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 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 (or strict) 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. We further show that AGGs can compactly represent additional, realistic games that require exponential space under all of these existing representations. We give a dynamic programming algorithm for computing a player's expected utility under an arbitrary mixed-strategy profile, which can achieve running times polynomial in the size of an AGG representation. We show how to use this algorithm to achieve exponential speedups of existing methods for computing sample Nash and correlated equilibria. Finally, we present the results of extensive 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. 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

[*]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. 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.

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 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 those that exploit 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, 2005]. Furthermore, graphical games are also computationally well-behaved in other ways; efficient algorithms exist for computing other quantities of interest for these games such as Nash equilibria on restricted graphs [Kearns *et al.*, 2001; Elkind *et al.*, 2006] or subject to a fairness criterion [Elkind *et al.*, 2007], pure Nash equilibrium [Daskalakis & Papadimitriou, 2006], 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.[2] Recently, researchers such as Papadimitriou and Roughgarden [2005], Kalai [2005] and Daskalakis and Papadimitriou [2007] 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. The congestion game 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

[2]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.

(see, e.g., [Roughgarden & Tardos, 2002]). Second, congestion games have attractive theoretical properties. 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], 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 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 the latter. Finally, AGGs can also compactly represent many games that are compact neither as graphical games nor as congestion games.

We begin this paper in Section 2 by defining the basic AGG representation, characterizing its representation size, and showing how it can be used to represent normal-form, graphical, and symmetric games. In Section 3 we introduce the idea of *function nodes*, show how this representational device can capture additional structure in several example games, and show how to represent anonymous games as AGGs. Section 4 describes how to represent additive structure in the utility functions of AGGs, and shows how congestion and polymatrix games can be succinctly written as AGGs. Then we turn from representational to computational issues. In Section 5 we present a dynamic programming algorithm for computing an agent's expected utility under an arbitrary mixed-strategy profile, prove its correctness and complexity, and explore several elaborations. In Section 6 we prove that the problem of finding a Nash equilibrium of an AGG is PPAD-complete (a positive result, as AGGs can be exponentially smaller than normal-form games), and show how to use our dynamic programming algorithm to speed up existing methods for computing sample Nash and correlated equilibria. Finally, in Section 7 we present the results of extensive experiments with some of these algorithms, confirming our theoretical predictions and 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 involve 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, the relationship between our dynamic programming algorithm and polynomial multiplication, 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 7) 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, and in Thompson and Leyton-Brown [2008] we used AGGs to compute the Nash equilibria of perfect-information advertising auction problems. Daskalakis *et al.* [2008] (a separate group of researchers) recently considered the design of algorithms for the computation of ϵ-Nash equilibrium of AGGs.

2 Action Graph Games: The Basic Representation

We begin with an intuitive description of an action-graph game. 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.

2.1 Definition of AGGs

We now turn to a formal definition of 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 $\mathcal{A} = \bigcup_{i \in N} A_i$.*[3] *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.*

- *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.

[3] 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.

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 $\mathcal{C} : A \mapsto C$ 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 define a configuration over a subset of nodes. In particular, we will be interested in configurations over a 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.*[4] *Let $\mathcal{C}^{(\alpha)} : A \mapsto C^{(\alpha)}$ be the function which maps from an action profile to the corresponding configuration over $\nu(\alpha)$.*

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

Definition 2.4 (Action-graph game) *An action-graph game (AGG) 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 $u^\alpha : C^{(\alpha)} \mapsto \mathbb{R}$ 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}.

2.2 Example: Ice Cream Vendors

The following example helps to illustrate the elements of the AGG 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 a problem introduced by Hotelling [1929], and elaborates an example used in Leyton-Brown and Tennenholtz [2003].

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

[4] 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.

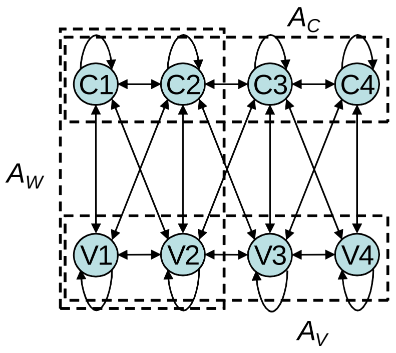


Figure 1: AGG representation of the Ice Cream Vendor game.

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 chocolate vendors have action set A_C. Note that this game exhibits context-specific independence without any strict independence, and that the graph structure is independent of n.

2.3 Size of an AGG Representation

Intuitively, AGGs (as defined so far) 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 that action graph games provide a way of representing games compactly. But what exactly is the size of an AGG representation? And how does this size grow 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 (N, A, G, u) can always be represented in space polynomial in n and $|A_i|$, the size of the utility functions can be exponential in the worst case. So the size of the utility functions determines whether an AGG 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. The following theorem gives an upper bound on the number of payoff values stored.

Theorem 2.6 *Given an AGG Γ, 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 $u^\alpha : C^{(\alpha)} \mapsto \mathbb{R}$, 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. So we do not need to explicitly specify $C^{(\alpha)}$; we can just specify a list of $|C^{(\alpha)}|$ utility values that correspond to the (ordered) set of configurations.[5] $|C^{(\alpha)}|$, the number of distinct configurations over $\nu(\alpha)$, in general does not have a closed-form expression. Instead, we consider the operation of extending all agents' action sets via $\forall i : A_i \mapsto \mathcal{A}$. This would increase the number of configurations. Thus the number of configurations over $\nu(\alpha)$ under the new action sets is an upper bound on $|C^{(\alpha)}|$. The bound 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, 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.

Theorem 2.7 *The number of payoff values stored in an AGG representation is always less than or equal to the number of payoff values in the induced normal form representation.*

> **Proof.** For each entry in the induced normal form that represents i's utility under action profile a, there exists a unique action profile a in the AGG with the corresponding action for each player. This a induces a unique configuration $\mathcal{C}(a)$ over the AGG's action nodes. By construction of the AGG utility functions, $\mathcal{C}(a)$ together with a_i determines a unique utility $u^{a_i}(\mathcal{C}^{(a_i)}(a))$ in the AGG. Furthermore, there are no entries in the AGG utility functions that do not correspond to any action profile (a_i, a_{-i}) in the normal form. This means that there exists a many-to-one mapping from entries of the normal form to utilities in the AGG. ■

Of course, the AGG representation has the extra overhead of representing the action graph, which is bounded by $|\mathcal{A}|\mathcal{I}$. But this overhead is dominated by the size of the induced normal

[5] 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 in the order of $O(\mathcal{I}|C^{(\alpha)}|)$.

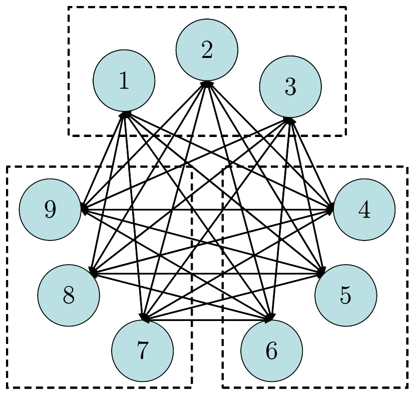


Figure 2: AGG representation of an arbitrary 3-player, 3-action game.

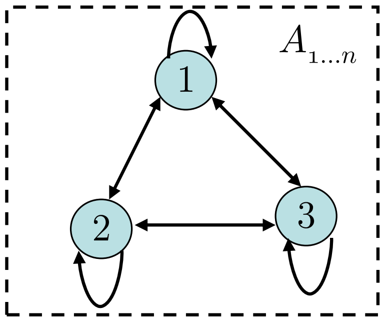


Figure 3: AGG representation of a three-action symmetric game.

form, $n \prod_j |A_j|$. Thus, an AGG's asymptotic space complexity is never worse than that of an equivalent normal form game.

It is also possible to describe a reverse transformation that encodes any arbitrary game in normal form as an AGG. 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, 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 induced normal form.

Example 2.8 (Normal-form game) *Consider an arbitrary 3-player, 3-action game encoded as an AGG (see Figure 2). Observe that there is always an edge between pairs of nodes belonging to different action sets, and that there is never an edge between nodes in the same action set.*

2.4 Representing Graphical Games as AGGs

In a graphical game [Kearns *et al.*, 2001] nodes denote agents and there is an edge connecting each agent i to each other agent whose actions can affect i's utility. Each agent then has a payoff matrix representing his local game with neighboring agents. This representation is more compact than normal form whenever the graph is not a clique. Graphical games can be represented as AGGs by replacing each node i in the graphical game by a distinct cluster of nodes 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 so that $\forall a_i \in A_i, \forall a_j \in A_j,\ a_i \in \nu(a_j)$. The resulting AGGs are as compact as the original graphical games. Furthermore, we note that by removing more edges, additional context-specific independencies can also be represented.

Example 2.9 (Graphical game) *Consider the AGG representation of a graphical game having three nodes and two edges between them (i.e., player 1 and player 3 do not directly affect each others' payoffs; see Figure 4). The AGG may appear more complex than the graphical game; however, this is only because players' actions are drawn explicitly.*

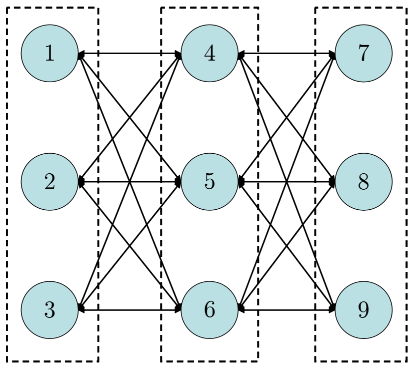


Figure 4: AGG representation of a 3-player, 3-action graphical game.

2.5 Representing Symmetric Games as AGGs

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. AGGs can capture the anonymity structure of symmetric games. An arbitrary symmetric game can be encoded as an AGG with $A_i = \mathcal{A}$ for all $i \in N$. The resulting action graph is a clique, i.e. $\nu(\alpha) = \mathcal{A}$ for all $\alpha \in \mathcal{A}$.

For each action node α, the size of its utility function u^α is proportional to the number of configurations of $n-1$ players among $|\mathcal{A}|$ actions (since at least one of the players is playing α), which is $\binom{n+|\mathcal{A}|-2}{|\mathcal{A}|-1}$. So in total the AGG representation needs to store only $|\mathcal{A}|\binom{n+|\mathcal{A}|-2}{|\mathcal{A}|-1}$ utility values. This is equal to the number of potentially distinct utility values in a symmetric game, and much less than the size of the corresponding normal form representation, $n|\mathcal{A}|^n$.

Example 2.10 (Symmetric game) *Consider the AGG representation of an n-player, three-action symmetric game (see Figure 3). The AGG has three action nodes which corresponds to the three actions of the symmetric game. Each action node has all three action nodes as neighbors.*

2.6 Example: A Job Market

Here we describe another class of example games that can be compactly represented as AGGs. Unlike the Ice Cream Vendors game, the following example does not involve choosing among actions that correspond to geographical locations.

Example 2.11 (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:*

- *a constant cost depending only on the chosen field and education level. This captures the difficulty of the studies and the cost of tuition and forgone wages.*
- *a variable reward, depending on:*
  - *the number of players who chose the same field and education level as i,*
  - *the number of players who chose a related field at the same education level,*
  - *the number of players who chose the same field at one level above or below i.*

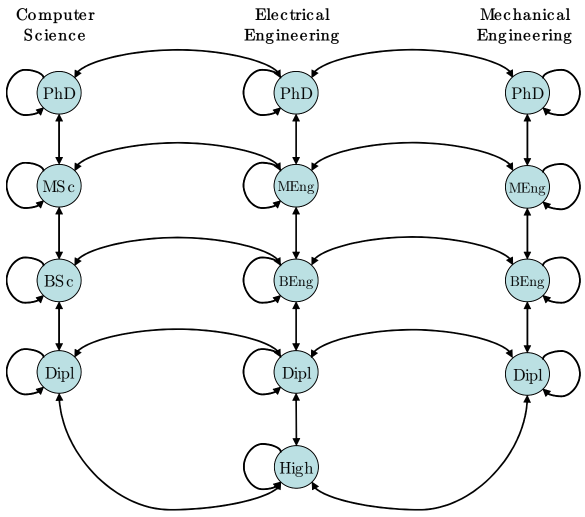


Figure 5: AGG representation of the Job Market game.

