Data Structures for Deviation Payoffs

Bryce Wiedenbeck
Davidson College
United States
brwiedenbeck@davidson.edu

Erik Brinkman
Independent Researcher
United States
erik.brinkman@gmail.com

ABSTRACT

We present new data structures for representing symmetric normal-form games. These data structures are optimized for efficiently computing the expected utility of each unilateral pure-strategy deviation from a symmetric mixed-strategy profile. The cumulative effect of numerous incremental innovations is a dramatic speedup in the computation of symmetric mixed-strategy Nash equilibria, making it practical to represent and solve games with dozens to hundreds of players. These data structures naturally extend to role-symmetric and action-graph games with similar benefits.

KEYWORDS

Game Theory; Equilibrium Computation; Data Structures

1 INTRODUCTION

Symmetric games, where all players face identical incentives, play a central role in game-theoretic analysis. Many of the classic examples used to teach game-theoretic concepts [15, 24] such as prisoner's dilemma and rock-paper-scissors are symmetric, and the original paper proposing the concept of a Nash equilibrium [19] addressed symmetry and proved that symmetric games must have symmetric equilibria. Symmetric games are quite common in the recent multi-agent systems literature [2, 28]. The role of symmetry becomes especially prominent in games with a large number of players, because it is often only with the help of player symmetry that incentives can even be tractably described, and multiple distinct sub-fields of computational game theory [4, 29] rely heavily on foundational assumptions about player symmetry to scale up game-theoretic analysis.

Despite this importance, data structures for representing and solving large symmetric games have received a paucity of attention in the research literature. Most libraries for representing and solving games, particularly Gambit [18], but also GameTracer [1] and QuantEcon [21], include few tools for efficiently solving large symmetric games. One notable exception is the work of Jiang et al. [14] on action-graph games, which incorporate player symmetry, and which have been partially implemented in Gambit. However, action-graph games focus primarily on theoretical compactness in the action space, and only consider expected utility calculations enough to ensure they take polynomial time. The data structures

that have been implemented are not specifically optimized for equilibrium computation under player symmetry, meaning that solving large symmetric games remains inefficient.

In the sub-fields that scale up symmetric game analysis, approximations play a central role. Simulation-based games [29] frequently employ player reduction methods [30] that use a reduced game with a very small number of players to replace a game with dozens or hundreds of players in the hope that analysis of the reduced-game will approximately hold in the large game. Mean-field games [4] give up on representing discrete players and replace them with an average effect that aggregates over a continuum. Other approaches aim to identify underlying representational compactness [9, 16] or to avoid explicitly constructing game models in favor of sampling-based techniques [10, 25]. These approximations are sometimes unavoidable, when the representation is too large or the payoff computations are too slow for a game to even be enumerated, but analysts often end up resorting to approximation when faced with any non-trivial number of players. The result is a substantial gap between the very small games that can be represented and solved exactly and the very large games where approximation is necessary. We bridge this gap by designing efficient data structures that make it practical to exactly solve much larger instances.

In this work, we present a detailed exploration of data structure improvements for symmetric normal-form games, with specific focus on the task of computing symmetric mixed-strategy Nash equilibria. We argue, following Cheng et al. [6], that this is by far the most compelling solution concept for symmetric games, because symmetric equilibria are guaranteed to exist, and it provides greater intuitive explanatory power for symmetries of the game to be reflected in its equilibria. While computing symmetric Nash equilibria can be hard in the worst case [7, 8], incomplete local-search algorithms such as replicator dynamics and gradient descent are often highly successful in practice.

To facilitate algorithms for identifying symmetric equilibria, we aim for data structures that optimize the calculation of *deviation payoffs* (and their derivatives). For a symmetric mixed strategy employed by all players, the deviation payoff vector gives, for each action, the expected utility a single agent would receive if they deviated unilaterally to that action. For a large fraction of the algorithms used to compute Nash equilibria, the ability to evaluate deviation payoffs for symmetric mixed strategies is both necessary and sufficient; for most other algorithms, deviation payoffs plus their derivatives suffice. We show that focusing the design of symmetric game data structures toward deviation payoffs leads to a number of optimizations that jointly yield a dramatic improvement in the practical efficiency of solving large symmetric games.

