The Positronic Economist: A Computational System for Analyzing Economic Mechanisms

David Thompson, Neil Newman, Kevin Leyton-Brown
Department of Computer Science
University of British Columbia, Canada
{daveth, newmanne, kevinlb}@cs.ubc.ca

Abstract

Computational mechanism analysis is a recent approach to economic analysis in which a mechanism design setting is analyzed entirely by a computer. For games with non-trivial numbers of players and actions, the approach is only feasible when these games can be encoded compactly, e.g., as Action-Graph Games. Such encoding is currently a manual process requiring expert knowledge; our aim is to simplify and automate it. Our contribution, the *Positronic Economist* is a software system having two parts: (1) a Python-based language for succinctly describing mechanisms; and (2) a system that takes such descriptions as input, automatically identifies computationally useful structure, and produces a compact Action-Graph Game.

1 Introduction

A mechanism is a protocol for collective decision making among self-interested agents. Mechanisms model many social processes from auctions to elections. They are widely studied in computer science, both because the participants in real-world mechanisms can be autonomous software systems (e.g. algorithmic bidding and trading agents) and because algorithms such as job schedulers give rise to mechanisms when users have competing interests. Mechanisms are complicated to understand because participants respond to rule changes strategically. Thus, although real-world mechanisms are often fairly simple (e.g., plurality voting), a mechanism's outcome depends not just on its functional description but on agents' strategic choices. Game theory provides principled methods for analyzing such choices. Unfortunately, game theoretic analysis is a difficult process requiring either substantial human effort or very large amounts of computation.

An alternative is offered by recent advances in algorithms for efficiently computing game-theoretic solution concepts (Roughgarden and Papadimitriou 2008; Jiang, Leyton-Brown, and Bhat 2011a), and on using such algorithms to analyze mechanisms (Thompson and Leyton-Brown 2009), an approach referred to as *computational mechanism analysis* (CMA). While these algorithms can operate on normal-form descriptions of games, applying them to computationally motivated *compact game representations* allows for exponential speedups (Kearns, Littman, and Singh 2001; Jiang, Leyton-Brown, and Bhat 2011a). For example, Action-Graph Games (AGGs) can compute an agent's expected utility under an arbitrary mixed strategy profile with a polynomial-time dynamic programming algorithm, and this expected utility problem constitutes the inner loop of many game-theoretic algorithms. (The standard method of computing expected utility is polynomial in the size of the normal form; since an AGG can be exponentially smaller, the AGG representation can yield exponential speedups.)

We have used this approach of combining efficient algorithms with compact representations to address open problems in mechanism analysis ranging from bidding in advertising auctions (Thompson and Leyton-Brown 2009; 2013) to strategic voting in plurality elections (Thompson et al. 2013). The approach has the advantage that games need not be approximated and that equilibria are computed exactly; we leverage compact representations and sophisticated, exact-equilibrium-finding algorithms to manage computational demands. However, we have found it nontrivial to produce the application-specific encodings into compact game representations upon which this process depends.

This paper's goal is to make CMA accessible to a broader audience without reducing its fidelity. We summarize our desiderata as the "Three Laws of Positronic Economics," inspired by Asimov's [1942] "Three Laws of Robotics."

1. **Precision:** Games are represented exactly. Equilibria are either exact or are ϵ-equilibria with ϵ roughly machine-ϵ.
2. **Speed:** Algorithms must be fast enough to be used in practice, except when this is in conflict with the first law.
3. **Autonomy:** Human effort should be minimized, except when this conflicts with the first two laws.

In this paper, we introduce a tool, *Positronic Economist* (PosEc), that delivers on these goals, dramatically reducing the human effort required by CMA without compromising the analysis itself. Our paper makes two major contributions. The first is a high-level Python-based declarative language for describing mechanisms in a form that closely resembles their natural mathematical representations; the second is a pair of complementary algorithms that automatically infer the structure of a game specified in this language and produce a compact Bayesian AGG (BAGG). We then draw on a portfolio of existing algorithms to solve the game. We chose to use BAGGs because of the availability of empirically fast tools available for working with them: while Nash equilibrium finding is an NP- or PPAD-hard problem for BAGGs, depending on the equilibrium type, good heuristics can often find exact equilibria quickly (Thompson, Leung, and Leyton-Brown 2011; Jiang, Leyton-Brown, and Bhat 2011a).

The structure of the remainder of this paper is as follows. Section 2 surveys related work. Section 3 formalizes the mechanisms and settings that PosEc can represent. Section 4 briefly describes BAGGs. Section 5 provides an overview of the PosEc representation language. Section 6 details PosEc's two structure inference algorithms. Section 7 discusses experiments that help characterize PosEc's performance. Finally, Section 8 discusses some directions for future work.