Figure 5 gives an action graph modeling one such job market scenario, in which there are three fields, Computer Science, Electrical Engineering and Mechanical Engineering. For each field there are four levels of postsecondary study: Diploma, Bachelor, Master and PhD. Computer Science and Electrical Engineering are considered related fields, and so are Electrical Engineering and Mechanical Engineering. There is another action representing high school education, which does not require a 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 2.5) would correspond to a complete action graph with in-degree 13. Thus this AGG representation is able to take advantage of the anonymity as well as the context-specific independence structure of the game's utility functions.

3 AGGs with Function Nodes

There are games with certain kinds of context-specific independence structures that AGGs, as defined in Section 2, are not able to exploit. In Example 3.1 we show a class of games with one such kind of structure. In this section we extend the AGG representation by introducing *function nodes*, allowing us to exploit a much wider variety of utility structures.

3.1 Examples: Coffee Shops and Parity

Example 3.1 (Coffee Shop game) *Consider a game involving n players; each player plans to open a new coffee shop in a downtown area, but has to decide on the location. The downtown area is represented by a $r \times k$ 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 number of players that chose the same block,*
- *the number of players that chose any of the surrounding blocks, and*

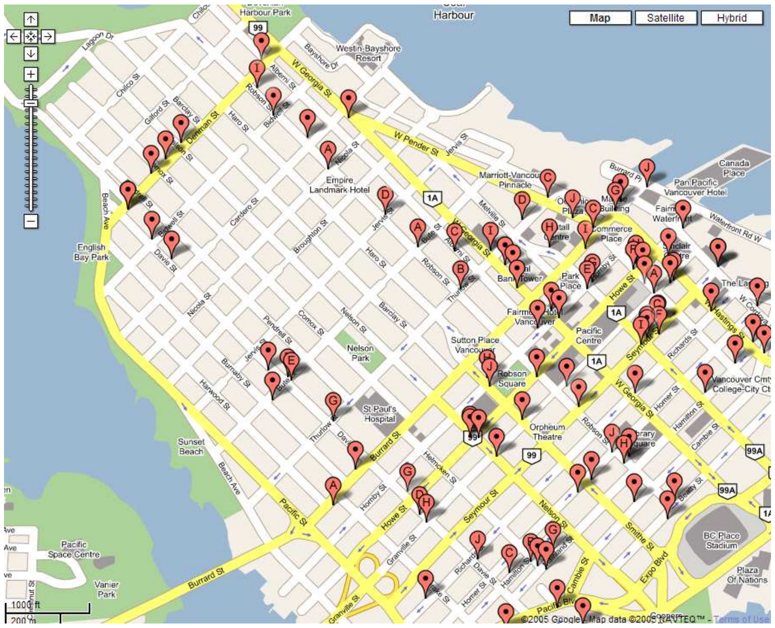


Figure 6: A Google map of coffee shops in downtown Vancouver.

- *the number of players that chose any other location.*

Figure 6 shows a Google map of coffee shops in downtown Vancouver, perhaps illustrating a Nash equilibrium of the Coffee Shop game.

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 size of the graphical game representation would be asymptotically the same. Let us now represent the game as an AGG. 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. The action graph has in-degree $\mathcal{I} = B$. Since the action sets completely overlap, the representation size is $O(|\mathcal{A}||C^{(\alpha)}|) = O\left(B\frac{(n-1+B)!}{(n-1)!B!}\right)$. If we hold B constant, this becomes $O(Bn^B)$, which is exponentially more compact than the normal form and the graphical game representation. If we instead hold n constant, the size of the representation is $O(B^n)$, which is only slightly better than the normal form and graphical game representations.

Intuitively, the AGG representation is able to exploit anonymity structure in this game. However, this game's payoff function also has context-specific structure which the AGG 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. Because the AGG representation is not able to exploit this context-specific information, it duplicates some utility values.

In the above example we showed a kind of context-specific independence structure that AGGs (as defined in Section 2) cannot exploit. 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 3.2 (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:*

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

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

3.2 Definition of AGGFNs

Structure such as that in Examples 3.1 and 3.2 can be exploited within the AGG framework by introducing *function nodes* to the action graph G. Now G's vertices consist of both the set of action nodes $\mathcal{A}$ 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. For each $p \in \mathcal{P}$, we introduce a function $f^p : C^{(p)} \mapsto \mathbb{R}$, 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. Intuitively, $c(p)$ is used to describe intermediate parameters that players' utilities depend on. 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 $u^\alpha : C^{(\alpha)} \mapsto \mathbb{R}$. We call this extended representation an Action Graph Game with Function Nodes (AGGFN), and define it formally as follows.

Definition 3.3 (AGGFN) *An Action Graph Game with Function Nodes (AGGFN) 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 $f^p : C^{(p)} \mapsto \mathbb{R}$ is an arbitrary mapping from neighbors of p to real numbers;*
- *u is a tuple $(u^\alpha)_{\alpha \in \mathcal{A}}$, where each $u^\alpha : C^{(\alpha)} \mapsto \mathbb{R}$ is the* utility function *for action α.*

3.3 Representation Size

Given an AGGFN, we can construct an equivalent AGG with the same players N and actions $\mathcal{A}$ and equivalent utility functions, but without any function nodes. We call this the *induced AGG* of the AGGFN. There is an edge from α' to α in the induced AGG either if there is an edge from α' to α in the AGGFN, or if there is a path from α' to α through a chain consisting entirely of function nodes. From the definition of AGGFNs, 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. As a result, the utility tables of the induced AGG can be filled in unambiguously.

What is the size of an AGGFN $(N, A, \mathcal{P}, G, f, u)$? How does it compare with the size of its induced AGG? We will give some formal answers presently; however, let us begin by building some intuition by considering each component of the representation.

- As discussed in Section 2.3, N and A can be represented efficiently; by a similar argument, so can $\mathcal{P}$.

- The action graph G of the AGGFN contains the extra function nodes compared to its counterpart in the induced AGG. The space complexity of the action graph becomes $O((|\mathcal{A}| + |\mathcal{P}|)^2)$, i.e., polynomial in $|\mathcal{A}|$ and $|\mathcal{P}|$.

- The number of utility values stored in an AGGFN is no greater than the number of utility values in the induced AGG. We can show this by arguments similar to those used earlier, establishing a many-to-one mapping from utilities in the AGG representation to utilities in the AGGFN. Define the *range* of f^p as $\mathcal{R}(f^p) \equiv \{f^p(c^{(p)}) : c^{(p)} \in C^{(p)}\}$. Intuitively, in order for the utility functions of the AGGFN to be significantly smaller than those of the induced AGG, there must exist some $p \in \mathcal{P}$ such that the range of f^p is a significantly smaller set than its domain $C^{(p)}$. We want f^p to map into a single value those configurations that have identical effects on the utilities of playing α; then we let p be a neighbor of α.[6]

- AGGFNs 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.

Now let us consider the representation size of AGGFNs more formally. The following theorem gives a sufficient condition for the representation size to be polynomial.

Theorem 3.4 *A class of AGGFNs 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.*

[6]Another source of compactness is the possibility that multiple actions could share the same function node as a neighbor. However, this form of structure can reduce the representation size by a factor of at most $|\mathcal{A}|$ relative to the induced AGG, so its usefulness is limited.

Proof. Given an AGGFN $(N, A, \mathcal{P}, G, f, u)$, 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 $\nu_{\mathcal{P}}(\alpha) = \nu(\alpha) \cap \mathcal{P}$ (action node neighbors and function node neighbors 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 $C^{(\alpha)} \leq (n+1)^{|\nu_{\mathcal{A}}(\alpha)|} \prod_{p \in \nu_{\mathcal{P}}(\alpha)} |\mathcal{R}(f^p)|$, which is polynomial since 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 3.5 *A class of AGGFNs 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 3.6 (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 3.5 applies. That is, the representation sizes of AGGFNs 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 Theorem 2.6 for AGGs. Intuitively, this works because both configurations on action nodes and configurations on simple aggregators count the numbers of players who behave in certain ways.

Theorem 3.7 *Consider a class of AGGFNs 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 AGGFNs' 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.

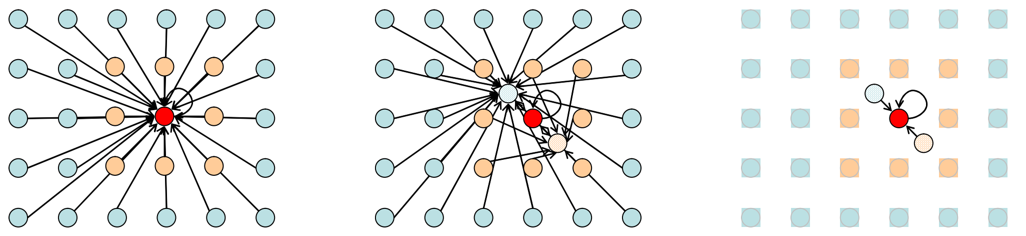


Figure 7: A 5×6 Coffee Shop game: Left: the AGG representation without function nodes (looking at only the neighborhood of α). Middle: we introduce two function nodes, p' (bottom) and p'' (top). Right: α now has only 3 neighbors.

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 Theorem 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 $\mathcal{I} = \max_{\alpha \in A} |\nu(\alpha)|$. ■

Consider the Coffee Shop game from Example 3.1. 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 7. Now each $c^{(\alpha)}$ is a configuration over only three nodes. Since each f^p is a simple aggregator, Corollary 3.5 applies and the size of this AGGFN is polynomial in n and $\mathcal{A}$. In fact since the game is symmetric and the $\beta()$'s as defined in Theorem 3.7 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 representation without function nodes, which has a representation size of $O(B\frac{(n-1+B)!}{(n-1)!B!})$.

We can represent the parity game from Example 3.2 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 to be $f^p(c^{(p)}) = \sum_{\alpha \in \nu(p)} c(\alpha) \mod 2$. Since $\mathcal{R}(f^p) = \{0, 1\}$, Corollary 3.5 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.

3.4 Representing Anonymous Games as AGGFNs

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?

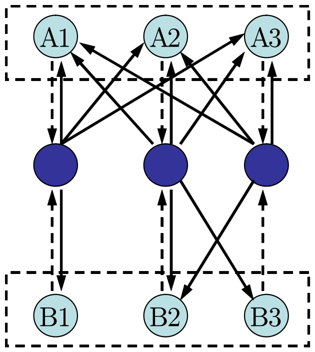


Figure 8: AGGFN representation of a game with agent-specific utility functions.

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 the summation operator: a simple aggregator. Now p gives the total number of agents who chose action α, expressing anonymity. Action nodes then include p as a neighbor instead of each α_i. This allows agents to have different utility functions without sacrificing representational compactness.

Example 3.8 (Anonymous game) *Consider the AGGFN representation of an anonymous game (see Figure 8). Consider two classes of players. Players from the first class have action set* $\{$*A1, A2, A3*$\}$*, and share utility functions that lack any independence structure. Players from the second class have action set* $\{$*B1, B2, B3*$\}$*, and share utility functions with context-specific independence structure as expressed by the absence of some of the possible edges from function nodes to action nodes.*

3.5 Example: Network Routing Game

In a network routing scenario, each player wants to transfer data from a source node to a destination node in a computer network, and needs to choose a path through the network to do so. In a simple and widely-studied model, the latency on each arc in the network is a function of the number of players who chose a path containing that arc, a player's total latency is the sum of latencies along each arc in her chosen path, and a player's utility is the negative of the latency along her chosen path. This model can be represented using congestion games (see, e.g., [Roughgarden & Tardos, 2002]). The set of facilities corresponds to the set of arcs of the network, and an action corresponds to selecting a set of arcs that form a path from source to destination. It is straightforward to see that as we have defined latencies above, congestion game utility functions can represent each player's total latencies.