1.1 Contributions

We describe seven distinct upgrades to classic data structures for representing symmetric normal-form games. Two of these (sections 3.3 and 3.4) constitute asymptotic improvements: changing from P-player profiles to $(P-1)$-opponent configurations reduces the number of stored payoff entries (with A actions) from $A\binom{P+A-1}{P}$ to $A\binom{P+A-2}{P-1}$, and pre-computing probability weights to avoid repeated multinomial calculations accelerates equilibrium searches by a factor of A. Four of the upgrades focus on vectorization, resulting in better constants and enabling SIMD acceleration. And we find—as has been well-established in the settings of neural networks and scientific computation—that such improvements can qualitatively change the scope of the computational tools. The remaining upgrade (section 3.6) follows because the overall expansion-of-scope enables us to analyze games so large that the probability calculations can overflow 64-bit integers, necessitating a switch to a log-space representation of payoffs and probabilities.

Our main result is a roughly ten-thousand-fold speedup in the running time of equilibrium computation algorithms for symmetric games with many players. This makes it possible to run practical-but-incomplete search methods like replicator dynamics on any symmetric game that can fit in memory, and can also facilitate other slower equilibrium-search techniques. Our results effectively close the gap between small and large symmetric games, relegating approximation techniques to only those games too large to be represented explicitly.

Two open-source libraries implement most of the data structures and algorithms we discuss. The gameanalysis.jl[1] repository provides simple Julia implementations of the best CPU and GPU versions of our symmetric game data structure, and also includes all of the code for experiments in this paper. The gameanalysis.py[2] module implements in Python the data structure variants for role-symmetric games that we discuss in Section 5.2 and provides a richer command-line interface with numerous tools for solving role-symmetric games.

2 BACKGROUND

2.1 Terminology

In a symmetric game, all players have the same action set and identical incentives. We therefore refer to the number of players P, but rarely distinguish individual players. We call the number of actions A, and will often index the actions by $a \in \{1, \ldots, A\}$.

A *profile* specifies an action for each player, and in a symmetric game, we can represent a profile by an integer-vector $\vec{s}$ specifying a non-negative number of players selecting each action. We denote the entries in a vector with a subscript, so $\vec{s}_a$ is the player-count for action a. We use a superscript i or j to index vectors in a collection: for example $\vec{s}^i$; to avoid ambiguity, exponents are generally applied to a quantity in parentheses. We will also distinguish a profile from an *opponent configuration* $\vec{c}$, which differs only in that $\sum_a \vec{s}_a = P$, while $\sum_a \vec{c}_a = P - 1$. We refer to the configuration resulting from removing action a from profile $\vec{s}$ as $(\vec{s} \mid a)$, since it will often appear in probability calculations and other contexts where it is *given* that

one player selects action a. In terms of the integer-vector representation, $(\vec{s} \mid a)$ subtracts 1 from dimension a of $\vec{s}$. A symmetric game's payoffs can be expressed as the value achieved by a player choosing action a when opponents play configuration $\vec{c}$, which we denote by $v_a(\vec{c})$ normally, or by $v_a(\vec{s} \mid a)$, when working in terms of profile $\vec{s}$.