2 Related Work

We are not aware of any other work with the goal of simplifying the process of generating compact representations. Perhaps the closest is by Duong et al. (2009), which also integrates with the compact games literature. They provide algorithms to construct a graphical game that best approximates an input game from a set of samples of strategies and payoffs, which either come from previously observed play or a simulator. The question of how to obtain samples from a simulator without unacceptably degrading fidelity is unanswered. We note that PosEc does not require any samples.

We are aware of two other CMA systems, which differ from PosEc in the human effort required to specify games, in the types of games that they are able to specify, and in their ability to leverage high performance algorithms. The system of Rabinovich et al. (2013) does not explicitly specify how games are represented. A user must provide code for computing expected utility given a strategy profile, which could require substantial human effort; observe that one of the main benefits of AGGs is that they provide such an efficient computational procedure. Their system is restricted to two player games of incomplete information, places restrictions on the form of utility functions, and requires strategy and type sets to be continuous and unbounded. While PosEc cannot handle infinite strategy or type sets due to the discrete nature of BAGGs, it can represent games with any number of players and does not impose restrictions on the form of the utility function. Their system only supports the fictitious play algorithm (FP), which is a relatively weak Nash equilibrium computation algorithm, prone to getting stuck in cycles.

Another CMA system was introduced by Vorobeychik, Reeves, and Wellman (2012). This system describes mechanisms and settings as piecewise linear equations. Given that many single-parameter mechanisms and settings are described algebraically in the literature, this representation requires very little human effort. However, the only supported equilibrium-finding algorithm is iterative best response (IBR), which is unable to compute mixed-strategy equilibria and can also fail to converge. In contrast, PosEc makes it easy for the user to leverage a wide range of high-performance solvers.

There is other work in the empirical game-theoretic analysis (EGTA) space that proposes specialized algorithms that operate directly on a simulator, without translation to an intermediate form. One example is by Vorobeychik and Wellman (2008), who used simulated annealing to find a strategy profile constituting an approximate Nash equilibrium of the simulated game. While the human effort of building a simulator is comparable the effort of specifying a game in PosEc, the major difference is that PosEc creates a compact game that represents the input game exactly and computes exact equilibria of the input game. While approximate equilibria are a major subject of research in algorithms and complexity, researchers in mechanism design and auction theory overwhelmingly favor exact equilibria.

3 Mechanisms and Settings

We now formally describe the games that PosEc is able to represent. An *epistemic-type Bayesian game* is specified by $\langle N, A, \Theta, p, \mathcal{U}\rangle$, where N is a set of agents, numbered 1 to n, $A = A_1 \times \cdots \times A_n$, where A_i is a set of actions that agent i can perform, Θ is the set of private types that an agent can have, p is the joint type distribution, $p \in \Delta(\Theta^n)$ where Δ denotes the set of probability distributions over a given domain, and $\mathcal{U}$ is a profile of n utility functions where $\mathcal{U}_i : A \times \Theta^n \to \mathbb{R}$.

This paper considers "mechanism-based games," and so splits games into two parts, a mechanism and a setting. A *mechanism* is given by $\langle A, M\rangle$ where $A = A_1 \times \cdots \times A_n$, where A_i is a set of actions that agent i can perform, and M is the choice function, $M : A \to \Delta(O)$. A *Bayesian setting* is given by $\langle N, O, \Theta, p, u\rangle$ where N is a set of agents, numbered 1 to n, O is a set of outcomes, Θ is the a set of private types that an agent can have, p is the joint type distribution, $p \in \Delta(\Theta^n)$, and u is a utility function $u : N \times \Theta^n \times O \to \mathbb{R}$. Any mechanism and setting that both use the same n and O can be combined to form a game where $\mathcal{U}_i(\theta_N, a_N) = u(i, \theta_N, M(a_N))$ and where $a_N \in A$ denotes an action profile.

