_{\mathcal{A}}(p)|} \prod_{q \in \nu_{\mathcal{P}}(p)} |\mathcal{R}(f^q)|$, which is polynomial since all function node in-degrees are bounded by a constant. ■

When the functions f^p do not have to be represented explicitly, we can drop the requirement on the in-degree of function nodes.

Corollary 2.11 *A class of AGG-FNs has representation size bounded by a function polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$ if the following conditions hold:*

1. *for all function nodes $p \in \mathcal{P}$, the function f^p has a representation whose size is polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$;*
2. *for each function node $p \in \mathcal{P}$ that is a neighbor of some action node α, the size of p's range $|\mathcal{R}(f^p)|$ is bounded by a function polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$; and*
3. $\max_{\alpha \in A} \nu(\alpha)$ *(the maximum in-degree among action nodes) is bounded by a constant.*

A very useful type of function node is the *simple aggregator*.

Definition 2.12 (Simple aggregator) *A function node $p \in \mathcal{P}$ is a* simple aggregator *if each of its neighbors $\nu(p)$ are action nodes and f^p is the summation function: $f^p(c^{(p)}) = \sum_{m \in \nu(p)} c(m)$.*

Simple aggregator function nodes take the value of the total number of players who chose any of the node's neighbors. Since these functions can be specified in constant space, and since $\mathcal{R}(f^p) = \{0, \ldots, n\}$ for all p, Corollary 2.11 applies. That is, the representation sizes of AGG-FNs whose function nodes are all simple aggregators are polynomial whenever the in-degrees of action nodes are bounded by a constant. In fact, under certain assumptions we can prove an even tighter bound on the representation size, analogous to Proposition 2.6 for AGG-$\emptyset$s. Intuitively, this works because both configurations on action nodes and configurations on simple aggregators count the numbers of players who behave in certain ways.

Proposition 2.13 *Consider a class of AGG-FNs whose function nodes are all simple aggregators. For each $m \in \mathcal{A} \cup \mathcal{P}$, define the function*

$$\beta(m) = \begin{cases} m & m \in A; \\ \nu(m) & \text{otherwise.} \end{cases}$$

Intuitively, $\beta(m)$ is the set of nodes whose counts are aggregated by node m. If for each $\alpha \in \mathcal{A}$ and for each $m, m' \in \nu(\alpha)$, $\beta(m) \cap \beta(m') = \{\}$ unless $m = m'$ (i.e., no action node affects α in more than one way), then the AGG-FNs' representation sizes are bounded by $|\mathcal{A}|\binom{n-1+\mathcal{I}}{\mathcal{I}}$ where $\mathcal{I} = \max_{\alpha \in A} |\nu(\alpha)|$ is the maximum in-degree of action nodes.

Proof. Consider the utility function u^α for an arbitrary action α. Each neighbor $m \in \nu(\alpha)$ is either an action or a simple aggregator. Observe that a configuration $c^{(\alpha)} \in C^{(\alpha)}$ is a tuple of integers specifying the numbers of players choosing each action in the set $\beta(m)$ for each $m \in \nu(\alpha)$. As in the proof of Proposition 2.6, we extend each player's set of actions to $|\mathcal{A}|$, making the game symmetric. This weakly increases the number of configurations. Since the sets $\beta(m)$ are non-overlapping, the number of configurations possible in the extended action space is equal to the number of (ordered) combinatorial compositions of $n-1$ into $|\nu(\alpha)|+1$ nonnegative integers, which is $\binom{n-1+|\nu(\alpha)|}{|\nu(\alpha)|}$. This includes one bin for each action or simple aggregator in $\nu(\alpha)$, plus one bin for agents that take an action that is neither in $\nu(\alpha)$ nor in the neighborhood of any simple aggregator in $\nu(\alpha)$. Then the total space required for representing u is bounded by $|\mathcal{A}|\binom{n-1+\mathcal{I}}{\mathcal{I}}$ where $\mathcal{I} = \max_{\alpha \in A} |\nu(\alpha)|$. ■

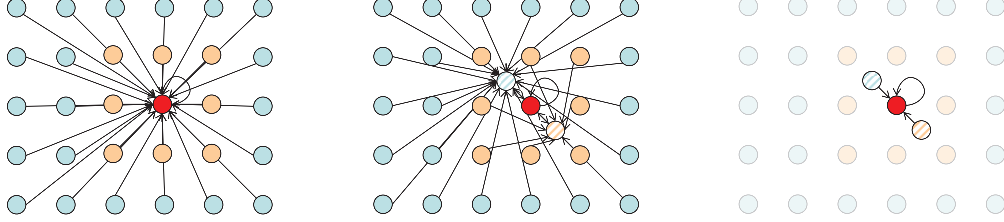

Figure 3: A 5×6 Coffee Shop game: Left: the AGG-$\emptyset$ representation without function nodes (looking at only the neighborhood of α). Middle: we introduce two function nodes, p' (bottom) and p'' (top). Right: α now has only 3 neighbors.

Consider the Coffee Shop game from Example 2.7. For each action node α corresponding to a location, we introduce two simple aggregator function nodes, p'_α and p''_α. Let $\nu(p'_\alpha)$ be the set of actions surrounding α, and $\nu(p''_\alpha)$ be the set of actions corresponding to other locations. Then we set $\nu(\alpha) = \{\alpha, p'_\alpha, p''_\alpha\}$, as shown in Figure 3. Now each $c^{(\alpha)}$ is a configuration over only three nodes. Since each f^p is a simple aggregator, Corollary 2.11 applies and the size of this AGG-FN is polynomial in n and $\mathcal{A}$. In fact since the game is symmetric and the $\beta()$'s as defined in Proposition 2.13 are non-overlapping, we can calculate the exact value of $|C^{(\alpha)}|$ as the number of compositions of $n-1$ into four nonnegative integers, $\frac{(n+2)!}{(n-1)!3!} = n(n+1)(n+2)/6 = O(n^3)$. We must therefore store $Bn(n+1)(n+2)/6 = O(Bn^3)$ utility values. This is significantly more compact than the AGG-$\emptyset$ representation, which has a representation size of $O(B\frac{(n-1+B)!}{(n-1)!B!})$.

We can represent the parity game from Example 2.8 in a similar way. For each action α we create a function node p_α, and let $\nu(p_\alpha) = \nu(\alpha)$. We then modify $\nu(\alpha)$ so that it has only one member, p_α. For each function node p we define f^p as $f^p(c^{(p)}) = \sum_{\alpha \in \nu(p)} c(\alpha) \mod 2$. Since $\mathcal{R}(f^p) = \{0, 1\}$, Corollary 2.11 applies. In fact, each utility function just needs to store two values, and so the representation size is $O(|\mathcal{A}|)$ plus the size of the action graph.

2.3 AGG-FNs with Additive Structure

So far we have assumed that the utility functions $u^\alpha : C^{(\alpha)} \to \mathbb{R}$ are represented explicitly, i.e., by specifying the payoffs for all $c^{(\alpha)} \in C^{(\alpha)}$. This is not the only way to represent a mapping; the utility functions could be defined as analytical functions, decision trees, logic programs, circuits, or even arbitrary algorithms. These alternative representations might be more natural for humans to specify, and in many cases are more compact than the explicit representation. However, this extra compactness does not always allow us to reason more efficiently with the games. In this section, we look at utility functions with *additive structure*. These functions can be represented compactly and do allow more efficient computation.

2.3.1 Definition of AGG-FNs with Additive Structure

We say that a multivariate function has *additive structure* if it can be written as a (weighted) sum of functions of subsets of the variables. This form is more compact because we only need to represent the summands, which have lower dimensionality than the entire function.

We extend the AGG-FN representation by allowing u^α to be represented as a weighted sum of the configuration of the neighbors of α.[7]

[7] Such a utility function could also be represented using standard function nodes representing summation. However,

Definition 2.14 *A utility function u^α of an AGG-FN is* additive *if for all $m \in \nu(\alpha)$ there exist $\lambda_m \in \mathbb{R}$, such that*

$$u^\alpha(c^{(\alpha)}) \equiv \sum_{m\in\nu(\alpha)} \lambda_m c(m). \tag{2.1}$$

Such an additive utility function can be represented as the tuple $(\lambda_m)_{m\in\nu(\alpha)}$. This is a very versatile representation of additivity, because the neighbors of α can be function nodes. Thus additive utility functions can represent weighted sums of arbitrary functions of configurations over action nodes. We now formally define an AGG-FN representation where some of the utility functions are additive.

Definition 2.15 *An* AGG-FN with additive structure (AGG-FNA) *is a tuple $(N, A, \mathcal{P}, G, f, \mathcal{A}_+, \Lambda, u)$ where $N, A, \mathcal{P}, G, f$ are as defined in Definition 2.9, and*

- *$\mathcal{A}_+ \subseteq \mathcal{A}$ is the set of actions whose utility functions are additive;*
- *$\Lambda = (\lambda^{\alpha_+})_{\alpha_+\in\mathcal{A}_+}$, where each $\lambda^{\alpha_+} = (\lambda_m^{\alpha_+})_{m\in\nu(\alpha)}$ is the tuple of coefficients representing the additive utility function u^{α_+};*
- *$u = (u^\alpha)_{\alpha\in\mathcal{A}\setminus\mathcal{A}_+}$, where each u^α is as defined in Definition 2.9. These are the non-additive utility functions of the game, which are represented explicitly.*

2.3.2 Representation Size

We only need $|\nu(\alpha)|$ numbers to represent the coefficients of an additive utility function u^α, whereas the explicit representation requires $|C^{(\alpha)}|$ numbers. Of course we also need to take into account the sizes of the neighboring function nodes $p \in \nu(\alpha)$ and their corresponding functions f^p, which represent the summands of the additive functions. Each f^p either has a simple description requiring negligible space, or is represented explicitly as a mapping. In the latter case its size can be analyzed the same way as utility functions on action nodes. That is, when the neighbors of p are all actions then Proposition 2.6 applies; otherwise the discussion in Section 2.2.3 applies.

2.3.3 Representing Congestion Games as AGG-FNAs

A congestion game is a tuple $(N, M, (A_i)_{i\in N}, (K_{jk})_{j\in M, k\leq n})$, where $N = \{1, \ldots, n\}$ is the set of players, $M = \{1, \ldots, m\}$ is a set of facilities (or resources); A_i is player i's set of actions; each action $a_i \in A_i$ is a subset of the facilities: $a_i \subset M$. K_{jk} is the cost on facility j when k players have chosen actions that include facility j. For notational convenience we also define $K_j(k) \equiv K_{jk}$. Let $\#(j, a)$ be the number of players that chose facility j given the action profile a. The total cost, or disutility of player i under pure strategy profile $a = (a_i, a_{-i})$ is the sum of the cost on each of the facilities in a_i,

$$Cost_i(a_i, a_{-i}) = -u_i(a_i, a_{-i}) = \sum_{j\in a_i} K_j(\#(j, a)). \tag{2.2}$$

Congestion games exhibit a specific combination of anonymity and additive structure, which allows them to be represented compactly. Only nm numbers are needed to specify the costs $(K_{jk})_{j\in M, k\leq n}$. The representation also needs to specify the $\sum_{i\in N} |A_i|$ actions, each of which

we treat the common case of additivity separately because it is amenable to special-purpose computational methods (intuitively, leveraging the linearity of expectation; see Section 4.3).

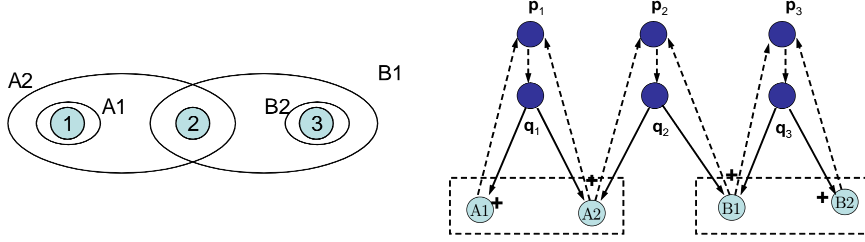

Figure 4: Left: a two-player congestion game with three facilities. The actions are shown as ovals containing their respective facilities. Right: the AGG-FNA representation of the same congestion game.

is a subset of M. If we use an m-bit binary string to represent each of these subsets, the total size of the congestion game representation is $\Theta(mn + m\sum_{i\in N}|A_i|)$.

An arbitrary congestion game can be encoded as an AGG-FNA with no loss of compactness, where all u^α are represented as additive utility functions. Given a congestion game $(N, M, (A_i)_{i\in N}, (K_{jk})_{j\in M, k\le n})$, we construct an AGG-FNA with the same number of players and same number of actions for each player as follows.

- Create $\sum_{i\in N}|A_i|$ action nodes, corresponding to the actions in the congestion game. In other words, the action sets do not overlap.
- Create $2m$ function nodes, labeled $(p_1, \ldots, p_m, q_1, \ldots, q_m)$. For each $j \in M$, there is an edge from p_j to q_j. For all $j \in M$ and for all $\alpha \in \mathcal{A}$, if facility j is included in action α in the congestion game, then in the action graph there is an edge from the action node α to p_j, and also an edge from q_j to α.
- For each p_j, define $c(p_j) \equiv \sum_{\alpha\in\nu(j)} c(\alpha)$, i.e., p_j is a simple aggregator. Since its neighbors are the actions that includes facility j, thus $c(p_j)$ is the number of players that chose facility j, which is $\#(j, a)$.
- Assign each q_j only one neighbor, namely p_j, and define $c(q_j) \equiv f^{q_j}(c(p_j)) \equiv K_j(c(p_j))$. In other words, $c(q_j)$ is exactly $K_j(\#(j, a))$, the cost on facility j.
- For each action node α, represent the utility function u^α as an additive function with weight -1 for each of its neighbors,

$$u^\alpha(c^{(\alpha)}) = \sum_{j\in\nu(\alpha)} -c(j) = -\sum_{j\in\nu(\alpha)} K_j(\#(j, a)). \tag{2.3}$$

Example 2.16 (Congestion game) *Consider the AGG-FNA representation of a two-player congestion game (see Figure 4). The congestion game has three facilities labeled {1, 2, 3}. Player A has actions A1={1} and A2={1, 2}; Player B has actions B1={2, 3} and B2={3}.*

Now let us consider the representation size of this AGG-FNA. The action graph has $|\mathcal{A}|+2m$ nodes and $O(m|\mathcal{A}|)$ edges; the function nodes $p_1, \ldots, p_m$ are simple aggregators and each only requires constant space; each f^{q_j} requires n numbers to specify so the total size of the AGG-FNA is $\Theta(mn + m|\mathcal{A}|) = \Theta(mn + m\sum_{i\in N}|A_i|)$. Thus this AGG-FNA representation has the same space complexity as the original congestion game representation.

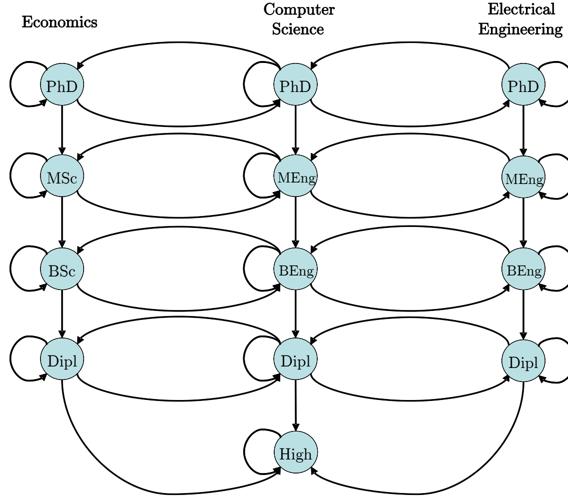

Figure 5: AGG-∅ representation of the Job Market game.