However, not all network routing problems can be captured under the utility model given

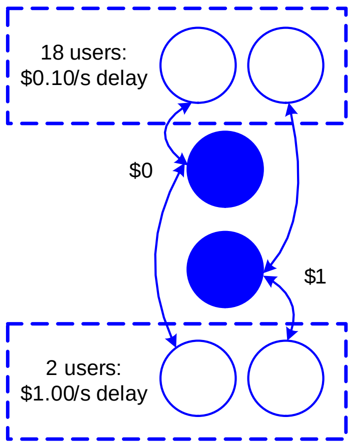


Figure 9: AGGFN representation of the network routing game.

above, and in many of these cases, the interaction ceases to be expressible as a congestion game. AGGFNs can model more general network routing scenarios, while still capturing the game's utility structure. Specifically, consider a case where players have different preferences over the network's "quality of service." This example is taken from Thompson *et al.* [2007]; it is the simplest of several problems considered in that work.

Example 3.9 (Network Routing game) *Consider a very simple network having one source node, one destination node, and two parallel arcs from the source to the destination. The two arcs are physically identical, and so have the same latency functions. However, one of the arcs has a toll of \$1 while the other arc is free. Further consider two different classes of network users. Each user of class i has a (negative) value v_i for each unit of latency experienced. A player's total utility for choosing path Π is therefore $v_i L_\Pi - T_\Pi$, where L_π is the total latency of the path, and T_Π is the toll for that path. Note that the total utility is negative. Unlike the Coffee Shop game example, the users don't have the choice of staying out of the network. So in this case maximizing utility means choosing a path with minimum cost $-v_i L_\Pi + T_\Pi$.*

This situation cannot be represented as a congestion game, while it can be efficiently represented as an AGGFN. Figure 9 shows the action graph of this game. There are two action nodes for each class of players, corresponding to the two possible paths. There are also two simple aggregator function nodes, which represent the numbers of players that chose each of the arcs.[7]

4 AGGFNs with Additive Structure

So far we have assumed that the utility functions $u^\alpha : C^{(\alpha)} \mapsto \mathbb{R}$ 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

[7]The example can be generalized to longer paths. However, in this case utility functions exhibit additivity across arcs. This means that we cannot achieve full representational compactness by using AGGFNs, but instead must use AGGFNs with additive structure, defined in Section 4.

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.

4.1 Definition of AGGFNs 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 allows compact representation because we only need to represent the summands, which have lower dimensionality than the entire function. Furthermore, due to the linearity of expectation, when we compute the expected values of such functions, we can just compute the expected values of the summands and take the weighted sum of the results.

Utility functions with additive structure appear in many domains. Previously, researchers have proposed several game representations that aim to exploit utility functions with additive structure. For example, in a *polymatrix game*, each player's utility is the sum of payoffs from separate bimatrix games between her and each of the other players. In a congestion game, each player's cost is the sum of the costs at each facility she has chosen. Each of these representations is able to exploit a specific form of additive structure, but is unable to exploit other forms of additive structure. In this section we present a unified representation of additive utility functions within our AGG framework.

We extend the AGGFN representation by allowing u^α to be represented as a weighted sum of the configuration of the neighbors of α.

Definition 4.1 *A utility function u^α of an AGGFN is* additive *if for all $m \in \nu(\alpha)$ there exist $\lambda_m \in \mathbb{R}$, such that*

$$u^\alpha(c^{(\alpha)}) \equiv \sum_{m\in\nu(\alpha)} \lambda_m c(m). \tag{4.1}$$

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 could be function nodes. Thus additive utility functions can represent weighted sums of arbitrary functions of configurations over action nodes.

We now formally define an AGGFN representation where some of the utility functions are additive.

Definition 4.2 *An* AGGFN with additive structure *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 3.3.*
- *$\mathcal{A}_+ \subseteq \mathcal{A}$ 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 3.3. These are the non-additive utility functions of the game, which are represented explicitly.*

4.2 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 Theorem 2.6 applies; otherwise the discussion in Section 3.3 applies.

4.3 Representing Congestion Games as AGGFNs with Additive Structure

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,

$$Cost_i(a_i, a_{-i}) = -u_i(a_i, a_{-i}) = \sum_{j\in a_i} K_j(\#(j, a)). \tag{4.2}$$

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 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 $O(mn + m\sum_{i\in N} |A_i|)$.

An arbitrary congestion game can be encoded as an AGGFN with no loss of compactness, where all u^α are represented as additive utility functions. Given a congestion game $(N, M, (A_i)_{i\in N}, (K_{jk})_{j\in M, k\leq n})$, we construct an AGGFN 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.

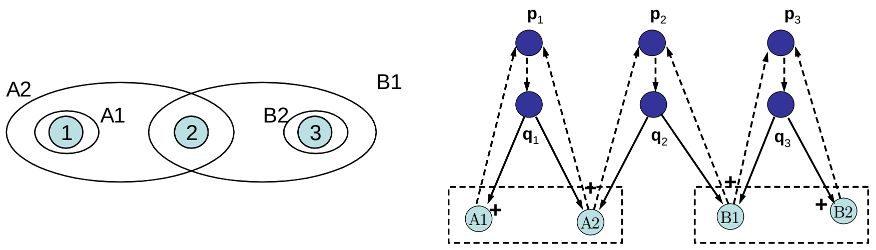


Figure 10: Left: a two-player congestion game with three facilities. The actions are shown as ovals containing their respective facilities. Right: the AGGFN representation of the same congestion game.

- For each action node α, represent the utility function u^α as an additive function with weight -1 for each of its neighbors,

$$u^\alpha(c^{(\alpha)}) = \sum_{j \in \nu(\alpha)} -c(j) = -\sum_{j \in \nu(\alpha)} K_j(\#(j,a)). \tag{4.3}$$

We can see immediately that the resulting AGGFN expresses the same utility function as the congestion game.

Example 4.3 (Congestion game) *Consider the AGGFN representation of a two-player congestion game (see Figure 10). 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 AGGFN, to substantiate our claim that it is asymptotically the same size as the congestion game. 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 AGGFN is $O(mn + m|\mathcal{A}|) = O(mn + m\sum_{i\in N}|A_i|)$. Thus this AGGFN representation has the same space complexity as the original congestion game representation.

When some of the actions correspond to the same set of facilities, it is possible to let them share the same action node in the AGGFN. The rest of the construction does not need to change; the resulting AGGFN has size $O(mn + m|\mathcal{A}|)$. Since $|\mathcal{A}| < \sum_i |A_i|$ in this case, the AGGFN can be smaller than its corresponding congestion game representation.

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

4.4 Representing Polymatrix Games as AGGFNs with Additive Structure

A polymatrix game can be compactly represented as an AGGFN with additive structure. The encoding is as follows. The AGGFN 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

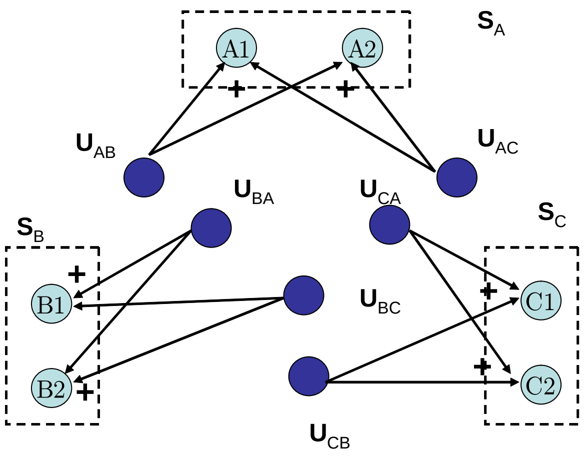


Figure 11: AGG representation of a three-player polymatrix game. Function node U_{AB} represents player A's payoffs in his bimatrix game against B, and U_{BA} represents player B's payoffs in his bimatrix game against A, and so on. To avoid clutter we do not show the edges from the action nodes to the 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}.

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 2.3, this AGGFN representation has the same space complexity as the total size of the bimatrix games.

Example 4.4 (Polymatrix game) *Consider the AGGFN representation of a three-player polymatrix game, given in Figure 11. 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.*

4.5 Representing Local Effect Games as AGGFNs with Additive Structure

Local Effect Games (LEGs), proposed by Leyton-Brown and Tennenholtz [2003], were the first action-based graphical representation of games, and hence can be seen as precursors to AGGs. However, LEGs have restricted utility functions, and hence cannot represent all games. Furthermore, despite the similarities between AGGs and LEGs, the former cannot compactly encode the latter. Here we show that LEGs *can* be compactly encoded as AGGs with additive structure.

First, we define LEGs. We begin with a graph whose nodes correspond to the actions of the game. Each player can choose any one of the nodes. Configurations are defined in the same way as in AGGs. There is a node function U_k associated with each node k, which maps the number of players choosing node k to a real number. There is an edge function $U_{k,m}$ associated with each edge (k, m) of the graph, which maps the configuration over nodes k and m to a real number. Finally, the utility of a player i choosing node k is the sum of the node function U_k and all incoming edge functions, evaluated at the current configuration c,

$$U_k(c(k)) + \sum_{m \in \nu(k)} U_{m,k}(c(m), c(k)).$$

Now we give the encoding of an LEG as an AGGFN with additive structure. The action graph of the AGGFN has the same set of nodes as the LEG. For each node k, we create a function node in the AGGFN to represent the node function U_k. For each edge (k, m) in the LEG we create a function node in the AGGFN to represent the edge function $U_{k,m}$. The neighbors of each action node k in the AGG are the function nodes corresponding to U_k and incoming edge functions $U_{m,k}$ in the LEG. Each action node's utility function is then an additive utility function. Since it takes $O(n)$ numbers to specify each node function and $O(n^2)$ numbers to specify each edge function, the sizes of an LEG and its corresponding AGGFN representation are both $O(|C|n + |E|n^2)$, where $|E|$ is the number of edges in the LEG.

4.6 Example: Congestion Games with Action-Specific Rewards

So far, we have showed that AGGFNs with additive structure can be used to bring existing game representations within 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. In Footnote 7 we briefly described an application of AGGFNs with additive structure to modeling network routing games with variable quality of service. We now give another example in more detail.

Example 4.5 (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*

$$u_i(a) = R_i(a_i) - \sum_{j \in a_i} K_j(\#(j, a)). \tag{4.4}$$

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 AGGFN. We create $\sum_i |A_i|$ action nodes, giving the agents nonoverlapping action sets. We have shown in Section 4.3 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

[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 AGGFN representation presented here.

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 $O(mn + m\sum_{i\in N}|A_i|)$.

5 Computing Expected Payoff with AGGs

So far we have concentrated on how AGGs may be used to compactly represent games of interest. But compact representation is only half the story. We now turn to the question of how to leverage this representational compactness in the computation of game-theoretic quantities of interest. 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, Govindan and Wilson's continuation methods for finding Nash equilibria [Govindan & Wilson, 2003; Govindan & Wilson, 2004], the simplicial subdivision algorithm for finding Nash equilibria [van der Laan *et al.*, 1987], and Papadimitriou's algorithm for finding correlated equilibria [Papadimitriou, 2005].

In the rest of this section, we first introduce our expected payoff algorithm for the basic AGG representation introduced in Definition 2.4. Then in Sections 5.7 and 5.8 we extend our algorithm to AGGFNs and AGGFNs with additive utility functions, respectively.