4 Bayesian Action-Graph Games

PosEc uses Bayesian Action-Graph Games to represent the games described in Section 3. We briefly and informally introduce BAGGs here, but refer the reader to Jiang and Leyton-Brown (2010) for more details. BAGGs are compact because they exploit *anonymity* (an agent's payoff may not depend on the specific identities of agents who played certain actions) and *context-specific independence* (an agent's payoff for playing a given action can be determined based only on the distribution of play over a strict subset of the actions). A BAGG is a directed graph in which the nodes correspond to type-action pairs. Play of the game can be thought of as each agent placing a token on one of their allowed subset of nodes. Given the locations of all of the tokens, an agent's utility can be computed by referring only to the count of tokens in the *neighborhood* of the node the agent chose. A node's neighborhood is the set of nodes having outgoing edges that point to it, with self-edges allowed. The counts of all of the nodes in a neighborhood are called a *projected configuration*, and each node stores a payoff table indexed by these configurations. We conclude by mentioning *function nodes*: no agent selects these nodes; instead, the count at a function

```
# Setting
n = 10
O = ("A", "B", "C")
Theta = [("A", "B", "C"), ("C", "B", "A")]
P = [UniformDistribution(Theta)] * n

def u(i, theta, o, a_i):
  return theta[i].index(o)

s = BayesianSetting(n, O, Theta, P, u)

# Mechanism
def A(setting, i, theta_i):
  return setting.O

def M(setting, a_N):
  scores = {o: a_N.count(o)
            for o in setting.O}
  maxScore = max(scores.values())
  winners = [o for o in scores.keys()
             if scores[o] == maxScore]
  return UniformDistribution(winners)

m = Mechanism(A, M)
agg = makeAGG(s, m, symmetry=True)
```

Figure 1: Sample code defining a mechanism and setting for plurality voting with randomized tie breaking.

node is an arbitrary deterministic function of the counts of the node's parents. For example, a sum node in a voting game might be used to count all of the agents that choose actions that influence a particular candidate's score. Since expected utility computation runtime depends asymptotically on the in-degree of action nodes, and this can be substantially reduced via the use of function nodes, they can lead to large computational savings.

5 Representing Games with PosEc

PosEc is a language that aims to make it easy for users to describe mechanisms and settings. Space constraints do not permit us to describe it in detail here; thus, we confine ourselves to examples and discussion of some of the key decisions that went into its design. The package is open source and pointers to the software, compatible equilibrium-finding algorithms, and further documentation are available at https://www.cs.ubc.ca/research/posec/.

Our modeling language is based on Python. The tuples, sets and utility functions of the mathematical representation map quite naturally to tuples, set and functions in Python. To specify a mechanism-based game, a user must create and combine a *setting* and a *mechanism*.

Figure 1 shows all the code that is required to define the mechanism and setting for plurality voting with randomized tie breaking. First a Bayesian setting is defined as consisting of ten agents, three outcomes (one in which each candidate wins), types corresponding to unique preference orderings, a uniform probability distribution for each agent over each type, and a utility function that returns the index of the elected candidate in the agent's preference ordering. Second, an action function `A` is defined as returning the actions available to a given agent (here, to vote for any candidate) and is combined with a choice function `M` that returns a distribution over outcomes given each agent's actions.[1] Finally, the `makeAGG` function is called,[2] and the structure inference algorithms described in Section 6 produce a BAGG.

One of our goals was to let users implement their utility and choice functions however they liked. Indeed, PosEc will convert any valid Python functions into a valid BAGG. However, as will be made more explicit in Section 6.1, PosEc can take advantage of game structure signaled through appropriate use of PosEc's *accessors*, which are used by the choice and utility functions to get information about agent types and actions played (i.e., any usage of `a_N` or `theta`). Note the call to `a_N.count(o)` in the choice function, which signals that the outcome can be determined based only on the *number* of agents that played a given action. Consider an anonymous mechanism with a constant number of actions c, like our voting game. The corresponding normal-form game requires $O(nc^n)$ space, while the corresponding AGG is $O(n^c)$.

In many single-good auction settings, each agent's payoff depends only on whether she is allocated the good and on her own payment. Such structure is computationally useful: an agent's utility may be computed without deriving a distribution over the entire outcome space. We call this idea *projection*. PosEc allows users to specify projected settings and mechanisms. While expressing projection structure can be more work for users, doing so can yield exponentially faster computation, because the resulting games can be much more compact than games based on the equivalent (unprojected) settings and mechanisms.

The overriding goal for the PosEc API is to allow users to precisely specify games as easily as possible. Thus, our design decisions emphasize a simple general language for users to build with, rather than a palette of options for users to choose from. Naturally, we hope that users will produce a library of reusable mechanisms and settings.