One extension of congestion games is *player-specific congestion games* [Milchtaich, 1996; Monderer, 2007]. Instead of all players having the same costs K_{jk}, in these games each player has a different set of costs. This can be easily represented as an AGG-FNA by following the construction above, but using a different set of function nodes $q_{i1}, \ldots, q_{im}$ for each player i.

3 Further Examples

In this section we provide several more examples of structured games that can be compactly represented as AGGs.

3.1 A Job Market

Here we describe a class of example games that can be compactly represented as AGG-∅s. Unlike the Ice Cream Vendor game, the following example does not involve choosing among actions that correspond to geographical locations.

Example 3.1 (Job Market game) *Consider the individuals competing in a job market. Each player chooses a field of study and a level of education to achieve. The utility of player i is the sum of two terms: (a) a constant cost depending only on the chosen field and education level, capturing the difficulty of studies and the cost of tuition and forgone wages; and (b) a variable reward, depending on (i) the number of players who chose the same field and education level as i, (ii) the number of players who chose a related field at the same education level, and (iii) the number of players who chose the same field at one level above or below i.*

Figure 5 gives an action graph modeling one such job market scenario, in which there are three fields, Economics, Computer Science and Electrical Engineering . For each field there are four levels of postsecondary study: Diploma, Bachelor, Master and PhD. Economics and Computer Science are considered related fields, and so are Computer Science and Electrical Engineering. There is another action representing high school education, which does not require a

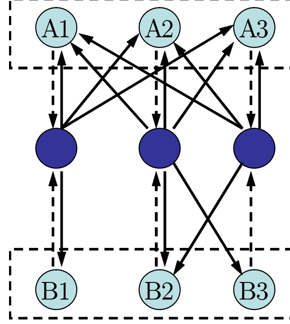

Figure 6: AGG-FN representation of a game with agent-specific utility functions.

choice of field. The maximum in-degree of the action graph is five, whereas a naive representation of the game as a symmetric game (see Section 2.1) would correspond to a complete action graph with in-degree 13. Thus this AGG-∅ representation is able to take advantage of anonymity as well as context-specific independence structure.

3.2 Representing Anonymous Games as AGG-FNs

One property of the AGG-∅ representation as defined in Section 2.1 is that utility function u^α is shared by all players who have α in their action sets. What if we want to represent games with *agent-specific* utility functions, where utilities depend not only on α and $c^{(\alpha)}$, but also on the *identity* of the player playing α?

Researchers have studied *anonymous games*, which deviate from symmetric games by allowing agent-specific utility functions [Kalai, 2004; Kalai, 2005; Daskalakis & Papadimitriou, 2007]. To represent games of this type as AGGs, we cannot just let multiple players share action α, because that would force those players to have the same utility function u^α. It does work to give agents non-overlapping action sets, replicating each action once for each agent. However, the resulting AGG-∅ is not compact; it does not take advantage of the fact that each of the replicated actions affects other players' utilities in the same way. Using function nodes, it is possible to compactly represent this kind of structure. We again split α into separate action nodes α_i for each player i able to take the action. Now we also introduce a function node p with every α_i as a neighbor, and define f^p to be a simple aggregator. Now p gives the total number of agents who chose action α, expressing anonymity, and action nodes include p as a neighbor instead of each α_i. This allows agents to have different utility functions without sacrificing representational compactness.

Example 3.2 (Anonymous game) *Consider an anonymous game with two classes of players, each class sharing the same utility functions. The AGG-FN representation of the game is shown in Figure 6. Players from the first class have action set $\{A1, A2, A3\}$, and players from the second class have action set $\{B1, B2, B3\}$. Furthermore, the utility functions of the second class of players exhibit certain context-specific independence structure, which are expressed by the absence of some of the possible edges from function nodes to action nodes B1, B2, B3.*

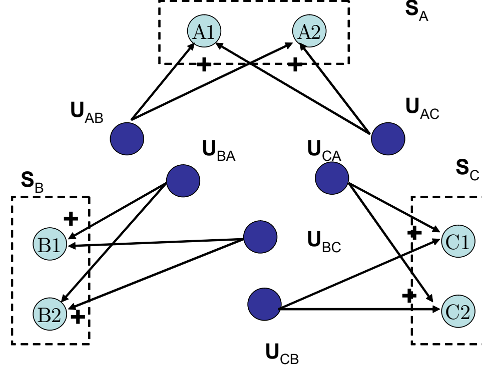

Figure 7: AGG-FNA representation of a 3-player polymatrix game. Function node U_{AB} represents player A's payoffs in his bimatrix game against B, U_{BA} represents player B's payoffs in his bimatrix game against A, and so on. To avoid clutter we do not show the edges from the action nodes to the function nodes in this graph. Such edges exist from A and B's actions to U_{AB} and U_{BA}, from A and C's actions to U_{AC} and U_{CA}, and from B and C's actions to U_{BC} and U_{CB}.

3.3 Representing Polymatrix Games as AGG-FNAs

In a *polymatrix game* [Yanovskaya, 1968], each player's utility is the sum of utilities resulting from her bilateral interactions with each of the $n-1$ other players. This can be represented by specifying for each pair of players i and j a bimatrix game (two-player normal form game) with set of actions A_i and A_j. A polymatrix game can be compactly represented as an AGG-FNA. The encoding is as follows. The AGG-FNA has non-overlapping action sets. For each pair of players (i, j), we create two function nodes to represent i and j's payoffs under the bimatrix game between them. Each of these function nodes has incoming edges from all of i's and j's actions. For each player i and each of his actions a_i, there are incoming edges from the $n-1$ function nodes representing i's payoffs in his bimatrix games against each of the other players. u^{a_i} is an additive utility function with weights equal to 1. Based on arguments similar to those in Section 2.1.2, this AGG-FNA representation has the same space complexity as the total size of the bimatrix games.

Example 3.3 (Polymatrix game) *Consider the AGG-FNA representation of a three-player polymatrix game, given in Figure 7. Each player's payoff is the sum of her payoffs in 2×2 game with played with each of the other players; she is only able to choose her action once. This additive utility function can be captured by introducing a function node U_{ij} to represent each player i's utility in the bimatrix game played with player j.*

3.4 Congestion Games with Action-Specific Rewards

So far the only use we have shown for AGG-FNAs is bringing existing game representations into the AGG framework. Of course, another key advantage of our approach is the ability to compactly represent games that would not have been compact under these existing game representations. We now give such an example.

Example 3.4 (Congestion game with action-specific rewards) *Consider the following game with n players. As in a congestion game, there is a set of facilities M, each action involves choosing a*

subset of the facilities, and the cost for facility j depends only on the number of players that chose facility j. Now further assume that, in addition to the cost of using the facilities, each player i also derives some utility R_i depending only on her own action, i.e., the set of facilities she chose. This utility is not necessarily additive across facilities. That is, in general if $A, B \in M$ and $A \cap B = \emptyset$, $R_i(A \cup B) \neq R_i(A) + R_i(B)$. So i's total utility is

$$u_i(a) = R_i(a_i) - \sum_{j \in a_i} K_j(\#(j, a)). \quad (3.1)$$

This game can model a situation in which the players use the facilities to complete a task, and the utility of the task depends on the facilities chosen. Another interpretation is given by Ben-Sasson et al. *[2006], in their analysis of "congestion games with strategy costs," which also have exactly this type of utility function. This work interpreted (the negative of) $R_i(a_i)$ as the computational cost of choosing the pure strategy a_i in a congestion game.*

This game cannot be compactly represented as a congestion game or a player-specific congestion game,[8] *but it can be compactly represented as an AGG-FNA. We create $\sum_i |A_i|$ action nodes, giving the agents nonoverlapping action sets. We have shown in Section 2.3.3 that we can use function nodes and additive utility functions to represent the congestion-game-like costs. Beyond this construction, we just need to create a function node r_i for each player i and define $c(r_i)$ to be equal to $R_i(a_i)$. The neighbors of r_i are i's entire action set: $\nu(r_i) = A_i$. Since the action sets do not overlap, there are only $|A_i|$ distinct configurations over A_i. In other words, $|C^{(r_i)}| = |A_i|$ and we need only $O(|A_i|)$ space to represent each R_i. The total size of the representation is $O(mn + m \sum_{i \in N} |A_i|)$.*

4 Computing Expected Payoff with AGGs

Up to this point, we have concentrated on how AGGs may be used to compactly represent games of interest. But compact representation is only half the story, and indeed by itself is relatively easy to achieve. Our goal is to identify a compact representation that can be used directly (e.g., without conversion to its induced normal form) for the computation of game-theoretic quantities of interest. We now turn to this computational perspective, and show that we can indeed leverage AGG's representational compactness in the computation of game-theoretic quantities. In this section we focus on the computational task of computing an agent's expected payoff under a mixed strategy profile. While this quantity can be important in itself, it is even more important as an inner-loop problem in the computation of many game-theoretic quantities. Some examples include computing best responses, checking if a given mixed strategy profile is a Nash equilibrium, 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], Turocy's algorithm for computing quantal response equilibria [Turocy, 2005], and Papadimitriou and Roughgarden's algorithm for finding correlated equilibria [Papadimitriou & Roughgarden, 2008]. We discuss some of these applications in Section 5.

Our main result of this section is an algorithm that efficiently computes expected payoffs of AGGs by exploiting their context-specific independence, anonymity and additivity structure. In Section 4.1 we introduce our expected payoff algorithm for AGG-∅s, and show (in Theorem

[8]Interestingly, Ben-Sasson *et al.* [2006] showed that this game belongs to the set of potential games, which implies that there exists an equivalent congestion game. However, building such a congestion game from the potential function following Monderer and Shapley's [1996] construction yields an exponential number of facilities, meaning that this congestion game representation is exponentially larger than the AGG-FNA representation presented here.

4.1) that the algorithm runs in time polynomial in the size of the input AGG-$\emptyset$. For the special case of symmetric strategies in symmetric AGG-$\emptyset$s, we present a different algorithm in Section 4.1.4 which runs asymptotically faster than our general algorithm for AGG-$\emptyset$s; in Section 4.1.5 we extend this approach to the broader class of k-*symmetric* AGG-$\emptyset$s. Finally, in Sections 4.2 and 4.3 we extend our expected payoff algorithm to AGG-FNs and AGG-FNAs respectively, and identify (in Theorems 4.5 and 4.6) conditions under which these extended algorithms run in polynomial time.

4.1 Computing Expected Payoff for AGG-$\emptyset$s

We must begin by introducing some notation. Let $\varphi(X)$ denote the set of all probability distributions over a set X. Define the set of mixed strategies for i as $\Sigma_i \equiv \varphi(A_i)$, and the set of all mixed strategy profiles as $\Sigma \equiv \prod_{i \in N} \Sigma_i$. Denote an element of Σ_i by σ_i, an element of Σ by σ, and the probability that i plays action α as $\sigma_i(\alpha)$. The *support* of a mixed strategy σ_i is the set of pure strategies played with positive probability (i.e., pure strategies a_i for which $\sigma_i(a_i) > 0$).

Now we can write the expected utility to agent i for playing pure strategy a_i, given that all other agents play the mixed strategy profile σ_{-i}, as

$$V^i_{a_i}(\sigma_{-i}) \equiv \sum_{a_{-i} \in A_{-i}} u_i(a_i, a_{-i}) \Pr(a_{-i}|\sigma_{-i}), \tag{4.1}$$

$$\Pr(a_{-i}|\sigma_{-i}) \equiv \prod_{j \neq i} \sigma_j(a_j). \tag{4.2}$$

Note that Equation 4.2 gives the probability of a_{-i} under the mixed strategy σ_{-i}. In the rest of this section we focus on the problem of computing $V^i_{a_i}(\sigma_{-i})$ given i, a_i and σ_{-i}. Having established the machinery to compute $V^i_{a_i}(\sigma_{-i})$, we can then compute the expected utility of player i under a mixed strategy profile σ as $\sum_{a_i \in A_i} \sigma_i(a_i) V^i_{a_i}(\sigma_{-i})$.

One might wonder why Equations (4.1) and (4.2) are not the end of the story. Notice that Equation (4.1) is a sum over the set A_{-i} of action profiles of players other than i. The number of terms is $\prod_{j \neq i} |A_j|$, which grows exponentially in n. If we were to use the normal form representation, there really would be $|A_{-i}|$ different outcomes to consider, each with potentially distinct payoff values. Thus, using normal form the evaluation of Equation (4.1) would be the best possible algorithm for computing $V^i_{a_i}$. Since AGGs are fully expressive, the same is true for games without any structure represented as AGGs. However, what about games that are exponentially more compact when represented as AGGs than when represented in the normal form? For these games, evaluating Equation (4.1) amounts to an exponential-time algorithm.

In this section we present an algorithm that given any i, a_i and σ_{-i}, computes the expected payoff $V^i_{a_i}(\sigma_{-i})$ in time polynomial in the size of the AGG-$\emptyset$ representation. In other words, our algorithm is efficient if the AGG-$\emptyset$ is compact, and requires time exponential in n if it is not. In particular, recall from Proposition 2.6 any AGG-$\emptyset$ with maximum in-degree bounded by a constant has a representation size that is polynomial in n. As a result our algorithm is polynomial in n for such games.

4.1.1 Exploiting Context-Specific Independence: Projection

First, we consider how to take advantage of the context-specific independence structure of an AGG-$\emptyset$: the fact that i's payoff when playing a_i only depends on configurations over the neighborhood of i. The key idea is that we can *project* other players' strategies onto a smaller action space that is strategically the same from the point of view of an agent who chose action a_i. That

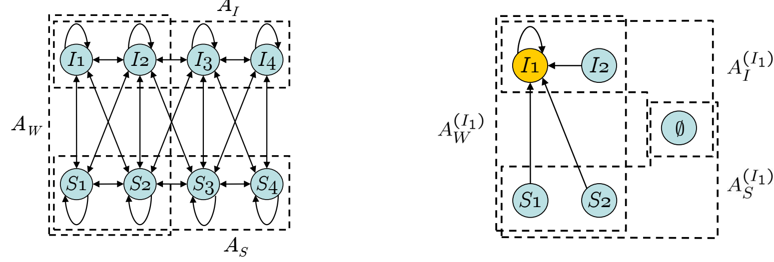

Figure 8: Projection of the action graph. Left: action graph of the Ice Cream Vendor game. Right: projected action graph and action sets with respect to the action C1.

is, we construct a graph from the point of view of a given agent, expressing his sense that actions that do not affect his chosen action are in a sense the "same action." This can be seen as inducing a context-specific graphical game. Formally, for every action $\alpha \in \mathcal{A}$ define a reduced graph $G^{(\alpha)}$ by including only the nodes $\nu(\alpha)$ and a new node denoted $\emptyset$. The only edges included in $G^{(\alpha)}$ are the directed edges from each of the nodes $\nu(\alpha)$ to the node α. Player j's action a_j is projected to a node $a_j^{(\alpha)}$ in the reduced graph $G^{(\alpha)}$ by the mapping

$$a_j^{(\alpha)} \equiv \begin{cases} a_j & a_j \in \nu(\alpha) \\ \emptyset & a_j \notin \nu(\alpha) \end{cases} . \tag{4.3}$$

In other words, actions that are not in $\nu(\alpha)$ (and therefore do not affect the payoffs of agents playing α) are projected onto a new action, $\emptyset$. The resulting *projected* action set $A_j^{(\alpha)}$ has cardinality at most $\min(|A_j|, |\nu(\alpha)| + 1)$. This is illustrated in Figure 8, using the Ice Cream Vendor game described in Example 2.5.