A mixed strategy specifies a probability distribution over actions, and a mixed strategy used by all players is called a symmetric mixed-strategy profile. We denote this with the variable $\vec{\sigma}$, and will often abbreviate the term by referring to a *mixture*. When computing probabilities, we will frequently refer to the number of asymmetric arrangements corresponding to a symmetric configuration, which we call *repetitions*. We denote this quantity by $\text{Reps}(\vec{c})$ or $\text{Reps}(\vec{s} \mid a)$, and calculate it with the following multinomial:

$$\text{Reps}(\vec{c}) = \binom{P-1}{\vec{c}_1, \vec{c}_2, \ldots, \vec{c}_A} = \frac{(P-1)!}{\vec{c}_1! \vec{c}_2! \ldots \vec{c}_A!} \tag{1}$$

2.2 Deviation Payoffs

When analyzing a symmetric game, we are most often interested in computing symmetric mixed-strategy Nash equilibria. For many algorithms that compute such equilibria a necessary and sufficient condition is the ability to compute deviation payoffs, and for most other algorithms, deviation payoffs plus deviation derivatives suffice. We begin by formally defining these terms, and then describe their application to our preferred equilibrium computation algorithms in section 2.3.

Given a symmetric mixed-strategy profile $\vec{\sigma}$, we define the *deviation payoff* $\vec{u}_a(\vec{\sigma})$ for action a as the expected utility one player would receive if they played a while all opponents randomized according to $\vec{\sigma}$. This expectation is often expressed as a sum over all profiles in which a is played of the probability that profile occurs times the payoff to a in that profile:

$$\vec{u}_a(\vec{\sigma}) = \sum_{\vec{s}: \vec{s}_a > 0} \Pr_{\vec{\sigma}}(\vec{s} \mid a) \, v_a(\vec{s} \mid a) \tag{2}$$

but can be stated much more cleanly using configurations:

$$\vec{u}_a(\vec{\sigma}) = \sum_{\vec{c}} v_a(\vec{c}) \Pr_{\vec{\sigma}}(\vec{c}) \tag{3}$$

$$= \sum_{\vec{c}} v_a(\vec{c}) \, \text{Reps}(\vec{c}) \prod_{a'} (\vec{\sigma}_{a'})^{\vec{c}_{a'}} \tag{4}$$

The deviation payoff vector $\vec{u}(\vec{\sigma})$ collects these values for all actions $a \in \{1, \ldots, A\}$. We call the partial derivatives of deviation payoffs with respect to mixture probabilities *deviation derivatives*. Specifically, $\frac{\partial \vec{u}_a(\vec{\sigma})}{\partial \vec{\sigma}_s}$ gives the change in the deviation payoff for action a as the probability of action s is varied. Again this can be expressed in terms of profiles, but is more straightforward in terms of configurations: [3]

$$\frac{\partial \vec{u}_a(\vec{\sigma})}{\partial \vec{\sigma}_s} = \sum_{\vec{c}} v_a(\vec{c}) \text{Reps}(\vec{c}) (\vec{c}_s) (\vec{\sigma}_s)^{\vec{c}_s - 1} \prod_{a' \neq s} (\vec{\sigma}_{a'})^{\vec{c}_{a'}} \tag{5}$$

---

[1]https://github.com/Davidson-Game-Theory-Research/gameanalysis.jl
[2]https://github.com/egtaonline/gameanalysis

[3]Note that efficient computation of $(\vec{\sigma}_s)^{\vec{c}_s - 1}$ can result in numerical errors for mixtures where $\vec{\sigma}_s = 0$. This sort of error can be avoided here and elsewhere with no real loss of precision by lower-bounding mixture probabilities at machine-epsilon.
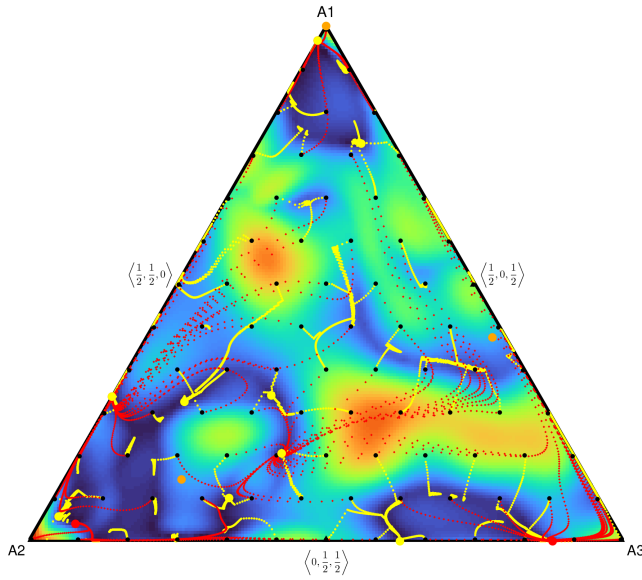
Figure 1: Execution traces for both replicator dynamics (red) and gradient descent (yellow) on a 100-player, 3-action game. The underlying heatmap shows mixture regret. Black points are starting mixtures. Large points indicate ε-equilibria. The 3 orange equilibria were found by both algorithms. RD found 2 unique equilibria, while GD found 6.