6 Structure Inference Algorithms

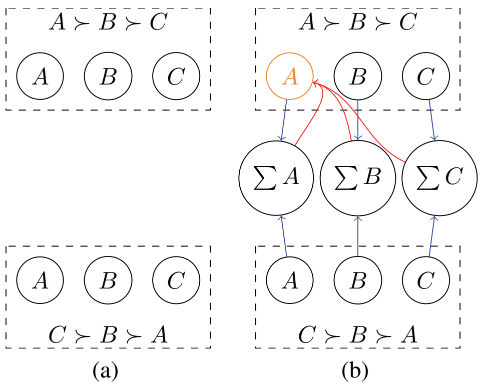
The second main component of PosEc is structure inference: automatically generating compact BAGGs given setting and mechanism descriptions in PosEc's own modeling language. We provide two approaches for doing this. First, *white-box structure inference* (WBSI) uses structure made explicit in the PosEc representation—e.g., via use of the count operator, projection, etc.—to generate a BAGG. (In the degenerate case, no such structure is explicitly given, and we obtain an exponential-size BAGG.) Second, *black-box structure inference* (BBSI) takes the BAGG generated in the first step and probes it to find additional structure to obtain a more compact BAGG.

6.1 White-Box Structure Inference

We aim to obtain what we call the *straightforward BAGG*: a BAGG that contains only those function nodes and edges that

[1]Note that PosEc requires randomized mechanisms, like our tie-breaking scheme, to return distributions over outcomes rather than performing randomization internally (e.g., via the Python `random` module). If choice functions randomized on their own, PosEc would need to sample the choice function and would then only approximate the distribution.

[2]`symmetry=True` specifies that agents of the same type share the same action node set.



$\sum A$	$\sum B$	$\sum C$	Expected Payoff
1	0	1	1
1	1	0	1.5
2	0	0	2

(c)

Figure 2: WBSI's first steps on the plurality voting example in Figure 1. Beginning with a disconnected action graph (a), WBSI selects an action node (here, voting for candidate A with type $A \succ B \succ C$) and creates an empty payoff table. As the choice function is run, WBSI will infer from accessor calls such as `a_N.count("A")` that it needs to create summation nodes aggregating the number of players that voted for each candidate across types, and that these function nodes should be inputs to the payoff table. (b) shows the action graph once the payoff table is computed, and (c) shows the payoff table (for a 2 agent game).

are necessary to compute features used by the input game. Let ℓ_s denote the representation length of this game. Our goal is to compute the straightforward BAGG using only poly(ℓ_s) calls to the utility function of the input game. We do so via a relatively direct algorithm. Essentially, it works by beginning with a totally disconnected action graph, and progressively adding function nodes and edges whenever their absence means that the utility function cannot compute a payoff. We give pseudocode for WBSI in Algorithm 1 and work through the first steps of WBSI on the plurality voting example from Section 5 in Figure 2.

We now provide two theorems about the behavior of WBSI: the first relates the size of the BAGG produced by WBSI to parameters of the game under certain conditions, while the second relates the runtime of WBSI to the size of the straightforward BAGG.

Theorem 1 *For any game parameterized by the number of agents n, the number of actions per agent m, and the number of types t, where the choice and utility functions each can make a constant number of different calls to PosEc accessor functions, and where any weighted-sum calls involve a maximum weight of w bounded by poly($n \cdot m \cdot t$), the straightforward BAGG will require only poly($n \cdot m \cdot t$) space.*

Proof sketch. WBSI introduces at most one function node per accessor call. A weighted max node, which returns an action node index, can only have as many different projected configurations as there are action nodes, of which there are at most $O(n \cdot m \cdot t)$. A weighted sum node with maximum w can only have at most $O(w \cdot n)$ different projected configurations. For each action node, there are a constant number of neighbors. Thus the possible projected configurations on the neighborhood of any action node is at most the Cartesian product of the possible projected configurations of each neighbor. Since these spaces are all poly($n \cdot m \cdot t$) and there are boundedly many of them, the total configuration space in the neighborhood of every action is at most poly($n \cdot m \cdot t$).□

Small outputs are important because the BAGGs produced by WBSI are typically used as inputs to game-solving algorithms, which often require worst-case time exponential in the size of their inputs. It is also important that WBSI be fast, so as not to become the main bottleneck.

Theorem 2 *The white-box structure-inference algorithm (Algorithm 1) runs in $O(c(\ell_s)^2)$ time, where ℓ_s denotes the size of its output, the straightforward BAGG, and c denotes the amount of time that the input code requires to compute a single agent's payoff for a single type-action-profile.*

Proof sketch. The runtime is dominated by the computation of payoff tables. The outer *for* and *repeat* loops jointly take $O(\ell_s)$ time: the *for* loop runs once per action node, and each iteration of the *repeat* loop after the first one involves creating a new edge, and both actions and edges take up space in the BAGG representation. The inner *for* loop iterates over projected configurations, where one payoff per projected configuration is also part of the BAGG representation. Because this loop only deals with BAGGs that contain weakly fewer edges than the straightforward BAGG, it can only iterate over projected-configuration spaces that are weakly smaller than the projected configuration space of the straightforward AGG. Thus, this inner loop also takes $O(\ell_s)$ time. □

As long as the outcome and utility functions passed to WBSI can be evaluated in polynomial time and make bounded numbers of calls to accessor functions, WBSI will run in polynomial time and output a polynomial-sized BAGG.