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

$$\sigma_j^{(\alpha)}(a_j^{(\alpha)}) \equiv \begin{cases} \sigma_j(a_j) & a_j \in \nu(\alpha) \\ \sum_{\alpha' \in A_j \setminus \nu(\alpha)} \sigma_j(\alpha') & a_j^{(\alpha)} = \emptyset \end{cases} . \tag{4.4}$$

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

$$V_{a_i}^i(\sigma_{-i}) = \sum_{a_{-i}^{(a_i)} \in A_{-i}^{(a_i)}} u\left(a_i, \mathcal{C}^{(a_i)}(a_i, a_{-i})\right) \Pr\left(a_{-i}^{(a_i)} | \sigma_{-i}^{(a_i)}\right),$$

$$\Pr\left(a_{-i}^{(a_i)} | \sigma_{-i}^{(a_i)}\right) = \prod_{j \neq i} \sigma_j^{(a_i)}\left(a_j^{(a_i)}\right).$$

The summation is over $A_{-i}^{(a_i)}$, which in the worst case has $(|\nu(a_i)| + 1)^{(n-1)}$ terms. So for AGG-$\emptyset$s with strict or context-specific independence structure, computing $V_{a_i}^i(\sigma_{-i})$ in this way is exponentially faster than doing the summation in (4.1) directly. However, the time complexity of this approach is still exponential in n.

4.1.2 Exploiting Anonymity: Summing over Configurations

Next, we want to take advantage of the anonymity structure of the AGG-$\emptyset$. 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 $V^i_{a_i}(\sigma_{-i})$ as a sum over the possible configurations, weighted by their probabilities:

$$V^i_{a_i}(\sigma_{-i}) = \sum_{c^{(a_i)} \in C^{(a_i,i)}} u_i\left(a_i, c^{(a_i)}\right) \Pr\left(c^{(a_i)} | \sigma^{(a_i)}\right), \tag{4.5}$$

$$\Pr\left(c^{(a_i)} | \sigma^{(a_i)}\right) = \sum_{\substack{a: \\ \mathcal{C}^{(a_i)}(a) = c^{(a_i)}}} \prod_{j=1}^{n} \sigma_j(a_j). \tag{4.6}$$

where $\sigma^{(a_i)} \equiv (a_i, \sigma^{(a_i)}_{-i})$ and $\Pr(c^{(a_i)}|\sigma^{(a_i)})$ is the probability of $c^{(a_i)}$ given the mixed strategy profile $\sigma^{(a_i)}$. Recall that $C^{(a_i,i)}$ is the set of configurations over $\nu(a_i)$ given that i played a_i. So Equation (4.5) is a summation of size $|C^{(a_i,i)}|$, the number of configurations given that i played a_i, which is polynomial in n if $|\nu(a_i)|$ is bounded by a constant. The difficult task is to compute $\Pr(c^{(a_i)}|\sigma^{(a_i)})$ for all $c^{(a_i)} \in C^{(a_i,i)}$, i.e., the probability distribution over $C^{(a_i,i)}$ induced by $\sigma^{(a_i)}$. We observe that the sum in Equation (4.6) is over the set of all action profiles corresponding to the configuration $c^{(a_i)}$. The size of this set is exponential in the number of players. Therefore directly computing the probability distribution using Equation (4.6) would take time exponential in n.

Can we do better? We observe that the players' mixed strategies are independent, i.e., σ is a product probability distribution $\sigma(a) = \prod_i \sigma_i(a_i)$. Also, each player affects the configuration c independently. This structure allows us to use dynamic programming (DP) to efficiently compute the probability distribution $\Pr(c^{(a_i)}|\sigma^{(a_i)})$. The intuition behind our algorithm is to apply one agent's mixed strategy at a time, effectively adding one agent at a time to the action graph. Let $\sigma^{(a_i)}_{1\ldots k}$ denote the projected strategy profile of agents $\{1, \ldots, k\}$. Denote by $C^{(a_i)}_k$ the set of configurations induced by actions of agents $\{1, \ldots, k\}$. Similarly, write $c^{(a_i)}_k \in C^{(a_i)}_k$. Denote by P_k the probability distribution on $C^{(a_i)}_k$ induced by $\sigma^{(a_i)}_{1\ldots k}$, and by $P_k[c]$ the probability of configuration c. At iteration k of the algorithm, we compute P_k from P_{k-1} and $\sigma^{(a_i)}_k$. After iteration n, the algorithm stops and returns P_n. The pseudocode of our DP algorithm is shown as Algorithm 1, and our full algorithm for computing $V^i_{a_i}(\sigma_{-i})$ is summarized in Algorithm 2.

Each $c^{(a_i)}_k$ 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 $|\nu(a_i)|$. To achieve efficient enumeration of all elements of a trie, we store the elements in a list, in the order of their insertion. We omit the proof of correctness of our algorithm, which is relatively straightforward. It is given in Section 2.3.3 of [Jiang, 2006].

4.1.3 Complexity

Let $C^{(a_i,i)}(\sigma_{-i})$ denote the set of configurations over $\nu(a_i)$ that have positive probability of occurring under the mixed strategy (a_i, σ_{-i}). In other words, this is the number of terms we

Algorithm 1: Computing the induced probability distribution $\Pr(c^{(a_i)}|\sigma^{(a_i)})$.

```
Input: a_i, σ^(a_i)
Output: P_n, which is the distribution Pr(c^(a_i) | σ^(a_i)) represented as a trie.
c_0^(a_i) = (0, ..., 0);
P_0[c_0^(a_i)] = 1.0 ; // Initialization:  C_0^(a_i) = {c_0^(a_i)}
for k = 1 to n do
    Initialize P_k to be an empty trie;
    foreach c_{k-1}^(a_i) from P_{k-1} do
        foreach a_k^(a_i) ∈ A_k^(a_i) such that σ_k^(a_i)(a_k^(a_i)) > 0 do
            c_k^(a_i) = c_{k-1}^(a_i);
            if a_k^(a_i) ≠ ∅ then
                c_k^(a_i)(a_k^(a_i)) += 1 ; // Apply action a_k^(a_i)
            if P_k[c_k^(a_i)] does not exist yet then
                P_k[c_k^(a_i)] = 0.0;
            P_k[c_k^(a_i)] += P_{k-1}[c_{k-1}^(a_i)] × σ_k^(a_i)(a_k^(a_i));
return P_n
```

need to add together when doing the weighted sum in Equation (4.5). When σ_{-i} has full support, $C^{(a_i,i)}(\sigma_{-i}) = C^{(a_i,i)}$.

Theorem 4.1 *Given an AGG-$\emptyset$ representation of a game, i's expected payoff $V^i_{a_i}(\sigma_{-i})$ can be computed in $\Theta(n|\mathcal{A}| + n|\nu(a_i)|^2|C^{(a_i,i)}(\sigma_{-i})|)$ time, which is polynomial in the size of the representation. If $\mathcal{I}$, the in-degree of the action graph, is bounded by a constant, $V^i_{a_i}(\sigma_{-i})$ can be computed in time polynomial in n.*

Proof. Since looking up an entry in a trie takes time linear in the size of the key, which is $|\nu(a_i)|$ in our case, the complexity of doing the weighted sum in Equation (4.5) is $O(|\nu(a_i)||C^{(a_i,i)}(\sigma_{-i})|)$.

Algorithm 1 requires n iterations; in iteration k, we look at all possible combinations of $c^{(a_i)}_{k-1}$ and $\alpha^{(a_i)}_k$, and in each case do a trie look-up which costs $\Theta(|\nu(a_i)|)$. Since $|\mathcal{A}^{(a_i)}_k| \leq |\nu(a_i)| + 1$, and $|C^{(a_i)}_{k-1}| \leq |C^{(a_i,i)}|$, the complexity of Algorithm 1 is $\Theta(n|\nu(a_i)|^2|C^{(a_i,i)}(\sigma_{-i})|)$. This dominates the complexity of summing up Equation (4.5). Adding the cost of computing $\sigma^{(\alpha)}_{-i}$, we get the overall complexity of expected payoff computation $\Theta(n|\mathcal{A}| + n|\nu(a_i)|^2|C^{(a_i,i)}(\sigma_{-i})|)$.

Since $|C^{(a_i,i)}(\sigma_{-i})| \leq |C^{(a_i,i)}| \leq |C^{(a_i)}|$, and $|C^{(a_i)}|$ is the number of payoff values stored in payoff function u^{a_i}, this means that expected payoffs can be computed in polynomial time with respect to the size of the AGG-$\emptyset$. Furthermore, our algorithm is able to exploit strategies with small supports which lead to a small $|C^{(a_i,i)}(\sigma_{-i})|$. Since $|C^{(a_i)}|$ is bounded by $\frac{(n-1+|\nu(a_i)|)!}{(n-1)!|\nu(a_i)|!}$, this implies that if the in-degree of the graph is bounded by a constant, then the complexity of computing expected payoffs is $O(n|\mathcal{A}| + n^{\mathcal{I}+1})$. ■

The proof of Theorem 4.1 shows that besides exploiting the compactness of the AGG-$\emptyset$ representation, our algorithm is also able to exploit the cases where the mixed strategy profiles

Algorithm 2 Computing expected utility $V^i_{a_i}(\sigma_{-i})$, given a_i and σ_{-i}.

1. for each $j \neq i$, compute the projected mixed strategy $\sigma_j^{(a_i)}$ using Equation (4.4):

$$\sigma_j^{(a_i)}(a_j^{(a_i)}) \equiv \begin{cases} \sigma_j(a_j) & a_j \in \nu(a_i) \\ \sum_{\alpha' \in A_j \setminus \nu(a_i)} \sigma_j(\alpha') & a_j^{(a_i)} = \emptyset \end{cases}.$$

2. compute the probability distribution $\Pr(c^{(a_i)}|a_i, \sigma_{-i}^{(a_i)})$ by following Algorithm 1.
3. calculate the expected utility using the following weighted sum (Equation (4.5)):

$$V^i_{a_i}(\sigma_{-i}) = \sum_{c^{(a_i)} \in C^{(a_i,i)}} u_i\left(a_i, c^{(a_i)}\right) \Pr\left(c^{(a_i)}|\sigma^{(a_i)}\right).$$

given have small support sizes, because the time complexity depends on $|C^{(a_i,i)}(\sigma_{-i})|$ which is small when support sizes are small. This is important in practice, since we will often need to carry out expected utility computations for strategy profiles with small supports. Porter *et al.* [2008] observed that quite often games have Nash equilibria with small support, and proposed algorithms that explicitly search for such equilibria. In other algorithms for computing Nash equilibria such as Govindan-Wilson and simplicial subdivision, it is also quite often necessary to compute expected payoffs for mixed strategy profiles with small support.

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 $|C^{(a_i,i)}(\sigma_{-i})|$ remains the same, the ordering does affect the intermediate configurations $C_k^{(a_i)}$. We can use the following heuristic to try to minimize the number of intermediate configurations: sort the players in ascending order of the sizes of their projected action sets. This reduces the amount of work we do in earlier iterations of Algorithm 1, but does not change its overall complexity.

4.1.4 The Case of Symmetric Strategies in Symmetric AGG-$\emptyset$s

As described in Section 2.1, if a game is symmetric it can be represented as an AGG-$\emptyset$ with $A_i = \mathcal{A}$ for all $i \in N$. Given a symmetric game, we are often interested in computing expected utilities under *symmetric* mixed strategy profiles, where a mixed strategy profile σ is symmetric if $\sigma_i = \sigma_j \equiv \sigma_*$ for all $i, j \in N$. In Section 5.2.2 we will discuss algorithms that make use of expected utility computation under symmetric strategy profiles to compute a symmetric Nash equilibrium of symmetric games.

To compute the expected utility $V^i_{a_i}(\sigma_*)$, we could use the algorithm we proposed for general AGG-$\emptyset$s under arbitrary mixed strategies, which requires time polynomial in the size of the AGG-$\emptyset$. But we can gain additional computational speedup by exploiting the symmetry in the game and the strategy profile.

As before, we want to use Equation (4.5) to compute the expected utility, so the crucial task is again computing the probability distribution over projected configurations, $\Pr(c^{(a_i)}|\sigma^{(a_i)})$. Recall that $\sigma^{(a_i)} \equiv (a_i, \sigma_{-i}^{(a_i)})$. Define $\Pr(c^{(a_i)}|\sigma_*^{(a_i)})$ to be the distribution induced by $\sigma_{-i}^{(a_i)}$, the partial mixed strategy profile of players other than i, each playing the symmetric strategy $\sigma_*^{(a_i)}$. Once we have the distribution $\Pr(c^{(a_i)}|\sigma_*^{(a_i)})$, we can then compute the distribution $\Pr(c^{(a_i)}|\sigma^{(a_i)})$ straightforwardly by applying player i's strategy a_i. In the rest of this section we focus on computing $\Pr(c^{(a_i)}|\sigma_*^{(a_i)})$.

Define $\mathcal{S}(c^{(a_i)})$ to be the set containing all action profiles $a^{(a_i)}$ such that $\mathcal{C}(a^{(a_i)}) = c^{(a_i)}$.

Since all agents have the same mixed strategies, each pure action profile in $\mathcal{S}(c^{(a_i)})$ is equally likely, so for any $a^{(a_i)} \in \mathcal{S}(c^{(a_i)})$

$$\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right) = \left|\mathcal{S}(c^{(a_i)})\right| \Pr\left(a^{(a_i)}|\sigma_*^{(a_i)}\right), \tag{4.7}$$

$$\Pr\left(a^{(a_i)}|\sigma_*^{(a_i)}\right) = \prod_{\alpha\in\mathcal{A}^{(a_i)}} (\sigma_*^{(a_i)}(\alpha))^{c^{(a_i)}(\alpha)}. \tag{4.8}$$

The sizes of $\mathcal{S}(c^{(a_i)})$ are given by the multinomial coefficient

$$\left|\mathcal{S}\left(c^{(a_i)}\right)\right| = \frac{(n-1)!}{\prod_{\alpha\in\mathcal{A}^{(a_i)}}\left(c^{(a_i)}(\alpha)\right)!}. \tag{4.9}$$

Better still, using a Gray code technique we can avoid reevaluating these equations for every $c^{(a_i)} \in C^{(a_i)}$. Denote the configuration obtained from $c^{(a_i)}$ by decrementing by one the number of agents taking action $\alpha \in \mathcal{A}^{(a_i)}$ and incrementing by one the number of agents taking action $\alpha' \in \mathcal{A}^{(a_i)}$ as $c^{(a_i)\prime} \equiv c^{(a_i)}_{(\alpha\to\alpha')}$. Then consider the graph $H_{C^{(a_i)}}$ whose nodes are the elements of the set $C^{(a_i)}$, and whose directed edges indicate the effect of the operation $(\alpha \to \alpha')$. This graph is a regular triangular lattice inscribed within a $(|\mathcal{A}^{(a_i)}|-1)$-dimensional simplex. Having computed $Pr(c^{(a_i)}|\sigma_*^{(a_i)})$ for one node of $H_{C^{(a_i)}}$ corresponding to configuration $c^{(a_i)}$, we can compute the result for an adjacent node in $O(1)$ time,

$$\Pr\left(c^{(a_i)}_{(\alpha\to\alpha')}|\sigma_*^{(a_i)}\right) = \frac{\sigma_*^{(a_i)}(\alpha')c^{(a_i)}(\alpha)}{\sigma_*^{(a_i)}(\alpha)\left(c^{(a_i)}(\alpha')+1\right)} \Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right). \tag{4.10}$$

$H_{C^{(a_i)}}$ always has a Hamiltonian path (attributed to an unpublished result of Knuth by Klingsberg [1982]), so having computed $\Pr(c^{(a_i)}|\sigma_*^{(a_i)})$ for an initial $c^{(a_i)}$ using Equation (4.8), the results for all other projected configurations (nodes in $H_{C^{(a_i)}}$) can be computed by using Equation (4.10) at each subsequent step on the path. Generating the Hamiltonian path corresponds to finding a combinatorial Gray code for compositions; an algorithm with constant amortized running time is given by Klingsberg [1982]. Intuitively, it is easy to see that a simple, "lawnmower" Hamiltonian path exists for any lower-dimensional projection of $H_{C^{(a_i)}}$, with the only state required to compute the next node in the path being a direction value for each dimension.