5.1 An Algorithm for Computing Expected Payoff

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

$$V^i_{a_i}(\sigma_{-i}) \equiv \sum_{a_{-i}\in A_{-i}} u_i(a_i, a_{-i}) \Pr(a_{-i}|\sigma_{-i}), \tag{5.1}$$

$$\Pr(a_{-i}|\sigma_{-i}) \equiv \prod_{j\neq i}\sigma_j(a_j). \tag{5.2}$$

Note that Equation 5.2 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 (5.1) and (5.2) are not the end of the story. However, notice that Equation (5.1) 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. Thus Equation (5.1) corresponds to an exponential-time algorithm for computing $V^i_{a_i}(\sigma_{-i})$. 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 (5.1) 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 (5.1) 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 representation. In other words, our algorithm is efficient if the AGG is compact, and requires time exponential in n if it is not. In particular, recall from Theorem 2.6 any AGG 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.

5.1.1 Exploiting Context-Specific Independence: Projection

First, we consider how to take advantage of the context-specific independence structure of the AGG, i.e., the property that i's payoff when playing a_i only depends on the configurations over the neighborhood of i. The key idea is that we can *project* the 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 is, we construct a graph from the point of view of an agent who took a particular action, expressing his sense that actions that do not affect his chosen action are in a sense the "same action." This can be thought of 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 following mapping:

$$a_j^{(\alpha)} \equiv \begin{cases} a_j & a_j \in \nu(\alpha) \\ \emptyset & a_j \notin \nu(\alpha) \end{cases} . \tag{5.3}$$

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 12, 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 following mapping:

$$\sigma_j^{(\alpha)}(a_j^{(\alpha)}) \equiv \begin{cases} \sigma_j(a_j) & a_j \in \nu(\alpha) \\ \sum_{\alpha' \in A_j \setminus \nu(\alpha)} \sigma_j(\alpha') & a_j^{(\alpha)} = \emptyset \end{cases} . \tag{5.4}$$

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

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

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

The summation is over $A_{-i}^{(a_i)}$, which in the worst case has $(|\nu(a_i)|+1)^{(n-1)}$ terms. So for AGGs with strict or context-specific independence structure, computing $V^i_{a_i}(\sigma_{-i})$ in this way is exponentially faster than doing the summation in (5.1) directly. However, the time complexity of this approach is still exponential in n.

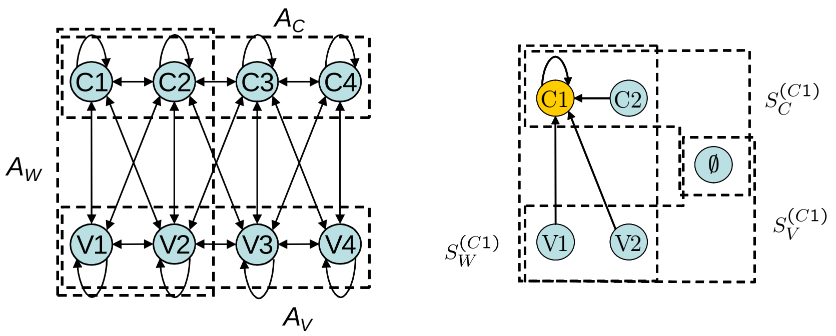


Figure 12: Projection of the action graph. Left: action graph of the Ice Cream Vendor game. Right: projected action graph and action sets with respect to the action C1.

5.1.2 Exploiting Anonymity: Summing over Configurations

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

$$V^i_{a_i}(\sigma_{-i}) = \sum_{c^{(a_i)} \in C^{(a_i,i)}} u_i\left(a_i, c^{(a_i)}\right) \Pr\left(c^{(a_i)} | \sigma^{(a_i)}\right), \tag{5.5}$$

$$\Pr\left(c^{(a_i)} | \sigma^{(a_i)}\right) = \sum_{\substack{a\,:\\ \mathcal{C}^{(a_i)}(a) = c^{(a_i)}}} \prod_{j=1}^{N} \sigma_j(a_j). \tag{5.6}$$

where $\sigma^{(a_i)} \equiv (a_i, \sigma_{-i}^{(a_i)})$ 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 (5.5) 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 (5.6) 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 (5.6) would take exponential time 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 $\sigma_{1\ldots k}^{(a_i)}$ denote the projected strategy profile of agents $\{1, \ldots, k\}$. Denote by $C_k^{(a_i)}$ the set of configurations induced by actions of agents $\{1, \ldots, k\}$. Similarly, write $c_k^{(a_i)} \in C_k^{(a_i)}$. Denote by P_k the probability distribution on $C_k^{(a_i)}$ induced by $\sigma_{1\ldots k}^{(a_i)}$, 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 $\sigma_k^{(a_i)}$. After

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

```
Algorithm ComputeP
Input: $a_i$, $\sigma^{(a_i)}$
Output: $P_n$, which is the distribution $\Pr(c^{(a_i)}|\sigma^{(a_i)})$ represented as a trie.
$c_0^{(a_i)} = (0, \ldots, 0)$
$P_0[c_0^{(a_i)}] = 1.0$ // Initialization: $C_0^{(a_i)} = \{c_0^{(a_i)}\}$
for $k = 1$ to $n$ do
    Initialize $P_k$ to be an empty trie
    for all $c_{k-1}^{(a_i)}$ from $P_{k-1}$ do
        for all $a_k^{(a_i)} \in A_k^{(a_i)}$ such that $\sigma_k^{(a_i)}(a_k^{(a_i)}) > 0$ do
            $c_k^{(a_i)} = c_{k-1}^{(a_i)}$
            if $a_k^{(a_i)} \neq \emptyset$ then
                $c_k^{(a_i)}(a_k^{(a_i)})$ += 1 // Apply action $a_k^{(a_i)}$
            end if
            if $P_k[c_k^{(a_i)}]$ does not exist yet then
                $P_k[c_k^{(a_i)}] = 0.0$
            end if
            $P_k[c_k^{(a_i)}]$ += $P_{k-1}[c_{k-1}^{(a_i)}] \times \sigma_k^{(a_i)}(a_k^{(a_i)})$
        end for
    end for
end for
return $P_n$
```

iteration n, the algorithm stops and returns P_n. The pseudocode of our DP algorithm is shown as Algorithm 1.

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

Our algorithm for computing $V^i_{a_i}(\sigma_{-i})$ is summarized in Algorithm 2.

5.2 Proof of correctness

The correctness of Algorithm 1 is not immediately obvious. It is straightforward to see that in iteration k, Algorithm 1 computes

$$\forall c_k \in C_k^{(a_i)}, \quad P_k[c_k] = \sum_{c_{k-1}, a_k^{(a_i)} : \mathcal{C}^{(a_i)}(c_{k-1}, a_k^{(a_i)}) = c_k} P_{k-1}[c_{k-1}] \times \sigma_k^{(a_i)}(a_k^{(a_i)}), \qquad (5.7)$$

where $\mathcal{C}^{(a_i)}(c_{k-1}, a_k^{(a_i)})$ denotes the configuration resulting from applying k's projected action $a_k^{(a_i)}$ to the configuration $c_{k-1} \in C_k^{(a_i)}$.

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 (5.4):

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

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 (5.5)):

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

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

$$\Pr(c_k|\sigma_{1\ldots k}) = \sum_{a_{1\ldots k}:\mathcal{C}^{(a_i)}(a_{1\ldots k})=c_k} \prod_{j=1}^{k} \sigma_j(a_j). \tag{5.8}$$

Now we prove that Algorithm 1 indeed computes the correct probability distribution, i.e., $P_k[c_k]$ as defined by Equation (5.7) is equal to $\Pr(c_k|\sigma_{1\ldots k})$.

Theorem 5.1 *For all k, and for all $c_k \in C_k^{(a_i)}$, $P_k[c_k] = \Pr(c_k|\sigma_{1\ldots k})$.*

Proof by induction on k. **Base case**: Applying Equation (5.7) for $k = 1$, it is straightforward to verify that $P_1[c_1] = \Pr(c_1|\sigma_1)$ for all $c_1 \in C_1^{(a_i)}$.

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

$$P_k[c_k] = \sum_{\substack{c_{k-1}, a_k : \\ \mathcal{C}(c_{k-1}, a_k) = c_k}} P_{k-1}[c_{k-1}] \times \sigma_k(a_k) \tag{5.9}$$

$$= \sum_{\substack{c_{k-1}, a_k : \\ \mathcal{C}(c_{k-1}, a_k) = c_k}} \sigma_k(a_k) \times \left(\sum_{a_{1\ldots k-1}:\mathcal{C}(a_{1\ldots k-1})=c_{k-1}} \prod_{j=1}^{k-1} \sigma_j(a_j) \right) \tag{5.10}$$

$$= \sum_{c_{k-1},a_k:\mathcal{C}(c_{k-1},a_k)=c_k} \left(\sum_{a_{1\ldots k-1}:\mathcal{C}(a_{1\ldots k-1})=c_{k-1}} \prod_{j=1}^{k} \sigma_j(a_j) \right) \tag{5.11}$$

$$= \sum_{a_{1\ldots k-1}} \sum_{a_k} \sum_{c_{k-1}} \mathbf{1}_{[\mathcal{C}(c_{k-1},a_k)=c_k]} \cdot \mathbf{1}_{[\mathcal{C}(a_{1\ldots k-1})=c_{k-1}]} \cdot \prod_{j=1}^{k} \sigma_j(a_j) \tag{5.12}$$

$$= \sum_{a_{1\ldots k}} \left(\sum_{c_{k-1}} \mathbf{1}_{[\mathcal{C}(c_{k-1},a_k)=c_k]} \cdot \mathbf{1}_{[\mathcal{C}(a_{1\ldots k-1})=c_{k-1}]} \right) \cdot \prod_{j=1}^{k} \sigma_j(a_j) \tag{5.13}$$

$$= \sum_{a_{1\ldots k}} \mathbf{1}_{[\mathcal{C}(a_{1\ldots k})=c_k]} \prod_{j=1}^{k} \sigma_j(a_j) \tag{5.14}$$

$$= \sum_{a_{1\ldots k}:\mathcal{C}(a_{1\ldots k})=c_k} \prod_{j=1}^{k} \sigma_j(a_j) \tag{5.15}$$

$$= \Pr(c_k|\sigma_{1\ldots k}) \tag{5.16}$$

■

Note that from Equation (5.12) to Equation (5.13) we use the fact that given an action profile $a_{1\ldots k-1}$, there is a unique configuration $c_{k-1} \in C_{k-1}^{(a_i)}$ such that $c_{k-1} = \mathcal{C}^{(a_i)}(a_{1\ldots k-1})$.

5.3 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 need to add together when doing the weighted sum in Equation (5.5). When σ_{-i} has full support, $C^{(a_i,i)}(\sigma_{-i}) = C^{(a_i,i)}$.

Theorem 5.2 *Given an AGG representation of a game, i's expected payoff $V_{a_i}^i(\sigma_{-i})$ can be computed in time polynomial in the size of the representation. If $\mathcal{I}$, the in-degree of the action graph, is bounded by a constant, $V_{a_i}^i(\sigma_{-i})$ 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 (5.5) 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 $c_{k-1}^{(a_i)}$ and $\alpha_k^{(a_i)}$, and in each case do a trie look-up which costs $O(|\nu(a_i)|)$. Since $|\mathcal{A}_k^{(a_i)}| \leq |\nu(a_i)| + 1$, and $|C_{k-1}^{(a_i)}| \leq |C^{(a_i,i)}|$, the complexity of Algorithm 1 is $O(n|\nu(a_i)|^2|C^{(a_i,i)}(\sigma_{-i})|)$. This dominates the complexity of summing up Equation (5.5). Adding the cost of computing $\sigma_{-i}^{(\alpha)}$, we get the overall complexity of expected payoff computation $O(n|\mathcal{A}| + n|\nu(a_i)|^2|C^{(a_i,i)}(\sigma_{-i})|)$.