6.2 Black-Box Structure Inference

The goal of black-box structure inference is to take a BAGG obtained from white-box structure inference—in the degenerate case, a completely unstructured BAGG—and return a new BAGG that more efficiently represents the same game. Thus, BBSI is a constrained optimization problem where the feasible region is the set of all BAGGs that are equivalent to the input game and the objective is to find the smallest BAGG, measured by input length, in the feasible region. A simple, polynomial-time algorithm is to try cutting an incoming edge to an action node if this results in a strategically equivalent payoff table (e.g., in a GFP auction, an agent's payoff is unaffected by each bid less than her own). When successful, we reduce by one the dimension of that action node's payoff table. Our algorithm iterates over nodes and cuts edges as long as it is possible to do so. Pseudocode is given in Algorithm 2.

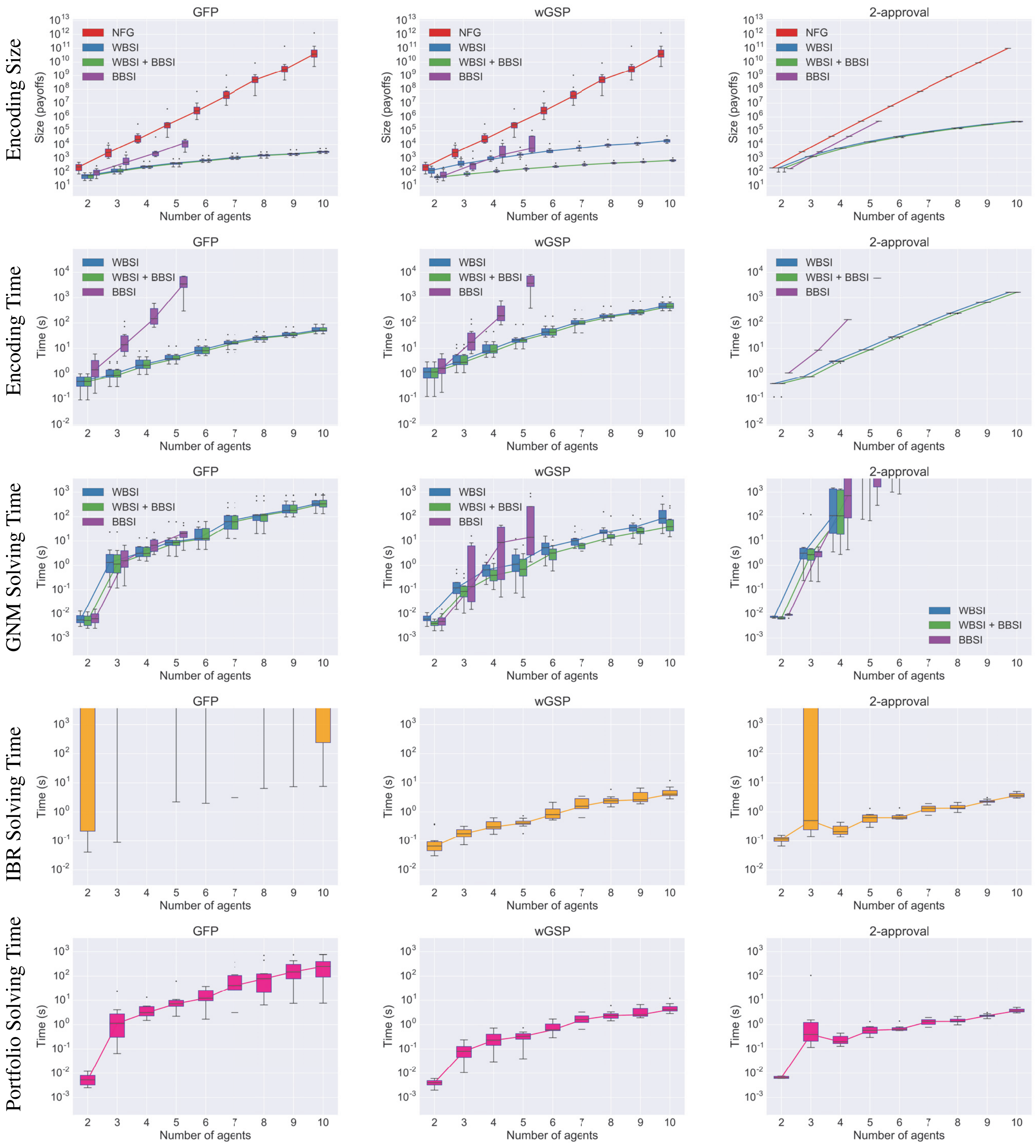
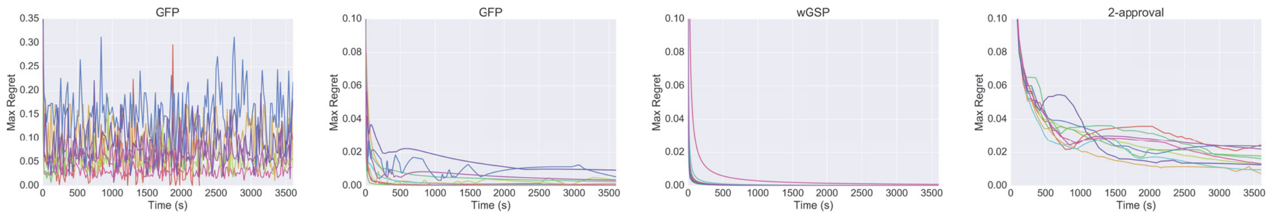