Our algorithm for computing the distribution $\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right)$ is summarized in Algorithm 3. For computing expected utility, we again use Algorithm 2, except with Algorithm 3 replacing Algorithm 1 as the subroutine for computing the distribution $\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right)$.

Theorem 4.2 *Computation of the expected utility $V^i_{a_i}(\sigma_*)$ under a symmetric strategy profile for symmetric action-graph games using Equations* (4.5)*,* (4.7)*,* (4.8) *and* (4.10) *takes time* $O(|\mathcal{A}| + |\nu(a_i)|\left|C^{(a_i)}(\sigma^{(a_i)})\right|)$.

Proof. Projection to $\sigma_*^{(a_i)}$ takes $O(|\mathcal{A}|)$ time since the strategies are symmetric. Equation (4.5) has $\left|C^{(a_i)}(\sigma^{(a_i)})\right|$ summands. The probability for the initial configuration requires $O(n)$ time. Using Gray codes the computation of subsequent probabilities can be done in constant amortized time for each configuration. Since each look-up of the utility function takes $O(|\nu(a_i)|)$ time, the total complexity of the algorithm is $O(|\mathcal{A}| + |\nu(a_i)|\left|C^{(a_i)}(\sigma^{(a_i)})\right|)$. ■

Algorithm 3 Computing distribution $\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right)$ in a symmetric AGG-$\emptyset$

1. let $c^{(a_i)} = c_0^{(a_i)}$, where $c_0^{(a_i)}$ is the initial node of a Hamiltonian path of $H_{C^{(a_i)}}$.
2. compute $\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right)$ using Equation (4.7):

$$\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right) = \frac{(n-1)!}{\prod_{\alpha \in \mathcal{A}^{(a_i)}} (c^{(a_i)}(\alpha))!} \prod_{\alpha \in \mathcal{A}^{(a_i)}} (\sigma_*^{(a_i)}(\alpha))^{c^{(a_i)}(\alpha)}.$$

3. While there are more configurations in $C^{(a_i)}$:
   (a) get the next configuration $c^{(a_i)}_{(\alpha \to \alpha')}$ in the Hamiltonian path, using Klingsberg's algorithm [Klingsberg, 1982].
   (b) compute $\Pr\left(c^{(a_i)}_{(\alpha \to \alpha')}|\sigma_*^{(a_i)}\right)$ using Equation (4.10):

$$\Pr\left(c^{(a_i)}_{(\alpha \to \alpha')}|\sigma_*^{(a_i)}\right) = \frac{\sigma_*^{(a_i)}(\alpha')c^{(a_i)}(\alpha)}{\sigma_*^{(a_i)}(\alpha)\left(c^{(a_i)}(\alpha')+1\right)} \Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right).$$

   (c) let $c^{(a_i)} = c^{(a_i)}_{(\alpha \to \alpha')}$.
4. output $\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right)$ for all $c^{(a_i)} \in C^{(a_i)}$.

Algorithm 4 Computing the probability distribution $\Pr(c^{(a_i)}|\sigma^{(a_i)})$ in a k-symmetric AGG-$\emptyset$ under a k-symmetric mixed strategy profile $\sigma^{(a_i)}$.

1. Partition the players according to $\{N_1, \ldots, N_k\}$.
2. For each $l \in \{1, \ldots, k\}$, compute $\Pr(c^{(a_i)}|\sigma_{N_l}^{(a_i)})$, the probability distribution induced by $\sigma_{N_l}^{(a_i)}$, the partial strategy profile of players in N_l. Since $\sigma_{N_l}^{(a_i)}$ is symmetric, this can be computed efficiently using Algorithm 3 as discussed in Section 4.1.4.
3. Combine the k probability distributions together using Algorithm 1, resulting in the distribution $\Pr(c^{(a_i)}|\sigma^{(a_i)})$.

Note that this is faster than our dynamic programming algorithm for general AGG-$\emptyset$s under arbitrary strategies, whose complexity is $\Theta(n|\mathcal{A}| + n|\nu(a_i)|^2 \left|C^{(a_i)}(\sigma^{(a_i)})\right|)$ by Theorem 4.1. In the usual case where the second term dominates the first, the algorithm for symmetric strategies is faster by a factor of $n|\nu(a_i)|$.

4.1.5 k-symmetric Games

We now move to a generalization of symmetry in games that we call k-symmetry.

Definition 4.3 *An AGG-$\emptyset$ is k-symmetric if there exists a partition $\{N_1, \ldots, N_k\}$ of N such that for all $l \in \{1, \ldots, k\}$, for all $i, j \in N_l$, $A_i = A_j$.*

Intuitively, k-symmetric AGG-$\emptyset$s represent games with k classes of identical agents, where agents within each class are identical. Note that all games are trivially n-symmetric. The Ice Cream Vendor game of Example 2.5 is a nontrivial k-symmetric AGG-$\emptyset$ with $k = 3$.

Given a k-symmetric AGG-$\emptyset$ with partition $\{N_1, \ldots, N_k\}$, a mixed strategy profile σ is k-symmetric if for all $l \in \{1, \ldots, k\}$, for all $i, j \in N_l$, $\sigma_i = \sigma_j$. We are often interested in computing expected utility under k-symmetric strategy profiles. For example in Section 5.2.2 we will discuss algorithms that make use of such expected utility computations to find k-symmetric Nash equilibria in k-symmetric games. To compute expected utility under a k-symmetric mixed strategy profile, we can use a hybrid approach when computing the probability distribution over configurations, shown in Algorithm 4. Observe that this algorithm combines our specialized Algorithm 3 for handling symmetric games from Section 4.1.4 with the idea of running Algorithm 1 on the joint mixed strategies of subgroups of agents discussed at the end of Section 4.1.3.

4.2 Computing Expected Payoff with AGG-FNs

Algorithm 1 cannot be directly applied to AGG-FNs with arbitrary f^p. First of all, projection of strategies does not work directly, because a player j playing an action $a_j \notin \nu(\alpha)$ could still affect $c^{(\alpha)}$ via function nodes. Furthermore, the general idea of using dynamic programming to build up the probability distribution by adding one player at a time does not work because for an arbitrary function node $p \in \nu(\alpha)$, each player would not be guaranteed to affect $c(p)$ independently. We could convert the AGG-FN to an AGG-$\emptyset$ in order to apply our algorithm, but then we would not be able to translate the extra compactness of AGG-FNs over AGG-$\emptyset$s into more efficient computation. In this section we identify two subclasses of AGG-FN for which expected utility can be efficiently computed. In Section 4.2.1 we show that when all function nodes belong to a restricted class of contribution-independent function nodes, expected utility can be computed in polynomial time. In Section 4.2.2 we reinterpret the expected utility problem as a Bayesian network inference problem, which can be computed in polynomial time if the resulting Bayesian network has bounded treewidth.

4.2.1 Contribution-Independent Function Nodes

Definition 4.4 *A function node p in an AGG-FN is* contribution-independent (CI) *if*

- *$\nu(p) \subseteq \mathcal{A}$, i.e., the neighbors of p are action nodes.*
- *There exists a commutative and associative operator $*$, and for each $\alpha \in \nu(p)$ an integer w_α, such that given an action profile $a = (a_1, \ldots, a_n)$, $c(p) = *_{i \in N : a_i \in \nu(p)} w_{a_i}$.*
- *The running time of each $*$ operation is bounded by a polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$. Furthermore, $*$ can be represented in space polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$.*

An AGG-FN is contribution-independent if all its function nodes are contribution-independent.

Note that it follows from this definition that $c(p)$ can be written as a function of $c^{(p)}$ by collecting terms: $c(p) \equiv f^p(c^{(p)}) = *_{\alpha \in \nu(p)} (*_{k=1}^{c(\alpha)} w_\alpha)$.

Simple aggregators can be represented as contribution-independent function nodes, with the $+$ operator serving as $*$, and $w_\alpha = 1$ for all α. The Coffee Shop game is thus an example of a contribution-independent AGG-FN. For the parity game in Example 2.8, $*$ is instead addition mod 2. An example of a non-additive CI function node arises in a perfect-information model of an (advertising) auction in which actions correspond to bid amounts [Thompson & Leyton-Brown, 2009]. Here we want $c(p)$ to represent the amount of the winning bid, and so we let w_α be the bid amount corresponding to action α, and $*$ be the $\max$ operator.

The advantage of contribution-independent AGG-FNs is that for all function nodes p, each player's strategy affects $c(p)$ independently. This fact allows us to adapt our algorithm to efficiently compute the expected utility $V^i_{a_i}(\sigma_{-i})$. For simplicity we present the algorithm for the

case where we have one operator $*$ for all $p \in \mathcal{P}$, but our approach can be directly applied to games with different operators and w_α associated with different function nodes.

We define the *contribution* of action α to node $m \in \mathcal{A} \cup \mathcal{P}$, denoted $\delta_\alpha(m)$, as 1 if $m = \alpha$, 0 if $m \in \mathcal{A} \setminus \{\alpha\}$, and $*_{m' \in \nu(m)}(*_{k=1}^{\delta_\alpha(m')} w_\alpha)$ if $m \in \mathcal{P}$. Then it is easy to verify that given an action profile $a = (a_1, \ldots, a_n)$, $c(\alpha) = \sum_{j=1}^n \delta_{a_j}(\alpha)$ for all $\alpha \in \mathcal{A}$ and $c(p) = *_{j=1}^n \delta_{a_j}(p)$ for all $p \in \mathcal{P}$. Given that player i played a_i, and for all $\alpha \in \mathcal{A}$, we define the *projected contribution* of action α under a_i, denoted $\delta_\alpha^{(a_i)}$, as the tuple $(\delta_\alpha(m))_{m \in \nu(a_i)}$. Note that different actions α may have identical projected contributions under a_i. Player j's mixed strategy σ_j induces a probability distribution over j's projected contributions, $\Pr(\delta^{(a_i)}|\sigma_j) = \sum_{a_j : \delta_{a_j}^{(a_i)} = \delta^{(a_i)}} \sigma_j(a_j)$. 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 $\sigma_j^{(a_i)}$ in our algorithm for AGG-$\emptyset$s.

Algorithm 1 for computing the distribution $\Pr(c^{(a_i)}|\sigma)$ can be straightforwardly adopted to work with contribution-independent AGG-FNs. Whenever we apply player k's contribution $\delta_{a_k}^{(a_i)}$ to $c_{k-1}^{(a_i)}$, the resulting configuration $c_k^{(a_i)}$ is computed componentwise as follows: $c_k^{(a_i)}(m) = \delta_{a_k}^{(a_i)}(m) + c_{k-1}^{(a_i)}(m)$ if $m \in \mathcal{A}$, and $c_k^{(a_i)}(m) = \delta_{a_k}^{(a_i)}(m) * c_{k-1}^{(a_i)}(m)$ if $m \in \mathcal{P}$.

To analyze the complexity of computing expected utility, it is necessary to know the representation size of a contribution-independent AGG-FN. For each function node p we need to specify $*$ and $(w_\alpha)_{\alpha \in \nu(p)}$ instead of f^p directly. Let $\| * \|$ denote the representation size of $*$. Then the total size of a contribution-independent AGG-FN is $O(\sum_{\alpha \in \mathcal{A}} |C^{(\alpha)}| + \| * \|)$. As discussed in Section 2.2.3, this size is not necessarily polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$; although when the conditions in Corollary 2.11 are satisfied, the representation size is polynomial.

Theorem 4.5 *Expected utility can be computed in time polynomial in the size of a contribution-independent AGG-FN. Furthermore, if the in-degrees of the action nodes are bounded by a constant and the sizes of ranges $|\mathcal{R}(f^p)|$ for all $p \in \mathcal{P}$ are bounded by a polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$, then expected utility can be computed in time polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$.*

> **Proof Sketch.** Following similar complexity analysis as Theorem 4.1, if an AGG-FN is contribution-independent, expected utility $V_{a_i}^i(\sigma_{-i})$ can be computed in $O(n|\mathcal{A}||C^{(a_i)}|(T_* + |\nu(a_i)|))$ time, where T_* denotes the maximum running time of an $*$ operation. Since T_* is polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$ by Definition 4.4, the running time for computing expected utility is polynomial in the size of the AGG-FN representation. The second part of the theorem follows from a direct application of Corollary 2.11. ■

For AGG-FNs whose function nodes are all simple aggregators, each player's set of projected contributions has size at most $|\nu(a_i) + 1|$, as opposed to $|\mathcal{A}|$ in the general case. This leads to a run time complexity of $O(n|\mathcal{A}| + n|\nu(a_i)|^2|C^{(a_i)}|)$, which is better than the complexity of the general case proved in Theorem 4.5. Applied to the Coffee Shop game, since $|C^{(\alpha)}| = O(n^3)$ and all function nodes are simple aggregators, our algorithm takes $O(n|\mathcal{A}| + n^4)$ time, which grows *linearly* in $|\mathcal{A}|$.

4.2.2 Beyond Contribution Independence

What about the case where not all function nodes are contribution-independent—is there anything we can do besides converting the AGG-FN into its induced AGG-$\emptyset$? It turns out that by reducing the problem of computing expected utility to a Bayesian network inference problem, we can still efficiently compute expected utilities for certain additional classes of AGG-FNs.

Bayesian networks compactly represent probability distributions exhibiting conditional independence structure (see, e.g., Russell and Norvig [2003]). A Bayesian network is a DAG in which nodes represent random variables and edges represent direct probabilistic dependence. Each node X is associated with a conditional probability distribution (CPD) specifying the probability of each realization of random variable X conditional on the realizations of its parent random variables.

A key step in our approach for computing expected utility in AGG-FNs is computing the probability distribution over configurations $\Pr(c^{(a_i)}|\sigma^{(a_i)})$. If we treat each node m's configuration $c(m)$ as a random variable, then the distribution over configurations can be interpreted as the joint probability distribution over the set of random variables $\{c(m)\}_{m\in\nu(a_i)}$. Given an AGG-FN, a player i and an action $a_i \in A_i$, we can construct an *induced Bayesian network* $\mathcal{B}^i_{a_i}$:

- The nodes of $\mathcal{B}^i_{a_i}$ consist of (i) one node for each element of $\nu(a_i)$; (ii) one node for each neighbor of a function node belonging to $\nu(a_i)$; and (iii) one node for each neighbor of a function node added in the previous step, and so on until no more function nodes are added. Each of these nodes m represents the random variable $c(m)$. We further introduce another kind of node: (iv) n nodes $\sigma_1, \ldots, \sigma_n$, representing each player's mixed strategy. The domain of each random variable σ_i is A_i.
- The edges of $\mathcal{B}^i_{a_i}$ are constructed by keeping all edges that go into the function nodes that are included in $\mathcal{B}$, ignoring edges that go into action nodes. Furthermore for each player j, we create an edge from σ_j to each of j's actions $a_j \in A_j$.
- The conditional probability distribution (CPD) at each function node p is just the deterministic function f^p. The CPD at each action node α' is a deterministic function that returns the number of its parents (observe that these are all mixed strategy nodes) that take the value α'. Mixed strategy nodes have no incoming edges; their (unconditional) probability distributions are the mixed strategies of the corresponding players, except for player i, whose node σ_i takes the deterministic value a_i.