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. 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 5.2 shows that besides exploiting the compactness of the AGG representation, our algorithm is also able to exploit the cases where the mixed strategy profiles 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 by the sizes of their projected action sets, in ascending order. This would reduce the amount of work we do in earlier iterations of Algorithm 1, but does not change the overall complexity of the algorithm.

In fact, we do not even have to apply *one* agent's strategy at a time. We could partition the set of players into subgroups, compute the distributions induced by each of these subgroups, then combine these distributions together. Algorithm 1 can be straightforwardly extended to deal with such distributions instead of mixed strategies of single agents. In Section 6.2 we apply this approach to compute Jacobians efficiently.

5.4 Relation to Polynomial Multiplication

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

$$\prod_{k=1}^{n}\left(\sigma_k^{(a_i)}(\emptyset) + \sum_{a_k \in A_k \cap \nu(a_i)} \sigma_k^{(a_i)}(a_k) x_{a_k}\right). \tag{5.17}$$

This is a multiplication of n multivariate polynomials, each corresponding to one player's projected mixed strategy $\sigma_k^{(a_i)}$. This expression expands to a polynomial of variables $(x_\alpha)_{\alpha\in\nu(a_i)}$. Each term of the polynomial can be identified by the tuple of exponents of the x_α variables. It is straightforward to verify that the set of terms exactly corresponds to the set of configurations $C^{(a_i,i)}$, where each term's tuple of exponents corresponds to a configuration $c^{(a_i)} = (c(\alpha), c(\alpha'), \ldots)$. The coefficient of the term with exponents $c \in C^{(a_i,i)}$ is

$$\sum_{a^{(a_i)}:\mathcal{C}^{(a_i)}(a^{(a_i)})=c} \left(\prod_{k=1}^{n} \sigma^{(a_i)}(a_k^{(a_i)})\right)$$

which is exactly $\Pr(c|\sigma^{(a_i)})$ by Equation (5.6)! So the whole expression in Equation (5.17) evaluates to

$$\sum_{c\in C^{(a_i,i)}} \Pr(c|\sigma^{(a_i)}) \prod_{\alpha\in\nu(a_i)} x_\alpha^{c(\alpha)}.$$

Thus the problem of computing $\Pr(c|\sigma^{(a_i)})$ is equivalent to the problem of computing the coefficients of the polynomial in Equation (5.17). Our DP algorithm corresponds to the strategy of multiplying one polynomial at a time. That is, at iteration k we multiply the polynomial corresponding to player k's strategy with the expanded polynomial of $1 \ldots (k-1)$ that we computed in the previous iteration.

5.5 Symmetric games

As described in Section 2.5, if a game is symmetric it can be represented as an AGG 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 6.2.2 we will discuss algorithms that make use of expected utility computation under symmetric strategy profiles to compute 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 AGGs under arbitrary mixed strategies, which requires time polynomial in the size of the AGG. 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 (5.5) 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 $\sigma^{(a_i)} \equiv (a_i, \sigma^{(a_i)}_{-i})$. Define $\Pr(c^{(a_i)}|\sigma^{(a_i)}_*)$ to be the distribution induced by $\sigma^{(a_i)}_{-i}$, 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 $a^{(a_i)} \in \mathcal{S}(c^{(a_i)})$

$$\Pr\left(c^{(a_i)}|\sigma^{(a_i)}_*\right) = \left|\mathcal{S}(c^{(a_i)})\right| \Pr\left(a^{(a_i)}|\sigma^{(a_i)}_*\right), \tag{5.18}$$

$$\Pr\left(a^{(a_i)}|\sigma^{(a_i)}_*\right) = \prod_{\alpha \in \mathcal{A}^{(a_i)}} \left(\sigma^{(a_i)}_*(\alpha)\right)^{c^{(a_i)}(\alpha)}. \tag{5.19}$$

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

$$\left|\mathcal{S}\left(c^{(a_i)}\right)\right| = \frac{(n-1)!}{\prod_{\alpha \in \mathcal{A}^{(a_i)}} \left(c^{(a_i)}(\alpha)\right)!}, \tag{5.20}$$

which is the multinomial coefficient.

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,

$$\Pr\left(c^{(a_i)}_{(\alpha \to \alpha')}|\sigma^{(a_i)}_*\right) = \frac{\sigma^{(a_i)}_*(\alpha') c^{(a_i)}(\alpha)}{\sigma^{(a_i)}_*(\alpha)\left(c^{(a_i)}(\alpha')+1\right)} \Pr\left(c^{(a_i)}|\sigma^{(a_i)}_*\right). \tag{5.21}$$

$H_{C^{(a_i)}}$ 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 (5.19), the results for all other projected configurations (nodes in $H_{C^{(a_i)}}$) can be computed by using Equation (5.21) 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]. To provide some intuition, 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 $\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right)$.

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

```
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 (5.18):
       $$\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right) = \frac{(n-1)!}{\prod_{\alpha\in\mathcal{A}^{(a_i)}}\left(c^{(a_i)}(\alpha)\right)!}\prod_{\alpha\in\mathcal{A}^{(a_i)}}(\sigma_*^{(a_i)}(\alpha))^{c^{(a_i)}(\alpha)}.$$
3. While there are more configurations in $C^{(a_i)}$:
   (a) get the next configuration $c_{(\alpha\to\alpha')}^{(a_i)}$ in the Hamiltonian path, using Klingsberg's algorithm [Klingsberg, 1982].
   (b) compute $\Pr\left(c_{(\alpha\to\alpha')}^{(a_i)}|\sigma_*^{(a_i)}\right)$ using Equation (5.21):
       $$\Pr\left(c_{(\alpha\to\alpha')}^{(a_i)}|\sigma_*^{(a_i)}\right) = \frac{\sigma_*^{(a_i)}(\alpha')c^{(a_i)}(\alpha)}{\sigma_*^{(a_i)}(\alpha)\left(c^{(a_i)}(\alpha')+1\right)}\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right).$$
   (c) let $c^{(a_i)} = c_{(\alpha\to\alpha')}^{(a_i)}$.
4. output $\Pr\left(c^{(a_i)}|\sigma_*^{(a_i)}\right)$ for all $c^{(a_i)} \in C^{(a_i)}$.
```

Theorem 5.3 *Computation of the expected utility $V_{a_i}^i(\sigma_*)$ under a symmetric strategy profile for symmetric action-graph games using Equations* (5.5), (5.18), (5.19) *and* (5.21) *takes time that is* $O(|\mathcal{A}| + |\nu(a_i)|\left|C^{(a_i)}(\sigma^{(a_i)})\right|)$.

Proof. Projection to $\sigma_*^{(a_i)}$ takes $O(|\mathcal{A}|)$ time since the strategies are symmetric. Equation (5.5) 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 $O(|\mathcal{A}| + |\nu(a_i)|\left|C^{(a_i)}(\sigma^{(a_i)})\right|)$. ■

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

5.6 k-symmetric Games

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

Definition 5.4 *An AGG 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 AGGs represent games having k classes of agents, where agents within each class are identical. We note that all AGGs are trivially n-symmetric. The ice cream game of Example 2.5 is an example of a nontrivial k-symmetric AGG with $k = 3$ (regardless of n).

Given a k-symmetric AGG 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 6.2.2 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.

Algorithm 4 Computing the probability distribution $\Pr(c^{(a_i)}|\sigma^{(a_i)})$ in a k-symmetric AGG 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 $\Pr(c^{(a_i)}|\sigma^{(a_i)}_{N_l})$, the probability distribution induced by $\sigma^{(a_i)}_{N_l}$, the partial strategy profile of players in N_l. Since $\sigma^{(a_i)}_{N_l}$ is symmetric, this can be computed efficiently using Algorithm 3 as discussed in Section 5.5.
3. Combine the k probability distributions together using Algorithm 1, resulting in the distribution $\Pr(c^{(a_i)}|\sigma^{(a_i)})$.

Observe that this algorithm combines our specialized Algorithm 3 for handling symmetric games from Section 5.5 with the idea of running Algorithm 1 on the joint mixed strategies of subgroups of agents discussed at the end of Section 5.3.

5.7 Computing Expected Payoff with AGGFNs

Algorithm 1 cannot be directly applied to AGGFNs with arbitrary f^p. First of all, projection of strategies does not work directly, because a player j playing an action $a_j \notin \nu(\alpha)$ 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 AGGFN to an AGG without function nodes in order to apply our algorithm, but then we would not be able to translate the extra compactness of AGGFNs over AGGs into more efficient computation.

5.7.1 Contribution-Independent Function Nodes

Luckily, the situation is better when all function nodes belong to a restricted class.

Definition 5.5 *A function node p in an AGGFN 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 $a = (a_1, \ldots, a_n)$, for all $p \in \mathcal{P}$, $c(p) = *_{i \in N: a_i \in \nu(p)} w_{a_i}$.*
- *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 AGGFN is contribution-independent if all its function nodes are contribution-independent.

Note that this definition entails 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 AGGFN. For the parity game in Example 3.2, $*$ 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, 2008]. 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 AGGFNs 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 AGGs without function nodes.

Algorithm 1 for computing the distribution $\Pr(c^{(a_i)}|\sigma)$ can be straightforwardly adopted to work with contribution-independent AGGFNs. 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 AGGFN. 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 AGGFN is $O(\sum_{\alpha \in \mathcal{A}} |C^{(\alpha)}| + \|*\|)$. As discussed in Section 3.3, this size is not necessarily polynomial in n, $|\mathcal{A}|$ and $|\mathcal{P}|$; although when the conditions in Corollary 3.5 are satisfied, the representation size is polynomial.

Theorem 5.6 *If an AGGFN is contribution-independent, then expected utility can be computed in polynomial time in the size of the AGGFN. 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 5.2, if an AGGFN is contribution-independent, expected utility $V^i_{a_i}(\sigma_{-i})$ 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 5.5, the running time for computing expected utility is polynomial in the size of the AGGFN representation. The second part of the theorem follows from a direct application of Corollary 3.5. ■

For AGGFNs 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 5.6. 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 $|\mathcal{A}|$.

5.7.2 Beyond Contribution Independence

What if not all function nodes are contribution-independent? Is there anything we can do besides converting the AGGFN into its induced AGG without function nodes? It turns out that by reducing the problem of computing expected utility to a Bayesian network inference problem, we can efficiently compute expected utilities for certain classes of non-contribution-independent AGGFNs.

Bayesian networks are used to compactly represent probability distributions by graphically describing independencies between random variables (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 between random variables. At each node X a conditional probability distribution (CPD) is defined, which specifies the probability of each realization of random variable X conditional on the realizations of neighboring random variables. Efficient algorithms have been developed to compute probabilities in Bayesian networks, such as clique tree propagation and variable elimination.

A key step in our approach for computing expected utility in AGGFNs 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 AGGFN, 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:
  - one node for each element of $\nu(a_i)$;
  - one node for each neighbor of a function node belonging to $\nu(a_i)$;
  - 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:

– 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 neighbors (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 exponential in the worst case; however, when the induced Bayesian networks have bounded tree-width, the 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.

5.8 Computing Expected Payoff with AGGFNs with Additive Structure

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

$$V^i_{a_i}(\sigma_{-i}) = \sum_{m\in\nu(a_i)} \lambda_m E[c(m)|a_i, \sigma_{-i}], \tag{5.22}$$

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 (5.22) 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

$$E[c(m)|a_i, \sigma_{-i}] = \sum_{c^{(m)}\in C^{(m)}} f^m(c^{(m)}) \Pr(c^{(m)}|a_i, \sigma_{-i}). \tag{5.23}$$

This has the same form as Equation (5.5) 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 (5.5) 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 5.7 *Suppose we are given an AGGFN in which u^α is represented as an additive utility function. If each of the neighbors of α is either*

- *an action node, or*
- *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 AGGFN representations of polymatrix games, congestion games, player-specific congestion games and the game in Example 4.5 all satisfy the conditions of Theorem 5.7.