Figure 3: Encoding sizes (first row); Encoding times (second row); Median time required to identify an equilibrium using the GNM algorithm (third row), IBR algorithm (fourth row), and a parallel portfolio of GNM and IBR running on two cores (fifth row). WBSI+BBSI achieved substantially more compression than WBSI only in the case of wGSP; this suggests that previous work identified very effective encodings. We did not run BBSI on games with more than 5 players as the runtime required grew prohibitively large. All equilibrium finding plots are truncated at our budget of one hour.



(a) (b) (c) (d)

Figure 4: Median regret (across random initializations) achieved by IBR in (a) and FP in (b)–(d) on 10 agent games as a function of time; each line corresponds to a different game and regret is normalized by the maximum payoff in each game.

```
Input: Bayesian game, utility represented as a function
Output: Bayesian action-graph game
Create action nodes but no edges or function nodes
foreach Action-node $a$ do
    repeat
        create an empty payoff table for action node $a$
        finished ← True
        foreach projected config $c$ on $a$'s neighbors do
            try to compute payoff given $c$
            if success then add $c$ and payoff to table
            else
                $v$ ← missing accessor in computation
                if no function node computes $v$ then
                    add function node and edges to
                      compute $v$
                add edge from function node to $a$
                finished ← False
                break
    until finished
```
Algorithm 1: White-Box Structure Inference

```
Input: Bayesian action-graph game $G$
Output: Bayesian action-graph game $G'$ (created in
          place)
foreach action-node $a$ do
    repeat
        randomly select an edge $e$ that ends at $a$
        if $e$ can be cut without breaking strategic
          equivalence then
            cut edge $e$
            remove $e$'s column from $a$'s payoff table
    until there are no cut-able edges to $a$
    remove childless function nodes and their edges
```
Algorithm 2: Black-box structure inference

7 Experiments and Results

We now present experimental evidence that our two structure inference algorithms produce compact games in a practical amount of time and that these games can be used to compute sample Nash equilibria efficiently. To evaluate our algorithms, we turned to games for which previous work has manually identified compact encodings; they happen to be perfect-information games (thus, AGGs rather than BAGGs). Specifically, we recreated GFP and wGSP position auctions games from Thompson and Leyton-Brown (2009) and two-approval voting games from Thompson et al. (2013). We generated straightforward specifications of these settings in PosEc and then ran WBSI and BBSI. For every setting, for every number of agents, we generated 10 different games. The results are summarized in Figure 3. For our position auction games we used 4 positions, 20 bid increments, and the Varian (2007) preference model. We varied the per-agent click-through rates, valuations, and quality scores across games. We restricted to "conservative" strategies (Caragiannis et al. 2011; Roughgarden and Tardos 2012) in which bidders do not play the weakly dominated strategies of bidding above their valuations. For our two-approval games, we considered settings with 5 candidates and a variable number of voters. For each game, we randomly assigned each bidder some permutation of 0, 1, 2, 3 and 4 utility points for each of the different candidates. All of our experiments were run on Intel Xeon E5-2640 v2 processors on nodes with 96 GB of RAM.

For each game, we considered four encodings: the normal form (NFG); white box structure inference (WBSI); WBSI followed by black-box structure inference (BBSI) refinement (WBSI+BBSI); and BBSI based on the raw normal-form games. Across our three settings, WBSI always produced a BAGG which was dramatically smaller than the corresponding normal-form games. WBSI+BBSI was only able to achieve much additional compression in the case of wGSP. BBSI was considerably worse; indeed, we were unable to test it on games with more than 5 agents because runtimes for encoding grew too large.