It is straightforward to verify that $\mathcal{B}^i_{a_i}$ is a DAG, and that the joint distribution on random variables $\{c(m)\}_{m\in\nu(\alpha)}$ is exactly the distribution over configurations $\Pr(c^{(a_i)}|(a_i, \sigma^{(a_i)}_{-i}))$. This joint distribution can then be computed using a standard algorithm such as clique tree propagation or variable elimination. The running times of such algorithms are worst-case exponential; however, for Bayesian networks with bounded tree-width, their running times are polynomial.

Further speedups are possible at nodes in the induced Bayesian network that correspond to action nodes and contribution-independent function nodes. The deterministic CPDs at such nodes can be formulated using independent contributions from each player's strategy. This is an example of *causal independence* structure in Bayesian networks studied by Heckerman and Breese [1996] and Zhang and Poole [1996], who proposed different methods for exploiting such structure to speed up Bayesian network inference. Such methods share the common underlying idea of decomposing the CPDs into independent contributions, which is intuitively similar to our approach in Algorithm 1.

4.3 Computing Expected Payoff with AGG-FNAs

Due to the linearity of expectation, the expected utility of i playing an action a_i with an additive utility function with coefficients $(\lambda_m)_{m\in\nu(a_i)}$ is

$$V^i_{a_i}(\sigma_{-i}) = \sum_{m\in\nu(a_i)} \lambda_m E[c(m)|a_i, \sigma_{-i}], \tag{4.11}$$

where $E[c(m)|a_i, \sigma_{-i}]$ is the expected value of $c(m)$ given the strategy profile (a_i, σ_{-i}). Thus we can compute these expected values for each $m \in \nu(a_i)$, then sum them up as in Equation (4.11) to get the expected utility. If m is an action node, then $E[c(m)|a_i, \sigma_{-i}]$ is the expected number of players that chose m, which is $\sum_{i \in N} \sigma_i(m)$. The more interesting case is when m is a function node. Recall that $c(m) \equiv f^m(c^{(m)})$ where $c^{(m)}$ is the configuration over the neighbors of m. We can write the expected value of $c(m)$ as

$$E[c(m)|a_i, \sigma_{-i}] = \sum_{c^{(m)} \in C^{(m)}} f^m(c^{(m)}) \Pr(c^{(m)}|a_i, \sigma_{-i}). \tag{4.12}$$

This has the same form as Equation (4.5) for the expected utility $V^i_{a_i}(\sigma_{-i})$, except that we have f^m instead of u^α. Thus our results for the computation of Equation (4.5) also apply here. That is, if the neighbors of m are action nodes and/or contribution-independent function nodes, then $E[c(m)|a_i, \sigma_{-i}]$ can be computed in polynomial time.

Theorem 4.6 *Suppose u^α is represented as an additive utility function in a given AGG-FNA. If each of the neighbors of α is either (i) an action node, or (ii) a function node whose neighbors are action nodes and/or contribution-independent function nodes, then the expected utility $V^i_\alpha(\sigma_{-i})$ can be computed in time polynomial in the size of the representation. Furthermore, if the in-degrees of the neighbors of α are bounded by a constant, and the sizes of ranges $|\mathcal{R}(f^p)|$ for all $p \in \mathcal{P}$ are bounded by a polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$, then the expected utility can be computed in time polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$.*

It is straightforward to verify that our AGG-FNA representations of polymatrix games, congestion games, player-specific congestion games and the game in Example 3.4 all satisfy the conditions of Theorem 4.6.

5 Computing Sample Equilibria with AGGs

In this section we consider some theoretical and practical applications of our expected utility algorithm. In Section 5.1 we analyze the complexity of finding a sample ϵ-Nash equilibrium in an AGG and show that it is PPAD-complete. In Section 5.2 we extend our expected utility algorithm to the computation of payoff Jacobians, which is a key step in several algorithms for computing ϵ-Nash equilibria, including the Govindan-Wilson algorithm. In Section 5.3 we show that it can also speed up the simplicial subdivision algorithm, and in Section 5.4 we show that it can be used to find an ϵ-correlated equilibrium in polynomial time.[9]

5.1 Complexity of Finding a Nash Equilibrium

In this section we consider the complexity of finding a Nash equilibrium of an AGG. Since a Nash equilibrium for a game of more that two players may require irrational numbers in the probabilities, for practical computation it is necessary to consider approximations to Nash equilibria. Here we consider the frequently-used notion of ϵ-Nash equilibrium:

[9]Here we focus on the problems of finding a sample Nash or correlated equilibrium: in other words, identifying any single equilibrium. In games with multiple equilibria, it can be useful to find all equilibria or to find the "best" equilibrium according to some criterion. However, negative complexity results hold for both problems (see e.g. [Conitzer & Sandholm, 2008; Papadimitriou & Roughgarden, 2008; Shoham & Leyton-Brown, 2009]). For the case of bimatrix games, Mangasarian [1964] proposed an algorithm for enumerating all Nash equilibria, and Sandholm *et al.* [2005] proposed and evaluated a practical approach for computing optimal Nash equilibria using mixed-integer programming. Finding practical heuristic algorithms for these problems on n-player games remains an interesting open problem, but has received relatively little study to date.

Definition 5.1 (ϵ-Nash Equilibrium) *A mixed strategy profile σ is an ϵ-Nash equilibrium for some $\epsilon \geq 0$ if for all $i \in N$, for all $a_i \in A_i$, $u_i(\sigma) + \epsilon \geq u_i(a_i, \sigma_{-i})$*

Intuitively, each player cannot gain more than ϵ by deviating from her mixed strategy. For any game representation, define its NASH problem to be the problem of finding an ϵ-Nash equilibrium of a game encoded in that representation, for some ϵ given as part of the input. A series of recent papers [Chen & Deng, 2006; Goldberg & Papadimitriou, 2006; Daskalakis *et al.*, 2006b] have shown that the NASH problem for n-player normal-form games with $n \geq 2$ is complete for the complexity class PPAD, which is contained in NP but not known to be in P. Turning to compact representations, Daskalakis *et al.* [2006a] showed that the complexity of computing expected utility plays a vital role in the complexity of finding an ϵ-Nash equilibrium.

Definition 5.2 (Polynomial type [Daskalakis *et al.*, 2006a]) *A game representation has* polynomial type *if the number of agents n and the sizes of the action sets $|A_i|$ are bounded by a polynomial in the size of the representation.*

All AGG variants have polynomial type, since action sets are represented explicitly.

Theorem 5.3 ([Daskalakis *et al.*, 2006a]) *If a game representation satisfies the following properties: (1) the representation has polynomial type, and (2) expected utility can be computed using an arithmetic binary circuit with polynomial length, with nodes evaluating to constant values or performing addition, substraction, or multiplication on their inputs, then the NASH problem for this representation can be polynomially reduced to the NASH problem for some two-player, normal-form game.*

Since the NASH problem is in PPAD for two-player, normal-form games, the theorem implies that if the above properties hold, the NASH problem for such a compact game representation is in PPAD.

Corollary 5.4 *The complexity of NASH for AGG-$\emptyset$s is PPAD-complete.*

Remark. It may not be clear why this would be surprising or encouraging; indeed, the PPAD-hardness part of the claim is neither. However, the PPAD-membership part of the claim is a positive result. Specifically, it implies that the problem of finding a Nash equilibrium in an AGG-$\emptyset$ can be reduced to the problem of finding a Nash equilibrium in a two-player normal-form game with size polynomial in the size of the AGG-$\emptyset$. This is in contrast to the normal form representation of the original game, which can be exponentially larger than the AGG-$\emptyset$. In other words, if we instead try to solve for a Nash equilibrium using the normal form representation of the original game, we would face a PPAD-complete problem with an input exponentially larger than the AGG-$\emptyset$ representation.

> **Proof sketch** We first show that the problem belongs to PPAD, by constructing a circuit that computes expected utility and satisfies the second condition of Theorem 5.3.[10] Recall that our expected utility algorithm consists of Equation (4.4), then Algorithm 1, and finally Equation (4.5). Equations (4.4) and (4.5) can be straightforwardly translated into arithmetic circuits using addition and multiplication nodes. Algorithm 1 involves for loops that cannot be directly translated to an arithmetic circuit, but we observe that we can unroll the for loops and still end up with a polynomial number of operations. The resulting circuit resembles a lattice with n levels; at the k-th level there are $|C_k^{(a_i)}|$ addition nodes. Each addition node corresponds to a configuration $c_k^{(a_i)} \in C_k^{(a_i)}$, and calculates $P_k[c_k^{(a_i)}]$ as in iteration k of

Algorithm 1. Also there are $|A_k^{(a_i)}|$ multiplication nodes for each $c_k^{(a_i)}$, in order to carry out the multiplications in iteration k of Algorithm 1.

To show PPAD-hardness, we observe that an arbitrary graphical game can be encoded as an AGG-$\emptyset$ without loss of compactness (see Section 2.1). Thus the problem of finding a Nash equilibrium in a graphical game can be reduced to the problem of finding a Nash equilibrium in an AGG-$\emptyset$. Since finding a Nash equilibrium in a graphical game is known to be PPAD-hard, finding a Nash equilibrium in an AGG-$\emptyset$ is PPAD-hard. ■

For AGG-FNs that satisfy the conditions for Theorem 4.5 or AGG-FNAs that satisfy Theorem 4.6, similar arguments apply, and we can prove PPAD-completeness for those subclasses of games if we make the reasonable assumption that the operator $*$ used to define the CI function nodes can be implemented as an arithmetic circuit of polynomial length that satisfies the second condition of Theorem 5.3.

Although any Nash equilibrium is close to an ϵ-Nash equilibrium (in the space of mixed strategy profiles), a given ϵ-Nash equilibrium may be arbitrarily far from any Nash equilibrium of the game. Etessami and Yannakakis [2007] considered the stricter notion of approximate Nash equilibrium in the sense of being close to an exact Nash equilibrium, and showed that the problem of finding such a solution given a normal-form game is complete for the complexity class FIXP, which is contained in PSPACE but not known to be in NP. The "weaker" notion of ϵ-Nash equilibrium that we consider here is more amenable to practical computation: the corresponding complexity class PPAD is in NP, implying that checking whether a mixed-strategy profile is an ϵ-Nash equilibrium is in P (e.g., for AGGs this can be done straightforwardly by computing best responses using the expected utility algorithm).

5.2 Computing a Nash Equilibrium: The Govindan-Wilson Algorithm

Now we move from the theoretical to the practical. The PPAD-hardness result of Corollary 5.4 implies that a polynomial-time algorithm for Nash equilibrium is unlikely to exist, and indeed known algorithms for identifying sample Nash equilibria have worst-case exponential running times. Nevertheless, we will show that our dynamic programming algorithm for expected utility can be used to achieve exponential speedups in such algorithms, as well as an algorithm for computing a sample correlated equilibrium. Specifically, we show how to speed up key computations in the inner loops of these algorithms.

First we consider Govindan and Wilson's [2003] continuation method, a state-of-the-art method for finding mixed-strategy Nash equilibria in multi-player games. This algorithm starts by perturbing the payoffs to obtain a game with a known equilibrium. It then follows a path that is guaranteed to lead to at least one equilibrium of the original, unperturbed game. To take 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.

The payoff Jacobian under a mixed strategy σ is a $(\sum_i |A_i|) \times (\sum_i |A_i|)$ matrix with entries

$$\frac{\partial V^i_{a_i}(\sigma_{-i})}{\partial \sigma_{i'}(a_{i'})} \equiv \nabla V^{i,i'}_{a_i,a_{i'}}(\overline{\sigma}) \tag{5.1}$$

$$= \sum_{\overline{a} \in \overline{A}} u\left(a_i, \mathcal{C}(a_i, a_{i'}, \overline{a})\right) \Pr(\overline{a}|\overline{\sigma}). \tag{5.2}$$

[10]Observe that the second condition in Theorem 5.3 implies that the expected utility algorithm must take polynomial time; however, some polynomial algorithms (e.g., those that rely on division) do not satisfy this condition.

Here an overbar is shorthand for the subscript $-\{i, i'\}$ where $i \neq i'$ are two players; e.g., $\overline{a} \equiv a_{-\{i,i'\}}$. The rows of the matrix are indexed by i and a_i while the columns are indexed by i' and $a_{i'}$. Given entry $\nabla V^{i,i'}_{a_i,a_{i'}}(\overline{\sigma})$, we call a_i its *primary action node*, and $a_{i'}$ its *secondary action node*.

We note that efficient computation of the payoff Jacobian is important for more than simply Govindan and Wilson's 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 often much faster than the continuation method. Also, it can be 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.

5.2.1 Computing the Payoff Jacobian

Now we consider how the payoff Jacobian may be computed. Equation (5.2) shows that the $\nabla V^{i,i'}_{a_i,a_{i'}}(\overline{\sigma})$ element of the Jacobian can be interpreted as the expected utility of agent i when she takes action a_i, agent i' takes action $a_{i'}$, and all other agents use mixed strategies according to $\overline{\sigma}$. So a straightforward—and quite effective—approach is to use our expected utility algorithm to compute each entry of the Jacobian.

However, the Jacobian matrix has certain extra structure that allows us to achieve further speedup. For example, observe that some entries of the Jacobian are identical. If two entries have the same primary action node α, then they are expected payoffs on the same utility function u^{α}, and so have the same values if their induced probability distributions over $C^{(\alpha)}$ are the same. We need to consider two cases:

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

   (a) The columns of the two entries belong to the same player j, but different actions a_j and a'_j. If $a_j^{(a_i)} = {a'}_j^{(a_i)}$, i.e., a_j and a'_j both project to the same projected action in a_i's projected action graph,[11] then $\nabla V^{i,j}_{a_i,a_j} = \nabla V^{i,j}_{a_i,a'_j}$. This implies that when $a_j, a'_j \notin \nu(a_i)$, $\nabla V^{i,j}_{a_i,a_j} = \nabla V^{i,j}_{a_i,a'_j}$.

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

2. If a_i and a_j correspond to the same action node α (but owned by agents i and j respectively), thus sharing the same payoff function u^{α}, then $\nabla V^{i,j}_{a_i,a_j} = \nabla V^{j,i}_{a_j,a_i}$. Furthermore, if there exist $a'_i \in A_i, a'_j \in A_j$ such that ${a'}_i^{(\alpha)} = {a'}_j^{(\alpha)}$ (or $\delta^{(\alpha)}_{a'_i} = \delta^{(\alpha)}_{a'_j}$ for contribution-independent AGG-FNs), then $\nabla V^{i,j}_{a_i,a'_j} = \nabla V^{j,i}_{a_j,a'_i}$.

[11] For contribution-independent AGG-FNs, the condition becomes $\delta^{(a_i)}_{a_j} = \delta^{(a_i)}_{a'_j}$, i.e., a_j and a'_j have the same projected contribution under a_i.