We can define most other standard game-theoretic quantities for symmetric games in terms of deviation payoffs. The expected utility experienced by all players when following a symmetric mixed strategy is given by the dot product $u(\vec{\sigma}) = \vec{u}(\vec{\sigma}) \cdot \vec{\sigma}$. The regret of a mixture is $\text{reg}(\vec{\sigma}) = \max_a (\vec{u}_a(\vec{\sigma}) - u(\vec{\sigma}))$. A symmetric Nash equilibrium is a mixture with $\text{reg}(\vec{\sigma}) = 0$, while an approximate Nash equilibrium has $\text{reg}(\vec{\sigma}) \leq \varepsilon$, for suitable ε.

2.3 Equilibrium Computation

Computing deviation payoffs and/or their derivatives is the key step for a number of algorithms that identify Nash equilibria in symmetric games. We describe two of the most practical algorithms here: replicator dynamics, which depends on deviation payoffs, and gradient descent on sum-of-gains, which depends on deviation derivatives. Our data structures can also efficiently support a number of other Nash algorithms, including fictitious play [3], Scarf's simplicial subdivision [22], and the global Newton method of Govindan and Wilson [11], as well as some algorithms for correlated equilibria [13]. The Nash Algorithms Appendix[4] presents further details on how some of these algorithms depend on deviation payoffs.

Replicator dynamics. [26] is often presented as a rule governing evolutionary dynamics, but can also be viewed as an algorithm for computing symmetric Nash equilibria. Expressed in terms of deviation payoffs, replicator dynamics starts from some initial mixture $\vec{\sigma}^0$ at $t = 0$, and performs iterative updates of the form:

---

[4]See Appendix C: Nash Algorithms. https://arxiv.org/abs/2302.13232

$$\vec{w}_a \leftarrow \vec{\sigma}_a^t \, \vec{u}_a(\vec{\sigma}^t)$$

$$\vec{\sigma}_a^{t+1} \leftarrow \frac{\vec{w}_a}{\sum_{a'} \vec{w}_{a'}}$$

This update assumes that all payoffs are non-negative; a positive affine transformation can be applied to any game to ensure this assumption holds (and to adjust the effective step-size).

Gradient descent. is a classic local-search algorithm for minimizing differentiable functions. We can easily define a function whose minima correspond to Nash equilibria based on the following sum of deviation gains:

$$g(\vec{\sigma}) = \sum_a \max\left(0, \vec{u}_a(\vec{\sigma}) - \vec{u}(\vec{\sigma}) \cdot \vec{\sigma}\right)$$

Then we can iteratively take steps in the direction of $-\nabla g(\vec{\sigma})$. The elements of this gradient vector are given by:

$$\nabla_s(g) = \sum_a \left(\frac{\partial \vec{u}_a(\vec{\sigma})}{\partial s} - \vec{u}_s(\vec{\sigma}) - \sum_{a'} \vec{\sigma}_{a'} \frac{\partial \vec{u}_{a'}(\vec{\sigma})}{\partial s} \right) 1_{g_s}$$

Where 1_{g_s} is an indicator variable for $\vec{u}_s(\vec{\sigma}) > \vec{u}(\vec{\sigma}) \cdot \vec{\sigma}$. So deviation payoffs and deviation derivatives suffice to compute the gain gradient. When performing gradient descent, the mixture resulting from $\vec{\sigma} - \nabla g(\vec{\sigma})$ may not lie on the probability simplex, so it is necessary to project each step back onto the simplex, which we do using the method from Wang and Carreira-Perpinán [27].