The runtime required for WBSI encoding was generally quite manageable, with WBSI and WBSI+BBSI growing subexponentially for GFP and GSP and exponentially for 2-approval. (Even in the latter case, runtimes only reached roughly half an hour for a setting with 10 agents and 5 candidates.) The same was not true for BBSI; runtimes grew sharply, and indeed we were unable to encode games involving more than 5 agents within a reasonable amount of time for any of our settings.

We then considered the amount of time required to identify a sample Nash equilibrium of the resulting games within a CPU budget of one hour. Recall that one of PosEc's strengths is that it is not tied to any single algorithm. We first consider Iterative Best Response (IBR) and the Global Newton Method (GNM) of Govindan and Wilson (2003), extended to BAGGs by Jiang, Leyton-Brown, and Bhat (2011a). Each algorithm's runtime depends on its starting point, which we initialized randomly; we considered the median time to find an equilibrium over 10 such randomly sampled points, reporting the distribution over games for each setting. For GNM, we observed subexponential scaling for GFP and wGSP, with median runtimes of only several minutes even in the case of 10 agents.[3] In the case of 2-approval voting, we observed exponential scaling, and GNM's median runtime exceeded our CPU budget for all games with more than 6 agents.

We implemented a version of IBR that operates on the raw game specification—i.e., that does not leverage either BBSI or WBSI. Players are initialized to random pure strategies; at each iteration, a random player best responds to the current strategy (choosing among best responses at random if more than one exists). We measure the maximum regret at the end of each iteration; a regret of zero corresponds to a Nash equilibrium. IBR can only discover pure Nash equilibria. We observed that IBR timed out on most of our GFP games (see Figure 4, which illustrates its cycling behavior), but was quick to find an equilibrium in all of our wGSP games and in nearly all of our 2-approval games.

PosEc makes it easy to use multiple algorithms. Given the different strengths and weaknesses of IBR and GNM given WBSI+BBSI AGGs, we thus considered an algorithm portfolio consisting of both algorithms running in parallel on two cores. This portfolio could find an equilibrium within our time budget for all but one four-agent GFP game.[4]

Finally, because it was used by the CMA system of (Rabinovich et al. 2013) we surveyed in Section 2, we also consider the Fictitious Play (FP) algorithm. FP relies on expected utility calculations, and hence only runs efficiently on large games by leveraging AGG structure and the polytime expected utility algorithm of Jiang and Leyton-Brown (2006); we thus produced such an implementation based on WBSI+BBSI. We ran it on each of our 10 agent games given 10 random initializations, again tracking regret after each iteration. The results are shown in Figure 4. FP converged relatively quickly (though slower than IBR) for the wGSP games, but cycled or stagnated on most GFP games and all 2-approval games.

8 Conclusions and Future Work

This paper makes two major technical contributions: the PosEc declarative language for describing mechanism-based games, and two structure-inference algorithms that make it possible to compactly represent such games as BAGGs. These contributions dramatically reduce the human effort necessary to perform computational mechanism analysis without leading to a substantial loss of accuracy or speed. There are many potential applications in which PosEc could shed light on hard-to-analyze economic settings. Even within the limited and well-studied sphere of single-good auctions, PosEc could be used to study non-linear utility for money (e.g., budgets; risk attitudes), asymmetric valuation distributions, other-regarding preferences (both altruism and spite), and conditional type dependence (including common and affiliated values).

One limitation of the current PosEc system is that it can only describe simultaneous-move games. In contrast, many real-world mechanisms proceed in multiple stages (e.g., sequential auctions; clock-proxy auctions). In such games, decisions made in one stage can affect which outcomes are possible or desirable in the next stage. BAGGs inherently represent single-stage games, but could be used to analyze multi-stage mechanisms by representing the different stages as individual BAGGs and solving the complete system by a process of backward induction. Such a process would likely resemble the special-purpose algorithm that Paes Leme, Syrgkanis, and Tardos (2011) proposed for computing the equilibria of sequences of single-good auctions. Alternatively, analysis could be performed using a compact game representation that explicitly supports multi-stage games, such as temporal AGGs (Jiang, Leyton-Brown, and Pfeffer 2009) or MAIDs (Koller and Milch 2003). Unfortunately, the algorithms for reasoning about such representations currently offer much poorer performance than algorithms for reasoning about BAGGs.