A consequence of 1(a) is that any Jacobian of an AGG has at most $\sum_i \sum_{a_i \in A_i} (n-1)(\nu(a_i)+1)$ distinct entries. For AGGs with bounded in-degree, this is $O(n \sum_i |A_i|)$. For each set of identical entries, we only need to do the expected utility computation once. Even when two entries in the Jacobian are not identical, we can exploit the similarity of the projected strategy profiles (and thus the similarity of the induced distributions) between entries, reusing 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, a_i) and the column's player j, then $\overline{\sigma}$ is the same for all secondary actions $a_j \in A_j$. We can compute the probability distribution $\Pr(c_{n-1}|a_i, \overline{\sigma}^{(a_i)})$, then for all $a_j \in A_j$, we just need to apply the action a_j to get the induced probability distribution for the entry $\nabla V^{i,j}_{a_i,a_j}$.

Now suppose we fix the row (i, a_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 AGG-$\emptyset$s, we can exploit this similarity by computing the distribution $\Pr(c_{n-1}|\sigma_{-i}^{(a_i)})$, then for each $j \neq i$, we "undo" j's mixed strategy to get the distribution induced by $\sigma_{-\{i,j\}}$, by treating distributions $\Pr(c_{n-1}|\sigma_{-i}^{(a_i)})$ and σ_j as coefficients of polynomials and computing their quotient using long division. (See Section 2.3.5 of [Jiang, 2006] for a more detailed discussion of interpreting distributions over configurations as polynomials.)

5.2.2 Finding equilibria of symmetric and k-symmetric games

Nash proved [1951] that all finite symmetric games have at least one symmetric Nash equilibrium. The Govindan-Wilson algorithm can be adapted to find symmetric Nash equilibria in symmetric AGG-$\emptyset$s. The modified algorithm now operates in the space of symmetric mixed strategy profiles, and follows a path of symmetric equilibria of perturbed symmetric games to a symmetric equilibrium of the unperturbed game. The corresponding payoff Jacobian under σ_* is now a $|\mathcal{A}| \times |\mathcal{A}|$ matrix whose entry at row α and column α' is $n-1$ multiplied by the expected utility of a player choosing action α, when another player is choosing action α' and the rest of the players play mixed strategy σ_*. Such an entry can be efficiently computed using the techniques for symmetric expected utility computation discussed in Section 4.1.4, which are faster than our expected utility algorithm for general AGGs. Techniques discussed in the current section can further be used to speed up the computation of Jacobians in the symmetric case. In particular, it is straightforward to check that the Jacobian has at most $\sum_{\alpha \in \mathcal{A}} (\nu(\alpha)+1) = O(|E|)$ identical entries, where E is the set of edges of the action graph.

A straightforward corollary of Nash's [1951] proof is that any k-symmetric AGG-$\emptyset$ has at least one k-symmetric Nash equilibrium. Relying on similar arguments as above, we can adapt the Govindan-Wilson algorithm to find k-symmetric equilibria in k-symmetric AGG-$\emptyset$s. The bottleneck is the computation of the k-symmetric version of payoff Jacobians, the entries of which can be shown to be equal to constant multiples of certain expected utilities. Such expected utilities can be efficiently computed using the techniques discussed in Section 4.1.5.

5.3 Computing a Nash Equilibrium: The Simplicial Subdivision Algorithm

Another algorithm for computing a sample Nash equilibrium is van der Laan, Talman & van der Heyden's [1987] simplicial subdivision algorithm, which is derived from Scarf's [1967] algorithm for computing fixed points. At a high level, the algorithm does the following.

1. The space of mixed strategy profiles $\Sigma = \prod_i \Sigma_i$ is partitioned into a set of subsimplexes.

2. We assign labels to vertices of the subsimplexes, in a way such that a "completely labeled" subsimplex corresponds to an approximate Nash equilibrium.
3. The algorithm follows a path of "almost completely labeled" subsimplexes, and eventually reaches a "completely labeled" subsimplex.
4. Such an approximate equilibrium can be refined by restarting the algorithm near the approximate equilibrium using a finer grid.

At each step along the path, the algorithm needs to compute labels of the subsimplex, which in turn depends on computation of expected utilities under mixed strategy profiles. By using our AGG-based Algorithm 2 for computing expected utility, this step can be sped up exponentially, as compared to the normal-form-based implementation.

5.4 Computing a Correlated Equilibrium

Papadimitriou and Roughgarden [2008] proposed a polynomial-time algorithm for computing a sample correlated equilibrium given a game representation with polynomial type and a polynomial-time subroutine for computing expected utility under mixed strategy profiles. Recently, Stein *et al.* [2010] showed that Papadimitriou and Roughgarden's algorithm can fail to find an exact correlated equilibrium, and presented a slight modification of the algorithm that efficiently computes an ϵ-correlated equilibrium. (An ϵ-correlated equilibrium is an approximation of the correlated equilibrium solution concept, where ϵ measures the extent to which the incentive constraints for correlated equilibrium are violated.) Incorporating this fix, we have the following.

Theorem 5.5 ([Papadimitriou & Roughgarden, 2008]) *If a game representation has polynomial type, and has a polynomial algorithm for computing expected utility, then an ϵ-correlated equilibrium can be computed in time polynomial in $\log \frac{1}{\epsilon}$ and the representation size.*

The reader might wonder why it is difficult to compute correlated equilibrium—even in exact form—given that there is a well-known linear programming formulation for computing a correlated equilibrium. The catch is that this LP has one variable for each action profile. Thus, while it amounts to a polynomial-time algorithm for games represented in normal form, its size is exponential in the size of any compact representation for which the simple algorithm for computing expected utility given by Equation 4.1 is inadequate. Indeed, in these cases even *describing* a correlated equilibrium using these probabilities of action profiles can require exponential space. Papadimitriou and Roughgarden's results are thus much deeper than they may first seem. The algorithm outputs an ϵ-correlated equilibrium represented as the mixture of a polynomial number of product distributions over action profiles. Since the theorem requires that the game representation has polynomial type, this polynomial mixture of product distributions can also be represented polynomially.

The second condition in this theorem involves the computation of expected utility. As a direct corollary of Theorem 5.5 and our own Theorem 4.1, there exists a polynomial algorithm for computing an ϵ-correlated equilibrium given an AGG-$\emptyset$.

Corollary 5.6 *Given a game represented as an AGG-$\emptyset$, an ϵ-correlated equilibrium can be computed in time polynomial in $\log \frac{1}{\epsilon}$ and the size of the AGG-$\emptyset$.*

Similarly, for AGG-FNs and AGG-FNAs for which the expected utility problem can be solved in polynomial time (see Theorems 4.5 and 4.6), ϵ-correlated equilibria can be computed in polynomial time.

6 Experiments

Although our theoretical results show that there are significant benefits to working with AGGs, they might leave the reader with two worries. First, the reader might be concerned that while AGGs offer asymptotic computational benefits, they might not be practically useful. Second, even if convinced about the usefulness of AGGs, the reader might want to know the size of problems that can be tackled by the computational tools we have developed so far. We address both of these worries in this section, by reporting on the results of extensive computational experiments. Specifically, we compare the performance of the AGG representation and our AGG-based algorithms against normal-form-based solutions using the (highly optimized) GameTracer package [Blum *et al.*, 2002]. As benchmarks, we used AGG and normal-form representations of instances of Coffee Shop games, Job Market games, and symmetric AGG-$\emptyset$s on random graphs. We compared the representation sizes of AGG and normal-form representations, and compared their performance resulting from using these representations to compute expected utility, to compute Nash equilibria using the Govindan-Wilson algorithm, and to compute Nash equilibria using the simplicial subdivision algorithm. Finally, we show how sample equilibria of these games can be visualized on action graphs.

6.1 Software Implementation and Experimental Setup

We implemented our algorithms in a freely-available software package, in order to make it easy for other researchers to use AGGs to model problems of interest. Our software is capable of:

- reading in a description of an AGG;
- computing expected utility and Jacobian given mixed strategy profile;
- computing Nash equilibria by adapting GameTracer's [Blum *et al.*, 2002] implementation of Govindan and Wilson's [2003] continuation method; and
- computing Nash equilibria by adapting GAMBIT's [McKelvey *et al.*, 2006] implementation of the simplicial subdivision algorithm [van der Laan *et al.*, 1987].

We extended GAMUT [Nudelman *et al.*, 2004], a suite of game instance generators, by implementing generators of instances of AGGs including Ice Cream Vendor games (Example 2.5), Coffee Shop games (Example 2.7), Job Market games (Example 3.1) and symmetric AGG-$\emptyset$s on a random action graph with random payoffs. Finally, with Damien Bargiacchi, we also developed a graphical user interface for creating and editing AGGs. All of our software is freely available at `http://agg.cs.ubc.ca`.

When using Coffee Shop games in our experiments, we set payoffs randomly in order to test on a wide set of utility functions. For the visualization of equilibria in Section 6.7 we set the Coffee Shop game utility functions to be

$$u^\alpha(c(\alpha), c(p'_\alpha), c(p''_\alpha)) = 20 - [c(\alpha)]^2 - c(p'_\alpha) - \log(c(p''_\alpha) + 1),$$

where p'_α is the function node representing the number of players choosing adjacent locations and p''_α is the function node representing the number of players choosing other locations.

When using Job Market games in our experiments, we set the utility functions to be

$$u^\alpha(c^{(\alpha)}) = \frac{R_\alpha}{c(\alpha) + \sum_{\alpha' \in \nu(\alpha) - \{\alpha\}} 0.1 c(\alpha')} - K_\alpha,$$

with R_α set to $2, 4, 6, 8, 10$ and K_α set to $1, 2, 3, 4, 5$ for the five levels from high school to PhD.

When using Ice Cream Vendor games for the visualization of equilibria in Section 6.7 we set the utilities so that for a player i choosing action α, each vendor choosing a location $\alpha' \in \nu(\alpha)$ contributes $w_f w_l$ utility to i. w_f is -1 when α' has the same food type as α, and 0.8 otherwise. w_l is 1 when α' and α correspond to the same location, and 0.6 when they correspond to different (but neighboring) locations. In other words, there is a negative effect from players choosing the same food type, and a weaker positive effect from players choosing a different food type. Furthermore, effects from neighboring locations are weaker than effects from the same location.

All our experiments were performed using a computer cluster consisting of 55 machines with dual Intel Xeon 3.2GHz CPUs, 2MB cache and 2GB RAM, running Suse Linux 10.1.

6.2 Representation Size

First, we compared the representation sizes of AGG-FNs and their induced normal forms. For each game instance we counted the number of payoff values that needed to be stored.

We first looked at 5×5 block Coffee Shop games, varying the number of players. Figure 9 (left) 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 became impractical. The size of the AGG representation grew polynomially with respect to n. As we can see from Figure 9 (right), even for a game instance with 80 players, the AGG-FN representation stored only about 2 million numbers. In contrast, the corresponding normal form representation would have had to store 1.2×10^{115} numbers.

We then fixed the number of players at 4 and varied the number of actions; for ease of comparison we fixed the number of columns at 5 and only changed the number of rows. Recall from Section 2.2.1 that the representation size of Coffee Shop games—expressed both as AGGs and in the normal form—depends only on the number of players and number of actions, but not on the shape of the region. (Recall that the number of actions is $B + 1$, where B is the total number of blocks.) Figure 9 (left) shows a log-scale plot of the number of payoff values versus the number of actions, and Figure 9 (right) gives a plot for just the AGG-FN representation. 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. For a Coffee Shop game with 4 players on an 80×5 grid, the AGG-FN representation stores only about 8000 numbers, whereas the normal form representation would have to store 1.0×10^{11} numbers.

We also tested on Job Market games from Example 3.1, which have 13 actions. We varied the number of players from 3 to 24. The results are similar, as shown in Figure 11 (left). This is consistent with our theoretical observation that the sizes of normal form representations grow exponentially in n while the sizes of AGG representations grow polynomially in n.

6.3 Expected Utility Computation

We tested the performance of our dynamic programming algorithm for computing expected utilities in AGG-FNs against GameTracer's normal-form-based algorithm for computing expected utilities. For each game instance, we generated 1000 random strategy profiles with full support, and measured the CPU (user) time spent computing $V^n_{a_n}(\sigma_{-n})$ under these strategy profiles. Then we divided this measurement by 1000 to obtain the average CPU time.

We first looked at Coffee Shop games of different sizes. We fixed the size of blocks at 5×5 and varied the number of players. Figure 10 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 AGG-based algorithm's running time grows polynomially, while the

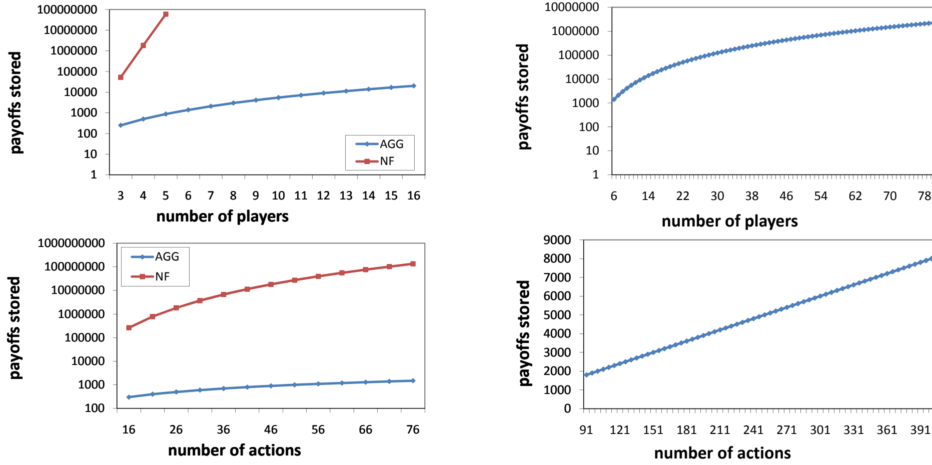

Figure 9: Representation sizes of coffee shop games. Top left: 5×5 grid with 3 to 16 players (log scale). Top right: AGG only, 5×5 grid with up to 80 players (log scale). Bottom left: 4-player $r \times 5$ grid, r varying from 3 to 15 (log scale). Bottom right: AGG only, up to 80 rows.

normal-form-based algorithm scales exponentially. For more than five players, we were not able to store the normal form representation in memory. Meanwhile, our AGG-based algorithm scaled to much larger numbers of players, averaging about a second to compute an expected utility for an 80-player Coffee Shop game.

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

We also 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 utilities for the Coffee Shop game instances mentioned above, while our AGG-based algorithm required about 20% of its full-support running times.

We also tested on Job Market games, varying the numbers of players. The results are shown in Figure 11 (right). The normal-form-based implementation ran out of memory for more than 6 players, while the AGG-based implementation averaged about a quarter of a second to compute expected utility in a 24-player game.