Neither replicator dynamics nor gradient descent is guaranteed to identify a Nash equilibrium. However, since these algorithms are many orders of magnitude faster than complete algorithms like simplicial subdivision or global Newton, it is practical to re-run them from many initial mixtures. We find that both algorithms tend to identify multiple equilibria in large symmetric games and that the sets of equilibria they find are often only partially overlapping. On the other hand, we find that fictitious play and other best-response-based updates are ineffective on large symmetric games. Therefore in practice, we recommend repeatedly running both replicator dynamics and gradient descent, filtering by regret to isolate ε-Nash mixtures, and merging the resulting equilibrium sets. An example trace from running both algorithms on a 100-player, 3-action Gaussian mixture game (represented using our data structures) appears in Figure 1.

3 DATA STRUCTURE IMPROVEMENTS

The classic payoff-matrix representation of a normal-form game has a dimension for each player, and a size along each dimension equal to the number of actions available to that player. In a symmetric game, it suffices to store just one player's payoffs, so if there are P players and A actions, a symmetric game can be represented by a symmetric tensor of size A^P.

Because symmetric tensors also arise in other settings, it is worth considering whether generic techniques for symmetric tensors would suffice for efficiently representing symmetric games. In particular, Schatz et al. [23] propose a block-representation of symmetric tensors and a cache-conscious algorithm for multiplying with them. In principle, their representation has the same level
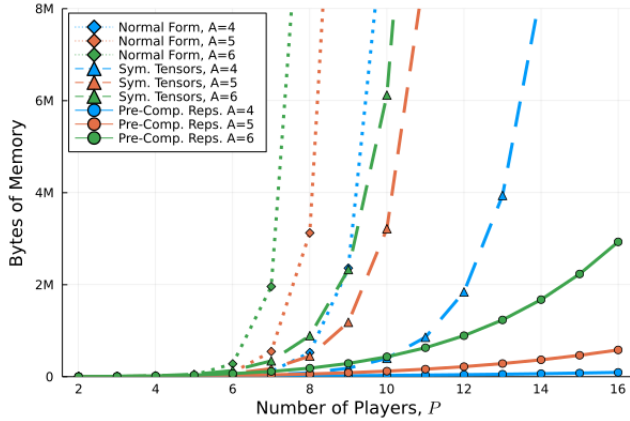
Figure 2: The normal-form (dotted lines) and symmetric-tensor (dashed lines) representations use significantly more memory than the largest of our proposed data structures.

of asymptotic compression as the data structures presented here. However, in Figure 2, we compare the memory footprint of the Schatz et al. symmetric-tensor representation (as implemented by the `SymmetricTensors.jl` library) against the largest of our symmetric-game data structures (pre-computed repetitions), showing that while the block-symmetric representation is better than storing the full payoff matrix, it is significantly larger than any of our data structure variants.

3.1 Payoff Dictionary

We therefore turn to purpose-built data structures for storing symmetric games. The classic approach is to store a mapping from profiles to payoffs. In Gambit [18] the implementation of symmetric action-graph games stores this mapping in a trie, while others have used a database [5]. Such a mapping is also implicit in many theoretical descriptions, for example when relating the size of a compact representation to the number of profiles [12, 14]. In our experiments, our baseline data structure stores this mapping using a hash table that maps a vector of integers representing the profile $\vec{s}$ to a vector of floats that stores the payoff $v_a(\vec{s})$ for each played action with $\vec{s}_a > 0$. With P players and A actions, this table will store $\binom{P+A-1}{P}$ profiles.