Another limitation of the PosEc system is the cost of explicitly representing types and actions: no BAGG can be asymptotically smaller than its number of types or actions. Thus, games with large type or action spaces—such as combinatorial auctions—cannot be succinctly represented. There is very little work on representing games with implicitly specified action spaces—Koller and Milch (2003) and Ryan, Jiang, and Leyton-Brown (2010) are the two exceptions, unfortunately both without good implementations—but as this literature develops there may be opportunities for extending our encode-and-solve CMA approach to new game families.

A third limitation is the requirement to discretize actions and (in the case of Bayesian games), types. We note that many real-world mechanisms do discretize actions; thus, we argue that PosEc's ability to give insight into such settings is also one of its benefits.

Finally, as mentioned earlier, this paper considered only non-Bayesian-game settings, even though PosEc supports BAGGs as well as AGGs. This was simply because we wanted to compare to manual AGG encodings, which existed only in the case of perfect-information games. In future work, we intend to investigate the computational cost of computational mechanism analysis in Bayesian game settings.

[3]One exception: a single 4-agent GFP game was not solved for any of our GNM starting points.

[4]Other settings might benefit from different algorithm portfolios. In particular, we note that Simplicial Subdivision (van der Laan, Talman, and van Der Heyden 1987; Jiang, Leyton-Brown, and Bhat 2011b) and the Support Enumeration Method (Porter, Nudelman, and Shoham 2008; Thompson, Leung, and Leyton-Brown 2011) have been extended to handle BAGGs efficiently.

References

Asimov, I. 1942. Runaround. In *Astounding Science Fiction*.

Caragiannis, I.; Kaklamanis, C.; Kanellopoulos, P.; Kyropoulou, M.; Lucier, B.; Paes Leme, R.; and Tardos, E. 2011.

On the efficiency of equilibria in generalized second price auctions. In *EC*.

Duong, Q.; Vorobeychik, Y.; Singh, S.; and Wellman, M. P. 2009. Learning graphical game models. In *IJCAI*.

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

Jiang, A. X., and Leyton-Brown, K. 2006. A polynomial-time algorithm for action-graph games. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 21, 679. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Jiang, A. X., and Leyton-Brown, K. 2010. Bayesian action-graph games. In *NIPS*.

Jiang, A. X.; Leyton-Brown, K.; and Bhat, N. A. R. 2011a. Action-graph games. *GEB* 71:141–173.

Jiang, A. X.; Leyton-Brown, K.; and Bhat, N. A. 2011b. Action-graph games. *Games and Economic Behavior* 71(1):141–173.

Jiang, A. X.; Leyton-Brown, K.; and Pfeffer, A. 2009. Temporal action-graph games: A new representation for dynamic games. In *UAI*.

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

Koller, D., and Milch, B. 2003. Multi-agent influence diagrams for representing and solving games. *GEB* 45:181–221.

Paes Leme, R.; Syrgkanis, V.; and Tardos, E. 2011. Sequential auctions and externalities. In *SODA*.

Porter, R.; Nudelman, E.; and Shoham, Y. 2008. Simple search methods for finding a nash equilibrium. *GEB* 63:642–662.

Rabinovich, Z.; Naroditskiy, V.; Gerding, E. H.; and Jennings, N. R. 2013. Computing pure Bayesian Nash equilibria in games with finite actions and continuous types. *AIJ* 195:106–139.

Roughgarden, T., and Papadimitriou, C. 2008. Computing correlated equilibria in multi-player games. *JACM* 37:49–56.

Roughgarden, T., and Tardos, E. 2012. Do externalities degrade GSP's efficiency? In *Workshop on Advertising Auctions*.

Ryan, C. T.; Jiang, A. X.; and Leyton-Brown, K. 2010. Computing pure strategy Nash equilibria in compact symmetric games. In *EC*.

Thompson, D. R. M., and Leyton-Brown, K. 2009. Computational analysis of perfect-information position auctions. In *ACM-EC*.

Thompson, D. R. M., and Leyton-Brown, K. 2013. Revenue optimization in the generalized second-price auction. In *EC*.

Thompson, D. R. M.; Lev, O.; Leyton-Brown, K.; and Rosenschein, J. 2013. Empirical analysis of plurality election equilibria. In *AAMAS*.

Thompson, D. R. M.; Leung, S.; and Leyton-Brown, K. 2011. Computing Nash equilibria of action-graph games via support enumeration. In *WINE*.

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

Varian, H. R. 2007. Position auctions. *International Journal of Industrial Organization* 25:1163–1178.

Vorobeychik, Y., and Wellman, M. P. 2008. Stochastic search methods for Nash equilibrium approximation in simulation-based games. In *AAMAS*.

Vorobeychik, Y.; Reeves, D. M.; and Wellman, M. P. 2012. Constrained automated mechanism design for infinite games of incomplete information. *JAAMAS* 25:313–351.