6.4 Computing Payoff Jacobians

We ran similar experiments to investigate the computation of payoff Jacobians. As discussed in Section 5.2, 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 entries. In Section 5.2 we discussed methods that exploit the structure of the Jacobian to further speed up the computation. GameTracer's normal-form-based implementation also exploits the structure of the Jacobian by reusing partial results of expected payoff computations. When comparing our AGG-based Jacobian algorithm (as described in Section 5.2) to GameTracer's implementation, we observed results very similar to those for computing expected payoffs: our implementation scaled polyno-

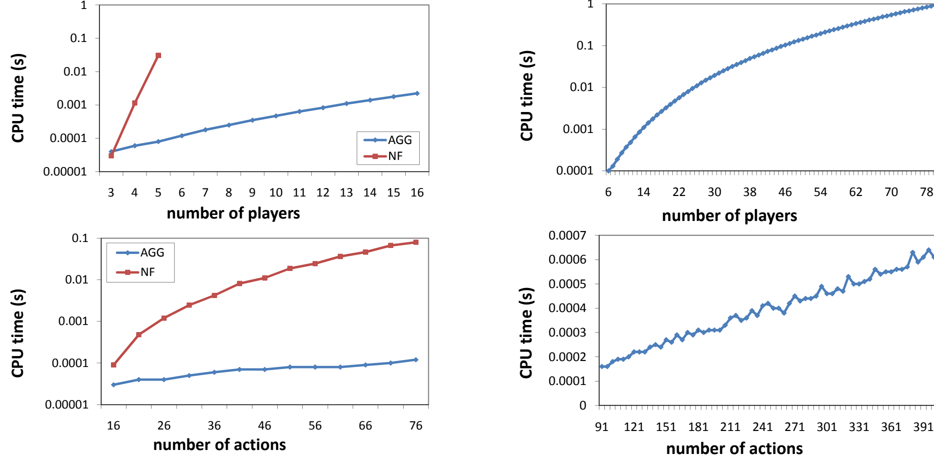

Figure 10: Running times for payoff computation in the Coffee Shop game. Top left: 5×5 grid with 3 to 16 players. Top right: AGG only, 5×5 grid with up to 80 players. Bottom left: 4-player $r \times 5$ grid, r varying from 3 to 15. Bottom right: AGG only, up to 80 rows.

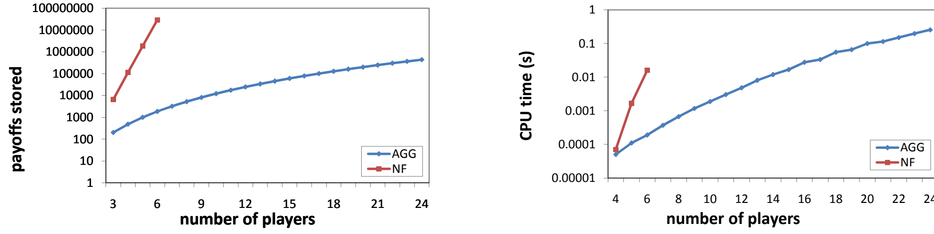

Figure 11: Job Market games, varying numbers of players. Left: comparing representation sizes. Right: running times for computing 1000 expected utilities.

mially in n while GameTracer scaled exponentially in n. We instead focus on the question of how much speedup the methods in Section 5.2 provided, by comparing our algorithm in Section 5.2 against the algorithm that computes expected payoffs (using our AGG-based algorithm described in Section 4) for each of the Jacobian's entries. We tested on Coffee Shop games on a 5×5 grid with 3 to 10 players, as well as Coffee Shop games with 4 players, 5 columns and varying numbers of rows. For each instance of the game we randomly generated 100 strategy profiles with partial support. For each of these game instances, our algorithm as described in Section 5.2 was consistently about 50 times faster than computing expected payoffs for each of the Jacobian's entries. This confirms that the methods discussed in Section 5.2 provide significant speedup for computing payoff Jacobians.

6.5 Finding a Nash Equilibrium using the Govindan-Wilson algorithm

Now we show experimentally that the speedup we achieved for computing Jacobians using the AGG representation led to a speedup in the Govindan-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 follow the same path and return exactly the same Nash equilibrium.

Again, we tested the two algorithms on Coffee Shop games of varying sizes: first we fixed the sizes 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 ran the two versions of the Govindan-Wilson algorithm. Although the algorithm can (sometimes) find more than one equilibrium, we stopped both versions of the algorithm after one equilibrium was found. Since the running time of the Govindan-Wilson algorithm is very sensitive to the initial perturbation, for each game instance the running times with different initial perturbations had large variance. To control for this, for each initial perturbation we looked at the *ratio* of running times between the normal-form implementation and the AGG implementation (i.e., a ratio greater than 1 means the AGG implementation ran more quickly than the normal form implementation). We present the results in Figure 12 (left). We see that as the size of the games grew (either in the number of players or in the number of actions), the speedup of the AGG implementation over that of the normal-form implementation increased. The normal-form implementation ran out of memory for game instances with more than 5 players, preventing us from reporting ratios above $n = 5$. Thus, we ran the AGG-based implementation alone on game instances with larger numbers of players, giving the algorithm a one-day cutoff time. As shown by the log-scale boxplot of CPU times in Figure 12 (top right), for game instances with up to 12 players, the algorithm terminated within one day for most initial perturbations. A normal form representation of such a game would have needed to store 7.0×10^{15} numbers. Figure 12 (bottom right) shows a boxplot of the CPU times for the AGG-based implementation, varying the number of actions while fixing the number of players at 4. For game instances with up to 49 actions (a 4×12 grid plus one action for not entering the market), the algorithm terminated within an hour.

We also tested on Job Market games with varying numbers of players. The results are shown in Figure 13. For the game instance with 6 players, the AGG-based implementation was about 100 times faster than the normal-form-based implementation. While the normal-form-based implementation ran out of memory for Job Market games with more than 6 players, the AGG-based implementation was able to solve games with 16 players in an average of 24 minutes.

6.6 Finding a Nash Equilibrium using Simplicial Subdivision

As discussed in Section 5.3, we can speed up the normal-form-based simplicial subdivision algorithm by replacing the subroutine that computes expected utility by our AGG-based algorithm. We have done so to GAMBIT's implementation of simplicial subdivision. As with the Govindan-Wilson algorithm, from a given starting point both the original version of simplicial subdivision and our AGG version follow a deterministic path to determine exactly the same equilibrium. Thus, all performance differences are due to the choice of representation. We compared the performance of AGG-based simplicial subdivision against normal-form-based simplicial subdivision on instances of Coffee Shop games as well as instances of randomly-generated symmetric AGG-$\emptyset$s on small world graphs. We always started from the mixed strategy profile in which each player gives equal probability to each of her actions.

We first considered instances of Coffee Shop games with 4 rows, 4 columns and varying numbers of players. For each game size we generated 10 instances with random payoffs. Figure 14 (left) gives a boxplot of the ratio of running times between the two implementations. The AGG-based implementation was about 3 times faster for the 3-player instances and about 30

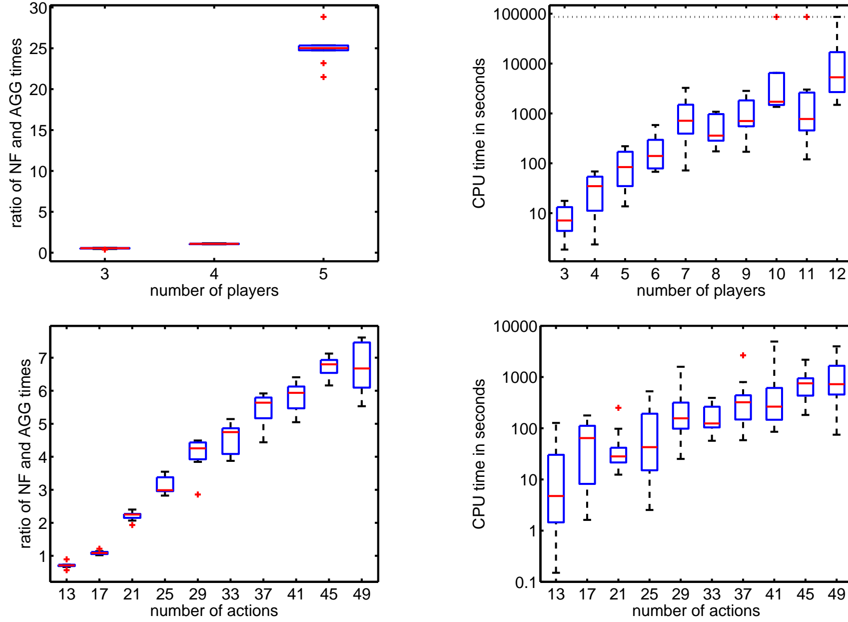

Figure 12: Govindan-Wilson algorithm; Coffee Shop game. Top row: 4×4 grid, varying number of players. Bottom row: 4-player $r \times 4$ grid, r varying from 3 to 12. For each row, the left figure shows ratio of running times; the right figure shows logscale plot of CPU times for the AGG-based implementation. The dashed horizontal line indicates the one day cutoff time.

times faster for the 4-player instances. We also tested on Coffee Shop games with 3 players, 3 columns and numbers of rows varying from 4 to 7, again generating 10 instances with random payoffs at each size. Figure 14 (right) gives a boxplot of the ratio of running times. As expected, the AGG-based implementation was faster and the gap in performance widened as games grew.

We then investigated symmetric AGG-$\emptyset$s on randomly generated small world graphs with random payoffs. The small world graphs were generated using GAMUT's implementation with parameters $K = 1$ and $p = 0.5$. For each game size we generated 10 instances. We first fixed the number of action nodes at 5 and varied the number of players. Results are shown in Figure 15 (top row). While there was large variance in the absolute running times across different instances, the ratios of running times between normal-form-based and AGG-based implementations showed a clear increasing trend as the number of players increased. The normal-form-based implementation ran out of memory for instances with more than 5 players. Meanwhile, we ran the AGG-based implementation on larger instances with a one-day cutoff time. As shown by the boxplot, the AGG-based implementation solved most instances with up to 8 players within 24 hours. We then fixed the number of players at 4 and varied the number of action nodes from 4 to 16. Results are shown in Figure 15 (bottom row). Again, while the actual running times on different instances varied substantially, the ratios of running times showed a clear increasing trend as the number of actions increased. The AGG-based implementation was able to solve a 16-action instance in an average of about 3 minutes, while the normal-form-based implementation averaged about 2 hours.

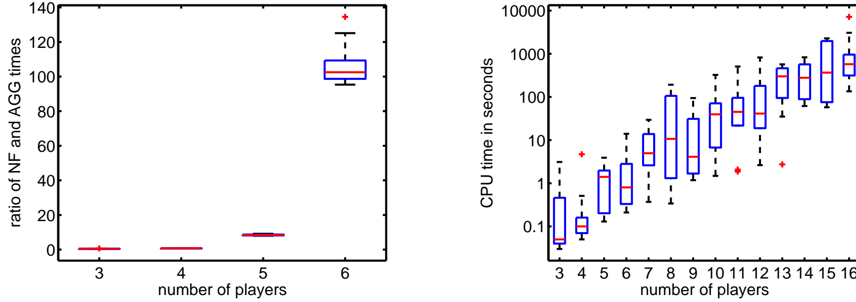

Figure 13: Govindan-Wilson algorithm; Job Market games, varying numbers of players. Left: ratios of running times. Right: logscale plot of CPU times for the AGG-based implementation.

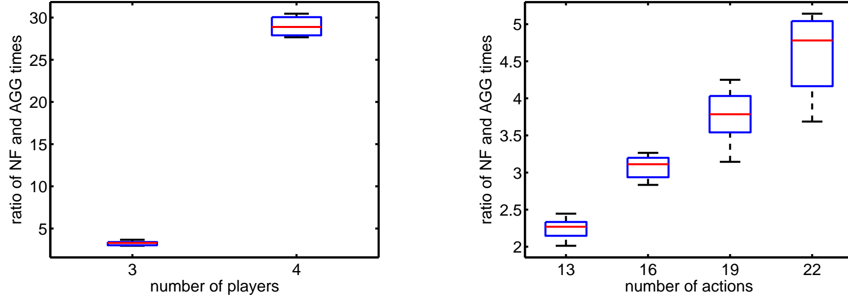

Figure 14: Ratios of running times of simplicial subdivision algorithms on Coffee Shop games. Left: 4×4 grid with 3 to 4 players. Right: 3-player $r \times 3$ grid, r varying from 4 to 7.

6.7 Visualizing Equilibria on the Action Graph

Besides facilitating representation and computation, the action graph can also be used to visualize strategy profiles in a natural way. A strategy profile σ (e.g., a Nash equilibrium) can be visualized on the action graph by displaying the expected numbers of players that choose each of the actions. We call such a tuple the *expected configuration* under σ. This can be easily computed given σ: for each action node α, we sum the probabilities of playing α, i.e. $E[c(\alpha)] = \sum_{i \in N} \sigma_i(\alpha)$ where $\sigma_i(\alpha)$ is 0 when $\alpha \notin A_i$. When the strategy profile consists of pure strategies, the result is simply the corresponding configuration.

The expected configuration often has natural interpretations. For example in Coffee Shop games and other scenarios where actions correspond to location choices, an expected configuration can be seen as a density map describing expected player locations. We illustrate using a 16-player Coffee Shop game on a 4×4 grid. We ran the (AGG-based) Govindan-Wilson algorithm, finding a Nash equilibrium in 77 seconds. The expected configuration of this (pure strategy) equilibrium is visualized in Figure 16.

We also examined a Job Market game with 20 players. A normal form representation of this game would have needed to store 9.4×10^{134} numbers. We ran the AGG-based Govindan-Wilson algorithm, finding a Nash equilibrium in 860 seconds. The expected configuration of this equilibrium is visualized in Figure 17 (left). Note that the equilibrium expected configuration on some of the nodes are non-integer values, as a result of mixed strategies by some of the players. We also visualize two players' mixed equilibrium strategies in Figure 17 (right).

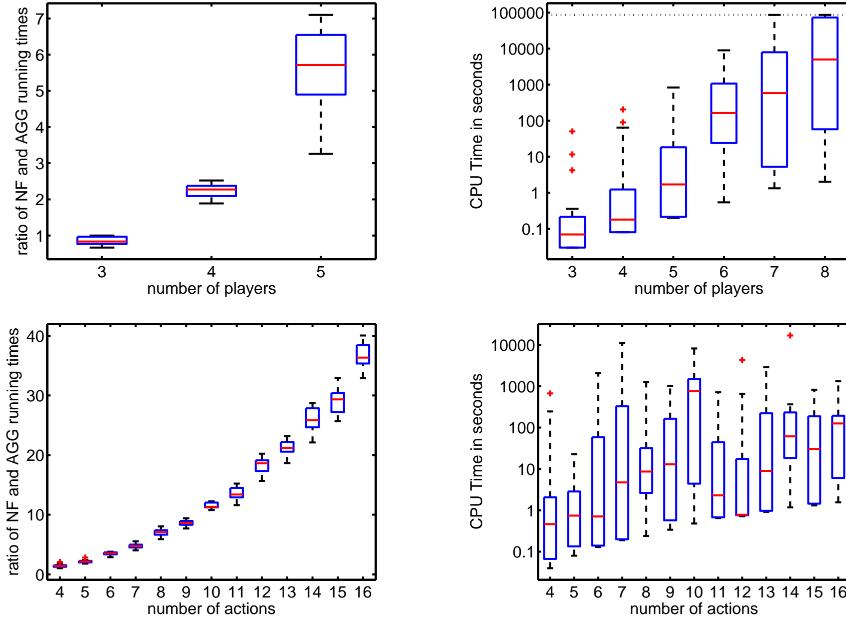

Figure 15: Simplicial subdivision algorithm; symmetric AGG-∅s on small world graphs. Top row: 5 actions, varying number of players. Bottom row: 4 players, varying number of actions. The left figures show ratios of running times; the right figures show logscale plots of CPU times for the AGG-based implementation. The dashed horizontal line indicates the one day cutoff time.