6 Computing Equilibria with AGGs

In this section we consider some theoretical and practical applications of our expected utility algorithm. In Section 6.1 we analyze the complexity of finding a Nash equilibrium in an AGG and show that it is PPAD-complete. In Section 6.2 we extend our expected utility algorithm to the computation of payoff Jacobians, which is a key step in several algorithms for computing Nash equilibria. In Section 6.3 we show that it can also speed up the simplicial subdivision algorithm, and in Section 6.4 we show that it can be used to find correlated equilibria in polynomial time.

6.1 Complexity of Finding a Nash Equilibrium

A series of recent papers have shown that the complexity of finding a Nash equilibrium in a n-player normal-form game is PPAD-complete for $n \geq 2$ [Chen & Deng, 2006; Goldberg & Papadimitriou, 2006; Daskalakis *et al.*, 2006b]. 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 a Nash equilibrium.

Definition 6.1 (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.*

AGGs and AGGFNs have polynomial type, since the agents' action sets are represented explicitly.

Theorem 6.2 ([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 problem of finding a Nash equilibrium in this representation can be polynomially reduced to finding a Nash equilibrium in a two-player normal-form game.

Since the problem of finding a Nash equilibrium in a two-player normal-form game is PPAD-complete, the theorem implies that if the above properties hold, the problem of finding a Nash equilibrium for a compact game representation is in PPAD.

Theorem 6.3 *The complexity of finding a Nash equilibrium in an AGG is PPAD-complete.*

Remark. It may not be clear why this is interesting or encouraging. However, observe that this claim implies that the problem of finding a Nash equilibrium in an AGG 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. This is in contrast to the normal form representation of the original game, which can be exponentially larger than the AGG. 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 representation. Therefore the PPAD-membership part of this theorem is a positive result that underscores the benefits of the AGG representation.

Proof sketch We first show that the problem belongs in PPAD, by constructing a circuit that computes expected utility and satisfies the second condition of Theorem 6.2.[9] Recall that our expected utility algorithm consists of Equation (5.4), then Algorithm 1, and finally Equation (5.5). Equations (5.4) and (5.5) 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 $P_k[c_k^{(a_i)}]$ using Equation (5.7). Also there are $|A_k^{(a_i)}|$ multiplication nodes for each $c_k^{(a_i)}$, in order to carry out the multiplications in Equation (5.7).

To show PPAD-hardness, we observe that an arbitrary graphical game can be encoded as an AGG without loss of compactness (see Section 2.4. 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. Since finding a Nash equilibrium in a graphical game is known to be PPAD-hard, finding a Nash equilibrium in an AGG is PPAD-hard. ■

For AGGFNs that satisfy the conditions for Theorem 5.6 or Theorem 5.7, similar arguments apply, and we can prove PPAD-completeness for those subclasses of AGGFNs 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 6.2.

[9]Observe that the second condition in Theorem 6.2 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.

6.2 Computing Nash Equilibria: The Govindan-Wilson Algorithm

Now we move from the theoretical to the practical. We show how our dynamic programming algorithm can be used to speed up two existing algorithms for computing Nash equilibrium, and then one for computing correlated equilibria.

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 one or more equilibria 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.

How is a game's payoff Jacobian defined? The payoff Jacobian under a mixed strategy σ is a $(\sum_i |A_i|) \times (\sum_i |A_i|)$ matrix with entries defined as

$$\frac{\partial V^i_{a_i}(\sigma_{-i})}{\partial \sigma_{i'}(a_{i'})} \equiv \nabla V^{i,i'}_{a_i,a_{i'}}(\overline{\sigma}) \tag{6.1}$$

$$= \sum_{\overline{a} \in \overline{A}} u\left(a_i, \mathcal{C}(a_i, a_{i'}, \overline{a})\right) Pr(\overline{a}|\overline{\sigma}). \tag{6.2}$$

Here whenever we use an overbar in our notation, it is shorthand for the subscript $-\{i, i'\}$. For example, $\overline{a} \equiv a_{-\{i,i'\}}$. 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*.

As an aside, 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.

6.2.1 Computing the Payoff Jacobian

Now we consider how the payoff Jacobian may be computed. Equation (6.2) 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 same primary action node α, then they are expected payoffs on the same utility function u^α. In other words, they have the same value 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,[10] 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 $\delta^{(act)}_{a'_i} = \delta^{(act)}_{a'_j}$ for contribution-independent AGGFNs), then $\nabla V^{i,j}_{a_i,a'_j} = \nabla V^{j,i}_{a_j,a'_i}$.

A consequence of 1(a) above is that any Jacobian of an AGG has at most $\sum_i \sum_{a_i \in A_i}(n-1)(\nu(a_i)+1)$ unique 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. One way to implement this idea is to use a cache system that stores the common value of each set of identical entries.

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, and re-use intermediate results when computing the induced distributions of different entries. Since computing the induced probability distributions is the bottleneck of our expected payoff algorithm, this provides significant speedup.

First we observe that if we fix the row (i, 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 AGGs without function nodes, 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 $\sigma_{-\{i,j\}}$. Recall from Section 5.4 that the distributions are coefficients of the multiplication of certain polynomials. So we can undo j's strategy by computing the long division of the polynomial for σ_{-i} by the polynomial for σ_j.

This method does not work for contribution-independent AGGFNs, because in general a player's contribution to the configurations are not reversible, i.e., given $\Pr(c_{n-1}|\sigma_{-i}^{(a_i)})$ and σ_j, it is not always possible to undo the contributions of σ_j. Instead, we can efficiently compute the distributions by recursively bisecting the set of players into subgroups, computing probability distributions induced by the strategies of these subgroups, and combining them. For example, suppose $n = 9$ and $i = 9$, so $\sigma_{-i} = \sigma_{1\ldots8}$. We need to compute the distributions induced by $\sigma_{-\{1,9\}}, \ldots, \sigma_{-\{8,9\}}$, respectively. Now we bisect σ_{-i} into $\sigma_{1\ldots4}$ and $\sigma_{5\ldots8}$. Suppose we have computed the distributions induced by $\sigma_{1\ldots4}$ as well as $\sigma_{234}, \sigma_{134}, \sigma_{124}, \sigma_{123}$, and similarly for

[10]For contribution-independent AGGFNs, 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.

the other group of $5 \dots 8$. Then we can compute $\Pr(\cdot|\sigma_{-\{1,9\}}^{(a_i)})$ by combining $\Pr(\cdot|\sigma_{234}^{(a_i)})$ and $\Pr(\cdot|\sigma_{5678}^{(a_i)})$, compute $\Pr(\cdot|\sigma_{-\{2,9\}}^{(a_i)})$ by combining $\Pr(\cdot|\sigma_{134}^{(a_i)})$ and $\Pr(\cdot|\sigma_{5678}^{(a_i)})$, etc. We have reduced the problem into two smaller problems over the subgroups $1 \dots 4$ and $5 \dots 8$, which can then be solved recursively by further bisecting the subgroups. This method saves the recomputation of subgroups of strategies when computing the induced distributions for each row of the Jacobian, and it works with any contribution-independent AGGFNs because it does not use long division to undo strategies.

6.2.2 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 also be adapted to find symmetric Nash equilibria in symmetric AGGs. In order to compute a symmetric equilibrium, the algorithm must be seeded with a symmetric equilibrium of the perturbed game. This is accomplished by giving all agents the same large bonus to some common action, so that in the only equilibrium of the initial perturbed game all agents take the same action. Then the algorithm follows a path of symmetric equilibria of perturbed games to a symmetric equilibrium of the unperturbed game. Thus each call to compute a payoff Jacobian would reference a symmetric strategy profile, and so the expected utility computations can be performed using the techniques discussed in Section 5.5, 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. Furthermore, the symmetry of the game and the strategy profile ensures 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 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 AGGs. The bottleneck is the computation of payoff Jacobians under k-symmetric strategy profiles, which can be efficiently performed using the techniques discussed in Section 5.6.

6.3 Computing Nash Equilibria: The Simplicial Subdivision Algorithm

Another algorithm for computing Nash equilibria 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, but using a finer grid.

The algorithm's bottleneck step is computation of labels of the subsimplexes along the path, 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 compared to a normal-form-based implementation.

6.4 Computing Correlated Equilibria: Papadimitriou's Algorithm

Papadimitriou [2005] proposed a very general, polynomial-time algorithm for computing correlated equilibria.

Theorem 6.4 ([Papadimitriou, 2005]) *If a game representation has polynomial type, and has a polynomial algorithm for computing expected utility, then a correlated equilibrium can be computed in polynomial time.*

The reader might wonder why this is interesting, since 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 5.1 is inadequate. Indeed, in these cases even *describing* a correlated equilibrium using these probabilities of action profiles can require exponential space. Papadimitriou's result is thus much deeper than it may first seem. Its proof begins by showing that every compactly represented game has a correlated equilibrium that can be written as the mixture of a polynomial number of product distributions. 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 Papadimitriou's theorem involves the computation of expected utility, which is a bottleneck step in his algorithm. As a direct corollary of Theorem 6.4 and our Theorem 5.2 which states that there is indeed a polynomial algorithm that computes expected payoffs for AGGs, we have a polynomial algorithm for computing a correlated equilibrium given an AGG.

Corollary 6.5 *Given a game represented as an AGG, a correlated equilibrium can be computed in polynomial time.*

Similarly, for the subclasses of AGGFNs for which the expected utility problem can be solved in polynomial time (see Theorems 5.6 and 5.7), correlated equilibria can be computed in polynomial time.

7 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 somehow be less useful than they appear in practice. 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 that we have performed with AGGs. The software tools that we have developed will make it easy for other researchers to use AGGs to model problems of interest.

In the rest of this section, we show the results of experiments comparing 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 AGGs 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, in Section 7.7 we show how sample equilibria of these games can be visualized on the action graphs.

7.1 Software Implementation and Experimental Setup

We implemented our algorithms as a software package written in C++. Our software is capable of the following:

- 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 3.1), Job Market games (Example 2.11) and symmetric AGGs 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 for download 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 7.7 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 7.7 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 flavor 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 flavor, and a weaker positive effect from players choosing a different flavor. 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.

7.2 Representation Size

First, we compared the representation sizes of AGGFNs to those of their induced normal form representations. For each game instance we counted the number of payoff values that needed to be stored in each representation.

We first looked at Coffee Shop games with 5×5 blocks, with varying number of players. Figure 13 (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 for large number of players. The size of the AGG representation grew polynomially with respect to n. As we can see from Figure 13 (right), even for a game instance with 80 players, the AGGFN representation stored only about 2 million numbers. In contrast, the 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 blocks. For ease of comparison we fixed the number of columns at 5 and only changed the number of rows. Recall from Section 3.1 that for both AGG and normal form representations of Coffee Shop games, the representation sizes depend only on the number of players and number of actions, but not on the shape of the region. (Recall that the number of actions equals to $B + 1$ where B is the total number of blocks.) Figure 13 (left) shows a log-scale plot of the number of payoff values versus the number of actions, and Figure 13 (right) gives a plot for just the AGGFN representation as we increased the number of rows to 80. The size of the AGG representation grew linearly with the number of rows, whereas the size of the normal form representation grew like a higher-order polynomial. This was consistent with our theoretical prediction that AGGFNs store $O(|\mathcal{A}|n^3)$ payoff values for Coffee Shop games while normal form representations store $n|\mathcal{A}|^n$ payoff values. For a Coffee Shop game with 4 players on an 80×5 grid, the AGGFN representation needs to store 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 2.11, which have 13 actions. We varied the number of players from 3 to 24. The results are similar, as shown in Figure 15 (left). This is consistent with our theoretical prediction that the sizes of normal form representations grow exponentially in n while the sizes of AGG representations grow polynomially in n.