Calculating a deviation payoff $u_a(\vec{\sigma})$ using this mapping requires iterating through all profiles to compute the sum from Equation (2), where the probabilities are given by this expression:

$$\Pr_{\vec{\sigma}}(\vec{s} \mid a) = \text{Reps}(\vec{s} \mid a)(\vec{\sigma}_a)^{\vec{s}_a - 1}\prod_{a' \neq a}(\vec{\sigma}_{a'})^{\vec{s}_{a'}} \quad (6)$$

$\text{Reps}(\vec{s} \mid a)$, calculated according to the multinomial from Equation (1), gives the number of asymmetric orderings of the symmetric configuration $(\vec{s} \mid a)$, while the remaining terms gives the probability of the opponents jointly playing one such asymmetric ordering. The Worked Examples Appendix shows the full mapping and a detailed walk-through of the deviation payoff calculation on an example 3-player, 3-strategy symmetric game for this version of the data structure as well as for all subsequent variants. We strongly

recommend stepping through these examples to help resolve any confusion that arises from the data structure and algorithm descriptions that follow.[5]

3.2 Array Vectorization

The first idea for improving this deviation payoff calculation is to store the profiles $\vec{s}$ and the payoffs v in a pair of two-dimensional arrays with parallel structure, denoted by a shared index i over all profiles. In both arrays, each row corresponds to an action and each column corresponds to a profile, resulting in arrays of size $A \times \binom{P+A-1}{P}$. Extracting column i of the profiles-array gives a profile vector $\vec{s}^i$ with the count for each action. The corresponding column of the payoffs-array stores the payoff $v_a(\vec{s}^i \mid a)$ for each action a where that profile has a non-zero count.

Of note, the array representation here (and in all subsequent data structure variants) hinders direct look-up of payoffs for a pure-strategy profile using the mapping. However, this trade-off is clearly worthwhile for several reasons:

(1) Calculating mixed-strategy deviation payoffs is a much tighter bottleneck than looking up pure-strategy payoffs.
(2) Symmetric mixed-strategy equilibria in symmetric games are more often relevant than asymmetric pure-strategy equilibria (and are guaranteed to exist).
(3) The profile-to-index mapping can be computed by a ranking algorithm for combinations-with-replacement [17], or can be stored alongside the arrays for fast lookup with only linear additional memory.

Using this array representation, we can vectorize each of the steps in the calculation of the deviation payoff $\vec{u}_a(\vec{\sigma})$. We describe each of the following operations in terms of profile i, and broadcast those operations across array columns. First, we can compute a mask m identifying the profiles where action a is played:

$$m^i \leftarrow \vec{s}_a^i \neq 0 \quad (7)$$

Then for each such profile, we can remove a player choosing a to get the opponent-configuration $\vec{c}^i = (\vec{s}^i \mid a)$ by subtracting an indicator vector $\vec{e}_a$ for action a:

$$\vec{c}^i \leftarrow \vec{s}^i - \vec{e}_a \quad (8)$$

For each profile, the probability $p^i = \Pr(\vec{c}^i)$ of the opponent configuration is calculated by a multinomial to compute repetitions along with a broadcast-and-reduce to apply exponents and take the product, resulting in an array of configuration probabilities:

$$p^i \leftarrow \text{Reps}\left(\vec{c}^i\right)\prod_{a'}(\vec{\sigma}_{a'})^{\vec{c}_{a'}^i} \quad (9)$$

Finally the deviation payoff for action a can be computed by extracting row a of the payoffs array, multiplying element-wise by the configuration probabilities, masking out the invalid configurations, and summing over the remaining profiles:[5]

$$\vec{u}_a(\vec{\sigma}) \leftarrow \sum_i m^i\, p^i\, v_a\left(\vec{c}^i\right)$$

Since these steps can be performed using only arithmetic primitives and indexed reduction, the computation is vectorizable in any numeric library. Further vectorization over actions is in principle

---

[5]See Appendix A: Worked Examples. https://arxiv.org/abs/2302.13232

possible, but we defer this to later sections. The degree of improvement from array vectorization will vary with the language used. More importantly, it sets the stage for our subsequent innovations.