Finally, we examined an Ice Cream Vendor game (Example 2.5) with 4 locations, 6 ice cream vendors, 6 strawberry vendors, and 4 west-side vendors. The Govindan-Wilson algorithm found an equilibrium in 9 seconds. The expected configuration of this (pure strategy) equilibrium is visualized in Figure 18. Observe that the west side is relatively denser due to the west-side vendors. The locations at the east and west ends were chosen relatively more often than the middle locations, because the ends have fewer neighbors and thus experience less competition.

7 Conclusions

We proposed action-graph games (AGGs), a fully expressive game representation that can compactly express utility functions with structure such as context-specific independence and anonymity. We also extended the basic AGG representation by introducing function nodes and additive utility functions, allowing us to compactly represent a wider range of structured utility functions. We showed that AGGs can efficiently represent games from many previously studied compact classes including graphical games, symmetric games, anonymous games, and congestion games. We presented a polynomial-time algorithm for computing expected utilities in AGG-∅s and contribution-independent AGG-FNs. For symmetric and k-symmetric AGG-∅s, we gave more efficient, specialized algorithms for computing expected utilities under symmetric and k-symmetric strategy profiles respectively. We also showed how to use these algorithms to achieve exponential speedups of existing methods for computing a sample Nash equilibrium and a sample correlated equilibrium. We showed experimentally that using AGGs allows us to model and

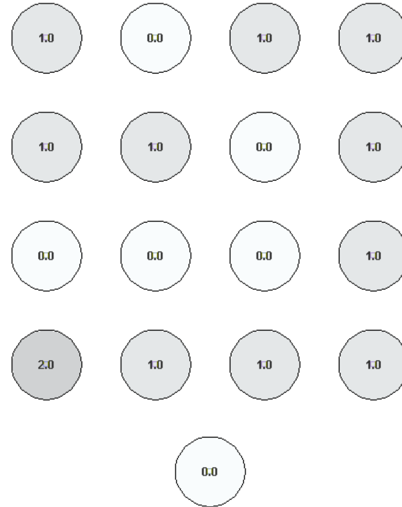

Figure 16: Visualization of a Nash equilibrium of a 16-player Coffee Shop game on a 4×4 grid. The function nodes and the edges of the action graph are not shown. The action node at the bottom corresponds to not entering the market.

analyze dramatically larger games than can be addressed with the normal-form representation.

We briefly mention a few of our current and future research directions. We are currently exploring applications of AGGs for modeling and analyzing large real-world systems, and have preliminary results for network routing problems [Thompson *et al.*, 2007] and complete-information advertising auction problems [Thompson & Leyton-Brown, 2009]. Another interesting problem is the computation of pure-strategy Nash equilibria in AGG-∅s. While the problem is NP-complete in general (Conitzer, personal communication; independently proved by Daskalakis *et al.* [2009]), in Jiang and Leyton-Brown [2007] we presented a polynomial time algorithm for the class of symmetric AGG-∅s whose action graphs have bounded in-degree and bounded tree-width. We are currently extending this algorithm to classes of asymmetric AGG-∅s and AGG-FNs. Another line of research is to extend our AGG framework beyond complete-information simultaneous-move games. In Jiang and Leyton-Brown [2010] we proposed Bayesian action-graph games (BAGGs) for representing Bayesian games, and in Jiang *et al.* [2009] we proposed temporal action-graph games (TAGGs) for representing imperfect-information dynamic games. In both cases, we showed that our representations can compactly represent games with anonymity or context-specific independencies; we also provided efficient algorithms for computing expected utility, which lead to speedups in the computation of Bayes-Nash equilibria and behavior-strategy Nash equilibria, respectively.

References

Ben-Sasson, E., Kalai, A., & Kalai, E. (2006). An approach to bounded rationality. *NIPS: Proceedings of the Neural Information Processing Systems Conference* (pp. 145–152).

Bhat, N., & Leyton-Brown, K. (2004). Computing Nash equilibria of action-graph games. *UAI: Proceedings of the Conference on Uncertainty in Artificial Intelligence* (pp. 35–42).

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

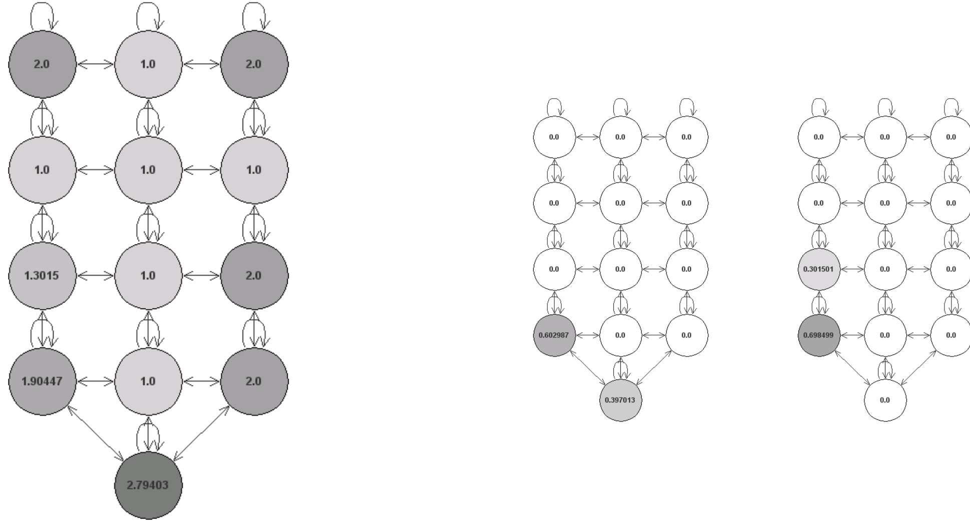

Figure 17: Visualization of a Nash equilibrium of a Job Market game with 20 players. Left: expected configuration of the equilibrium. Right: two mixed equilibrium strategies.

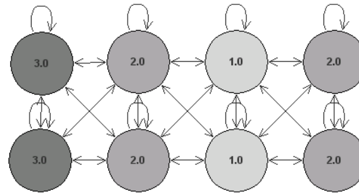

Figure 18: Visualization of a Nash equilibrium of an Ice Cream Vendor game.

Blum, B., Shelton, C., & Koller, D. (2006). A continuation method for Nash equilibria in structured games. *JAIR: Journal of Artificial Intelligence Research*, *25*, 457–502.

Brandt, F., Fischer, F., & Holzer, M. (2010). Equilibria of graphical games with symmetries. *Theoretical Computer Science*. Accepted subject to minor revisions.

Chen, X., & Deng, X. (2006). Settling the complexity of 2-player Nash-equilibrium. *FOCS: Proceedings of the Annual IEEE Symposium on Foundations of Computer Science* (pp. 261–272).

Conitzer, V., & Sandholm, T. (2008). New complexity results about nash equilibria. *Games and Economic Behavior*, *63*(2), 621 – 641. Second World Congress of the Game Theory Society.

Daskalakis, C., Fabrikant, A., & Papadimitriou, C. (2006a). The game world is flat: The complexity of Nash equilibria in succinct games. *ICALP: Proceedings of the International Colloquium on Automata, Languages and Programming* (pp. 513–524).

Daskalakis, C., Goldberg, P. W., & Papadimitriou, C. H. (2006b). The complexity of computing a Nash equilibrium. *STOC: Proceedings of the Annual ACM Symposium on Theory of Computing* (pp. 71–78).

Daskalakis, C., & Papadimitriou, C. (2006). Computing pure Nash equilibria via Markov random fields. *EC: Proceedings of the ACM Conference on Electronic Commerce* (pp. 91–99).

Daskalakis, C., & Papadimitriou, C. (2007). Computing equilibria in anonymous games. *FOCS: Proceedings of the Annual IEEE Symposium on Foundations of Computer Science* (pp. 83–93).

Daskalakis, C., Schoenebeck, G., Valiant, G., & Valiant, P. (2009). On the complexity of Nash equilibria of Action-Graph Games. *SODA: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (pp. 710–719).

Elkind, E., Goldberg, L., & Goldberg, P. (2006). Nash equilibria in graphical games on trees revisited. *EC: Proceedings of the ACM Conference on Electronic Commerce*, 100–109.

Elkind, E., Goldberg, L., & Goldberg, P. (2007). Computing good Nash equilibria in graphical games. *EC: Proceedings of the ACM Conference on Electronic Commerce*, 162–171.

Etessami, K., & Yannakakis, M. (2007). On the Complexity of Nash Equilibria and Other Fixed Points (Extended Abstract). *FOCS: Proceedings of the Annual IEEE Symposium on Foundations of Computer Science* (pp. 113–123).

Fredkin, E. (1962). Trie memory. *Communications of the ACM*, *3*, 490–499.

Goldberg, P. W., & Papadimitriou, C. H. (2006). Reducibility among equilibrium problems. *STOC: Proceedings of the Annual ACM Symposium on Theory of Computing* (pp. 61–70).

Gottlob, G., Greco, G., & Scarcello, F. (2005). Pure Nash equilibria: Hard and easy games. *Journal of Artificial Intelligence Research*, *24*, 357–406.

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

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

Heckerman, D., & Breese, J. S. (1996). Causal independence for probability assessment and inference using Bayesian networks. *IEEE Transactions on Systems, Man and Cybernetics*, *26*(6), 826–831.

Hotelling, H. (1929). Stability in competition. *Economic Journal*, *39*, 41–57.

Ieong, S., McGrew, R., Nudelman, E., Shoham, Y., & Sun, Q. (2005). Fast and compact: A simple class of congestion games. *AAAI: Proceedings of the AAAI Conference on Artificial Intelligence*, 489–494.

Jiang, A. X. (2006). Computational problems in multiagent systems. Master's thesis, University of British Columbia.

Jiang, A. X., & Leyton-Brown, K. (2006). A polynomial-time algorithm for Action-Graph Games. *AAAI: Proceedings of the AAAI Conference on Artificial Intelligence* (pp. 679–684).

Jiang, A. X., & Leyton-Brown, K. (2007). Computing pure Nash equilibria in symmetric Action-Graph Games. *AAAI: Proceedings of the AAAI Conference on Artificial Intelligence* (pp. 79–85).

Jiang, A. X., & Leyton-Brown, K. (2010). Bayesian action-graph games. *NIPS: Proceedings of the Neural Information Processing Systems Conference*. in press.

Jiang, A. X., Pfeffer, A., & Leyton-Brown, K. (2009). Temporal Action-Graph Games: A new representation for dynamic games. *UAI: Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

Kalai, E. (2004). Large robust games. *Econometrica*, *72*(6), 1631–1665.

Kalai, E. (2005). Partially-specified large games. *WINE: Proceedings of the Workshop on Internet and Network Economics* (pp. 3–13).

Kearns, M. (2007). Graphical games. In N. Nisan, T. Roughgarden, E. Tardos and V. Vazirani (Eds.), *Algorithmic game theory*, chapter 7, 159–180. Cambridge, UK: Cambridge University Press.

Kearns, M., Littman, M., & Singh, S. (2001). Graphical models for game theory. *UAI: Proceedings of the Conference on Uncertainty in Artificial Intelligence* (pp. 253–260).

Kearns, M., & Suri, S. (2006). Networks Preserving Evolutionary Stability and the Power of Randomization. *EC: Proceedings of the ACM Conference on Electronic Commerce*, 200–207.

Klingsberg, P. (1982). A Gray code for compositions. *Journal of Algorithms*, *3*, 41–44.

Koller, D., & Milch, B. (2003). Multi-agent influence diagrams for representing and solving games. *Games and Economic Behavior*, *45*(1), 181–221.

LaMura, P. (2000). Game networks. *UAI: Proceedings of the Conference on Uncertainty in Artificial Intelligence* (pp. 335–342).

Leyton-Brown, K., & Tennenholtz, M. (2003). Local-effect games. *IJCAI: Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 772–780).

Mangasarian, O. (1964). Equilibrium points in bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, *12*(4), 778–780.

McKelvey, R. D., McLennan, A. M., & Turocy, T. L. (2006). Gambit: Software tools for game theory. `http://econweb.tamu.edu/gambit`.

Milchtaich, I. (1996). Congestion games with player-specific payoff functions. *Games and Economic Behavior*, *13*, 111–124.

Monderer, D. (2007). Multipotential games. *IJCAI: Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 1422–1427).

Monderer, D., & Shapley, L. (1996). Potential games. *Games and Economic Behavior*, *14*, 124–143.

Nash, J. F. (1951). Non-cooperative games. *The Annals of Mathematics*, *54*(2), 286–295.

Nudelman, E., Wortman, J., Shoham, Y., & Leyton-Brown, K. (2004). Run the GAMUT: A comprehensive approach to evaluating game-theoretic algorithms. *AAMAS: Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems* (pp. 880–887).

Papadimitriou, C. (2005). Computing correlated equilibria in multiplayer games. *STOC: Proceedings of the Annual ACM Symposium on Theory of Computing* (pp. 49–56).

Papadimitriou, C., & Roughgarden, T. (2008). Computing correlated equilibria in multi-player games. *Journal of the ACM*, *55*(3), 14.

Porter, R., Nudelman, E., & Shoham, Y. (2008). Simple search methods for finding a nash equilibrium. *Games and Economic Behavior*, *63*(2), 642–662.

Rosenthal, R. (1973). A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, *2*, 65–67.

Roughgarden, T., & Tardos, É. (2002). How bad is selfish routing? *Journal of the ACM*, *49*(2), 236–259.

Russell, S., & Norvig, P. (2003). *Artificial intelligence: A modern approach, 2nd edition*. Englewood Cliffs, NJ: Prentice Hall.

Ryan, C. T., Jiang, A. X., & Leyton-Brown, K. (2010). Computing pure strategy Nash equilibria in compact symmetric games. *EC: Proceedings of the ACM Conference on Electronic Commerce* (pp. 63–72).

Sandholm, T., Gilpin, A., & Conitzer, V. (2005). Mixed-integer programming methods for finding Nash equilibria. *AAAI: Proceedings of the AAAI Conference on Artificial Intelligence* (pp. 495–501).

Scarf, H. (1967). The approximation of fixed points of a continuous mapping. *SIAM Journal of Applied Mathematics*, *15*, 1328–1343.

Shoham, Y., & Leyton-Brown, K. (2009). *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. New York: Cambridge University Press.

Stein, N. D., Parrilo, P. A., & Ozdaglar, A. (2010). *Exchangeable equilibria contradict exactness of the papadimitriou-roughgarden algorithm* (Technical Report 2852). MIT LIDS. `http://arxiv.org/abs/1010.2871v1`.

Thompson, D. R., Jiang, A. X., & Leyton-Brown, K. (2007). Game-theoretic analysis of network quality-of-service pricing. *BC.NET Conference*.

Thompson, D. R., & Leyton-Brown, K. (2009). Computational analysis of perfect-information position auctions. *EC: Proceedings of the ACM Conference on Electronic Commerce*.

Turocy, T. (2005). A dynamic homotopy interpretation of the logistic quantal response equilibrium correspondence. *Games and Economic Behavior*, *51*(2), 243–263.

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 Operations Research*, *12*(3), 377–397.

Vickrey, D., & Koller, D. (2002). Multi-agent algorithms for solving graphical games. *AAAI: Proceedings of the AAAI Conference on Artificial Intelligence* (pp. 345–351).

Yanovskaya, E. (1968). Equilibrium points in polymatrix games (in Russian). *Litovskii Matematicheskii Sbornik*, *8*, 381–384.

Zhang, N., & Poole, D. (1996). Exploiting causal independence in Bayesian network inference. *JAIR: Journal of Artificial Intelligence Research*, *5*, 301–328.