7.3 Expected Utility Computation

We tested the performance of our dynamic programming algorithm for computing expected utilities in AGGFNs 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 14 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 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 has no trouble with large 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,

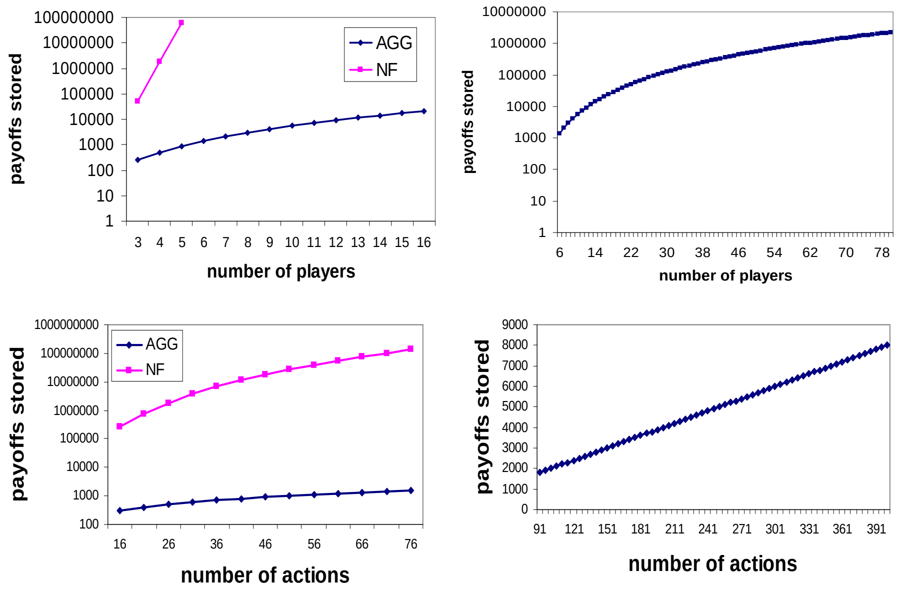


Figure 13: Comparing Representation Sizes of the Coffee Shop game. Top left: 5×5 grid with 3 to 16 players (log-scale). Top right: AGG only, 5×5 grid with up to 80 players (log-scale). Bottom left: 4-player $r \times 5$ grid with r varying from 3 to 15 (log-scale). Bottom right: AGG only, up to 80 rows.

while the normal-form-based algorithm grew like a higher-order polynomial. This was consistent with our theoretical prediction 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 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 15 (right). The normal-form-based implementation runs 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 Job Market game.

7.4 Computing Payoff Jacobians

We ran similar experiments to investigate the computation of payoff Jacobians. As discussed in Section 6.2, 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 6.2 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 6.2 to GameTracer's implementation, the results are very similar to the above results for computing expected payoffs: our implementation scales polyno-

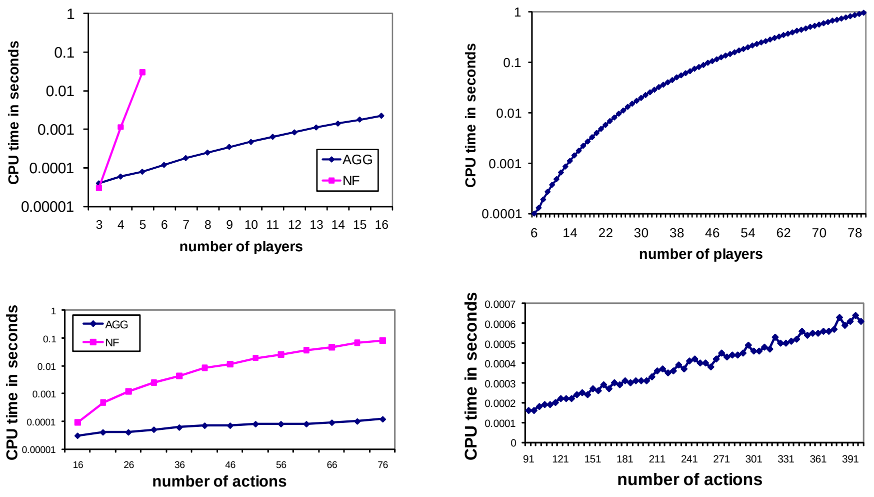


Figure 14: Running times for payoff computation in the Coffee Shop game. Top left: 5×5 grid with 3 to 16 players. Top right: AGG only, 5×5 grid with up to 80 players. Bottom left: 4-player $r \times 5$ grid with r varying from 3 to 15. Bottom right: AGG only, up to 80 rows.

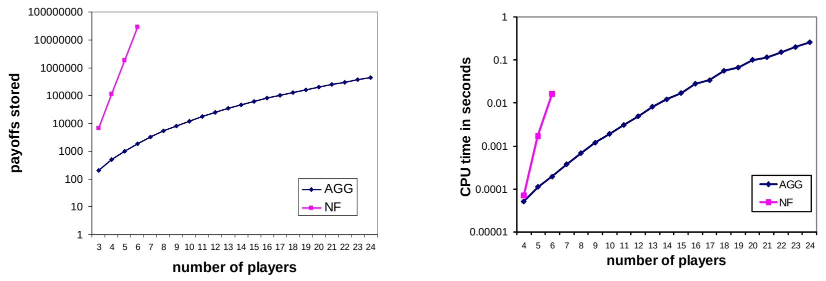


Figure 15: Results for Job Market games with varying numbers of players. Left: comparing representation sizes. Right: running times for computing 1000 expected utilities.

mially in n while GameTracer scales exponentially in n. We instead focus on the question of how much speedup the methods in Section 6.2 provide, by comparing our algorithm in Section 6.2 against the algorithm that computes expected payoffs (using our AGG-based algorithm described in Section 5) 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 6.2 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 6.2 provide significant speedup for computing payoff Jacobians.

7.5 Finding Nash Equilibria using the Govindan-Wilson algorithm

Now we show experimentally that the speedup we achieved for computing Jacobians using the AGG representation leads 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. Any difference in the two algorithms' running times is therefore due to their different methods of computing Jacobians.

Again, we tested the two algorithms on Coffee Shop games of varying sizes: first we fixed the size of blocks at 4×4 and varied the number of players; then we fixed the number of players at 4 and number of columns at 4 and varied the number of rows. For each game instance, we randomly generated 10 initial perturbation vectors, and for each initial perturbation we run the two versions of the Govindan-Wilson algorithm. Since the running time of the Govindan-Wilson algorithm highly depends on the initial perturbation, for each game instance the running times with different initial perturbations had large variance. Instead, for each initial perturbation we looked at the *ratio* of running times between the normal-form implementation and the AGG implementation. Thus a ratio greater than 1 means the AGG implementation spent less time than the normal form implementation. We plotted the results in Figure 16 (left). The results confirmed our theoretical prediction that as the size of the games grows (either in the number of players or in the number of actions), the speedup of the AGG implementation compared to the normal-form implementation increases. The normal-form implementation runs 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 16 (top-right), for game instances with up to 12 players, the algorithm terminated within one day for most of the initial perturbations. A normal form representation of such a game would have needed to store 7.0×10^{15} numbers. Figure 16 (bottom-right) shows a boxplot of the CPU times for the AGG-based implementation, as we vary 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 17. For the game instance with 6 players, the AGG-based implementation is about 100 times faster than the normal-form-based implementation. While the normal-form-based implementation runs out of memory for Job Market games with more than 6 players, the AGG-based implementation was able to solve the game with 16 players in 24 minutes on average. A normal form representation of such a game would have needed to store 1.1×10^{19} utility values.

7.6 Finding Nash Equilibria using Simplicial Subdivision

As discussed in Section 6.3, 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, given a 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 ran experiments that compared the performance of AGG-based simplicial subdivision

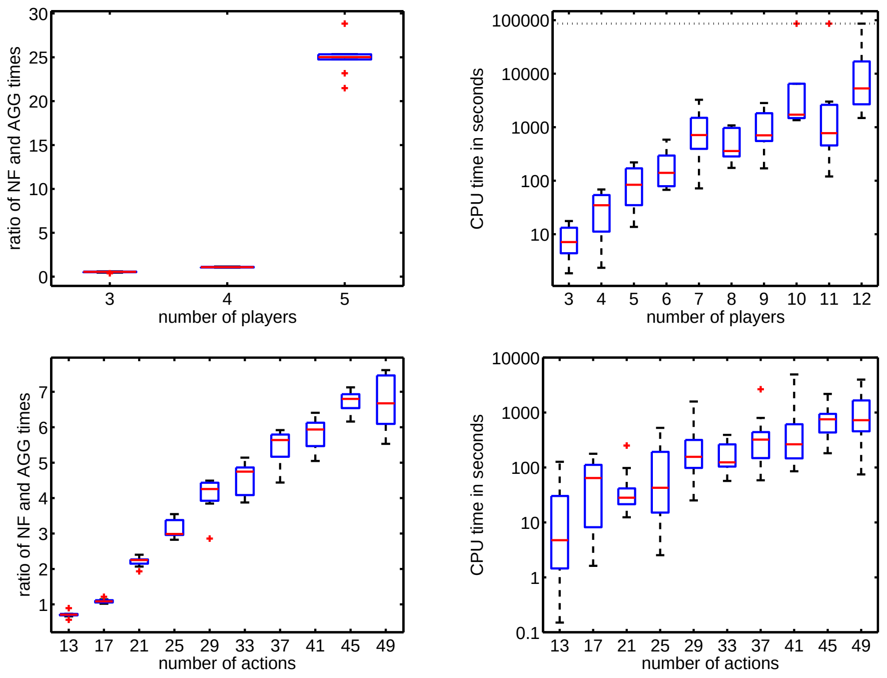
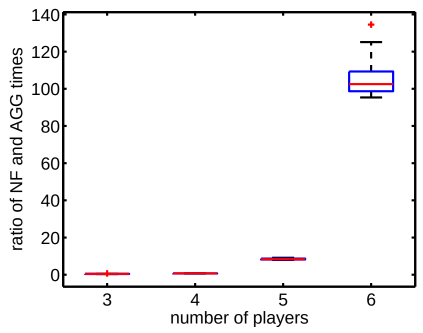


Figure 16: Govindan-Wilson algorithms in the Coffee Shop game. Top row: 4×4 grid with varying number of players. Bottom row: 4-player $r \times 4$ grid with r varying from 3 to 12. For each row, the left figure shows ratio of running times; the right figure shows logscale plot of CPU times for the AGG-based implementation (the black horizontal line indicates the one-day cutoff time).

against normal-form-based simplicial subdivision on instances of Coffee Shop games as well as instances of randomly-generated symmetric AGGs on small world graphs. For each game instance, we ran both implementations with the starting mixed strategy profile in which each player gives equal probability to each of her actions.

We first tested on 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 18 (left) has a boxplot of the ratio of running times between the two implementations. The AGG-based implementation is about 3 times faster for the 3-player instances and about 30 times faster in the 4-player instances. We also tested on Coffee Shop games with 3 players, 3 columns and varying numbers of rows from 4 to 7. For each game size we generated 10 instances with random payoffs. Figure 18 (right) has a boxplot of the ratio of running times. As expected, the AGG-based implementation is faster and the gap in performance widens as the games become larger.

We then tested on symmetric AGGs 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 19 (top row). While the actual running times on different instances show large variance, the ratios of running times between normal-form-based and AGG-based implementations show a clear in-



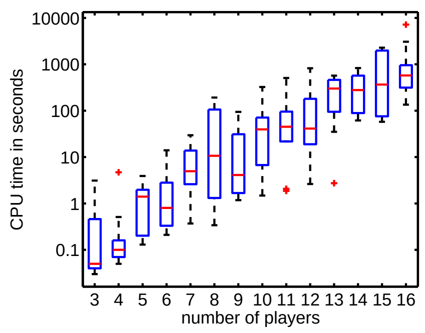
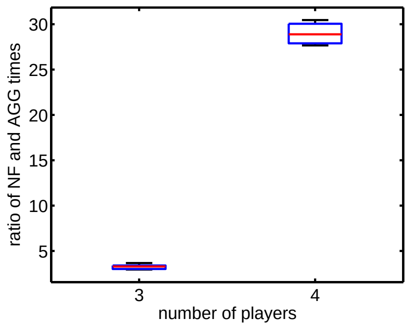


Figure 17: Govindan-Wilson algorithms on Job Market games with varying numbers of players. Left: ratios of running times. Right: logscale plot of CPU times for the AGG-based implementation.



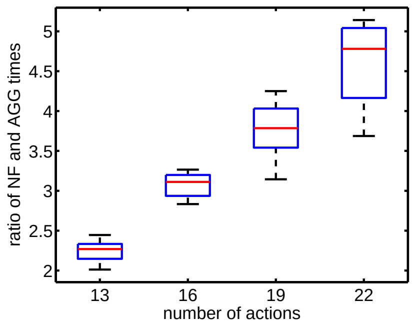


Figure 18: Ratios of running times of simplicial subdivision algorithms on Coffee Shop games. Left: 4×4 grid with 3 to 4 players. Right: 3-player $r \times 3$ grid with r varying from 4 to 7.

creasing trend as the number of players increases. The normal-form-based implementation ran out of memory for instances with greater than 5 players. Meanwhile, we ran the AGG-based implementation on larger instances with a one-day cutoff time. As shown by the log-scale boxplot of CPU times, the AGG-based implementation was able to solve a majority of instances with up to 8 players within one day for each instance.

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 19 (bottom row). Again, while the actual running times on different instances show large variance, the ratios of running times show a clear increasing trend as the number of actions increases. The AGG-based implementation was able to solve a 16-action instance in about 3 minutes on average, while the normal-form-based implementation took about 2 hours on average.

7.7 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 of interest (such as a Nash equilibrium of the game) can be visualized on the action graph by displaying the expected numbers of players that choose each of the actions under the strategy profile σ. We call such a tuple the *expected*

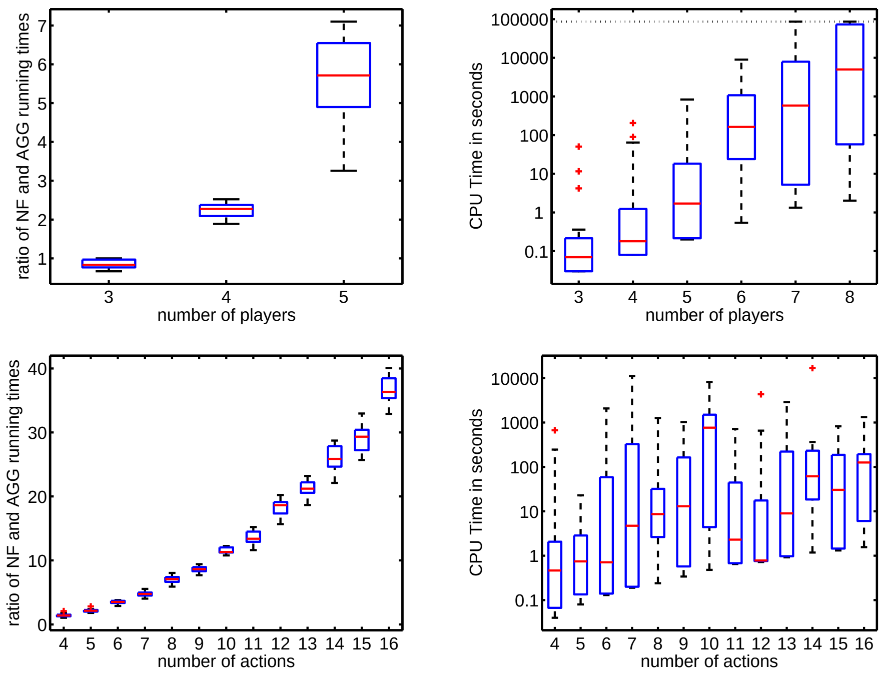


Figure 19: Simplicial subdivision algorithms on symmetric AGGs on Small World Graphs. Top row: 5 actions, varying numbers of players. Bottom row: 4 players, varying numbers of actions. For each row, the left figure shows ratio of running times; the right figure shows logscale plot of CPU times for the AGG-based implementation (the black horizontal line indicates the one-day cutoff time).

configuration under σ. This can be easily computed given σ: for each action node α, just add up 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 exactly 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 of players under the strategy profile. To illustrate, we generated a 16-player Coffee Shop game on a 4×4 grid. We ran the Govindan-Wilson algorithm with AGG-based implementation for the Jacobian computation, which found a Nash equilibrium in 77 seconds of CPU time. The expected configuration of the (pure strategy) equilibrium is visualized in Figure 20.

We also tested on 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 Govindan-Wilson algorithm with AGG-based implementation for the Jacobian computation, which found a Nash equilibrium in 860 seconds of CPU time. The expected configuration of the equilibrium is visualized in Figure 21 (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 isolated two mixed equilibrium strategies and show how they can be visualized in Figure 21 (right).

We also tested on a Ice Cream Vendor game (Example 2.5) with 4 locations, 6 chocolate

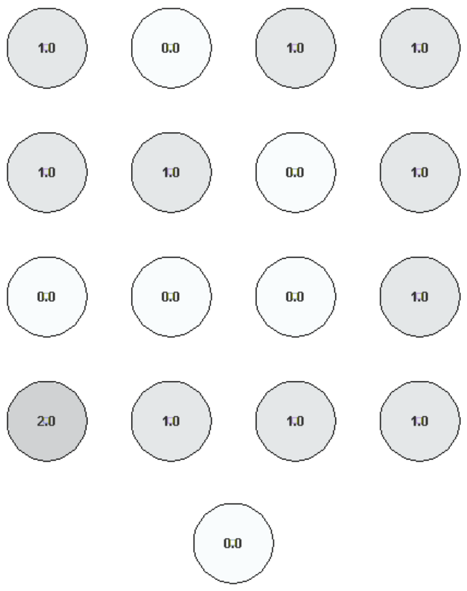


Figure 20: Visualization of a Nash equilibrium of a 16-player Coffee Shop game on a 4×4 grid. The function nodes and the edges of the action graph are not shown. The action node at the bottom corresponds to not entering the market.

vendors, 6 vanilla vendors, and 4 west-side vendors. The Govindan-Wilson algorithm found one equilibrium in 9 seconds of CPU time. The expected configuration of the (pure strategy) equilibrium is visualized in Figure 22. Observe that the west side is relatively denser, due to the presence of the west-side vendors. Furthermore, the locations at the east and west ends are chosen relatively more often than the middle locations, reflecting the fact that the ends have relatively fewer neighbors and thus less competition.

8 Conclusions

We proposed action-graph games (AGGs) as 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 many previously-studied compact game classes including graphical games, symmetric games, anonymous games and congestion games.

We presented a polynomial-time algorithm for computing expected utilities in AGGs and contribution-independent AGGFNs. For symmetric and k-symmetric AGGs, 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 sample Nash and correlated equilibria. We showed experimentally that using AGGs allows us to model and analyze dramatically larger games than can be addressed with the normal-form representation.

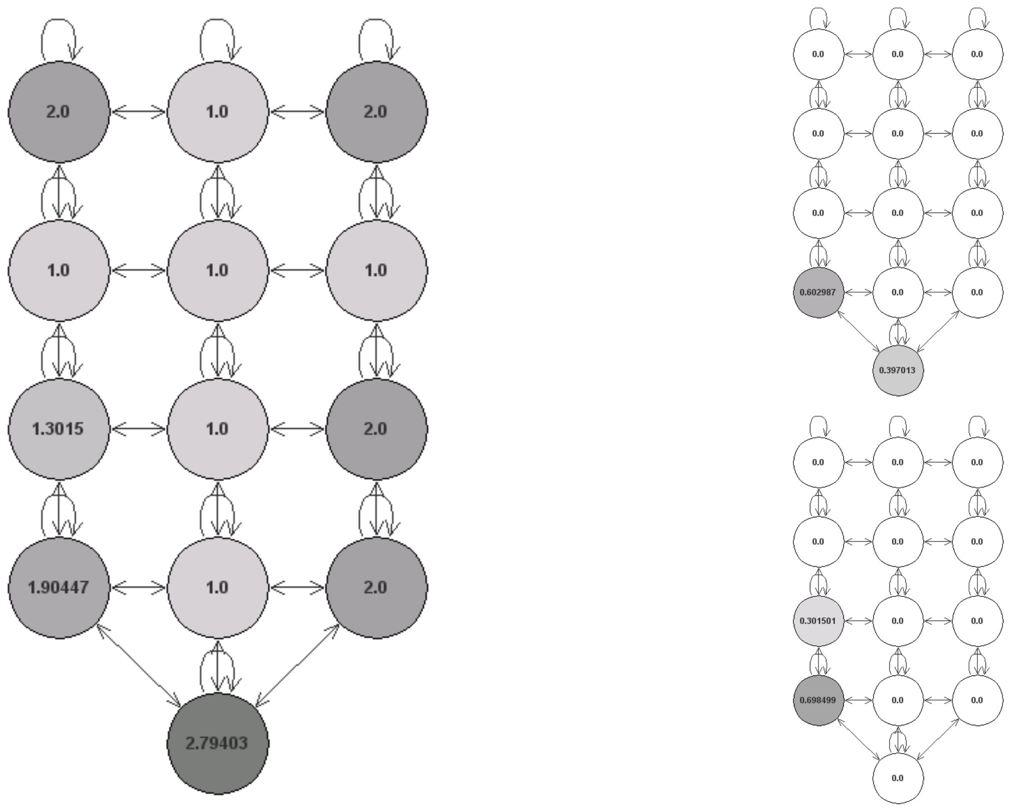


Figure 21: Visualization of a Nash equilibrium of a Job Market game with 20 players. Left: expected configuration of the equilibrium. Right: two mixed equilibrium strategies.

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 multi-agent systems, and have preliminary results for network routing problems [Thompson *et al.*, 2007] and complete-information advertising auction problems [Thompson & Leyton-Brown, 2008]. Another interesting problem is the computation of pure-strategy Nash equilibria in AGGs. While the problem is NP-complete in general, in [Jiang & Leyton-Brown, 2007] we presented a polynomial time algorithm for the class of symmetric AGGs whose action graphs have bounded in-degree and bounded tree-width. We are currently extending this algorithm to classes of asymmetric AGGs. We are also working on extending our AGG framework to represent games of incomplete information (Bayesian games) as well as dynamic games.

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).

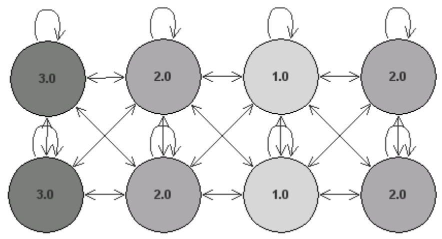


Figure 22: Visualization of a Nash equilibrium of an Ice Cream Vendor game.

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

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.

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).

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. (2008). On the complexity of Nash equilibria of Action-Graph Games. *submitted*.

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.

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).

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., & 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).

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).

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. H., & Roughgarden, T. (2005). Computing equilibria in multi-player games. *SODA: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (pp. 82–91).

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.

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

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. (2008). Tractable computational methods for finding Nash equilibria of perfect-information position auctions. *Workshop on Ad Auctions at the ACM Conference on Electronic Commerce*.

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).

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