3.3 Pre-computed Repetitions

The next improvement comes from recognizing that each time we compute a deviation-payoff vector $\vec{u}(\vec{\sigma})$, we make use of the $\mathrm{Reps}\,(\vec{s}\mid a)$ value for every profile-action pair. Since equilibrium search involves many, many deviation-payoff calculations, we can save significant work by pre-computing these repetitions and storing them in a third parallel array. Then the probability calculation in Equation (9) can extract row a of this repetitions array and element-wise multiply by the array of products. Our `gameanalysis.jl` library[1] and the Worked Examples Appendix[5] each include a version of this data structure that computes $\vec{u}(\vec{\sigma})$ in a single pass instead of computing $\vec{u}_a(\vec{\sigma})$ separately for each action, but we defer the detailed explanation of this vectorization to the following variant where it is dramatically simplified.

3.4 Opponent Configurations

A closer inspection of these parallel array structures reveals some redundancies: first the payoffs array contains a number of meaningless entries, since whenever $\vec{s}_a = 0$ the corresponding v_a is masked out of the calculations.[5] Second, the repetitions array contains a number of redundant entries: whenever two profiles differ by a single player's action, that is $(\vec{s}\mid a) = (\vec{s}'\mid a')$ we will end up storing identical entries for $\mathrm{Reps}\,(\vec{s}\mid a)$ and $\mathrm{Reps}\,(\vec{s}'\mid a')$.

Both of these issues can be avoided if we switch to storing opponent-configurations, with a shared index j over all configurations. This gives us two parallel arrays of size $A \times \binom{P+A-2}{P-1}$ for configurations and payoffs, and a $1 \times \binom{P+A-2}{P-1}$ repeats-array. The configurations-array now stores each configuration $\vec{c}^j$ over $P-1$ opponents. Each corresponding column of the repeats-array stores $\mathrm{Reps}\,(\vec{c}^j)$, and the same column of the payoffs-array stores, for each action a, the payoff $v_a(\vec{c}^j)$ for a player choosing a when their opponents play according to $\vec{c}^j$.

This also simplifies the deviation-payoffs calculation. We compute probability $p^j = \mathrm{Pr}(\vec{c}^j)$ of a configuration in same way as before, except we can skip the steps from Equations 7 and 8, instead simply accessing configuration $\vec{c}^j$ from storage:

$$p^j \leftarrow \mathrm{Reps}\left(\vec{c}^j\right) \prod_{a'} (\vec{\sigma}_{a'})^{\vec{c}^j_{a'}}$$

The key difference in this equation is that we compute the probability of a configuration j, instead of the probability of a configuration derived from a masked profile i. This change removes the need to mask payoffs and means that the configuration probabilities are identical for all deviations, meaning we do not have to perform a separate probability calculation for each action. This single array of configuration probabilities in turn simplifies vectorization over deviation actions, easily giving us the whole deviation-payoff vector at once. To get deviation payoffs, we multiply the configuration probabilities p^j by the payoffs-array (broadcasting over actions), and sum over the configurations dimension.

$$\vec{u}(\vec{\sigma}) \leftarrow \sum_j p^j \vec{v}^j \tag{10}$$

Here $\vec{v}^j$ refers to a column of the payoffs-array, and summing these configuration-payoff vectors gives us a vector of deviation payoffs. The code for this version is the easiest to understand, so despite our recommended further improvements, we include it in the Julia Code Appendix.[6]

3.5 Pre-Weighting by Repetitions

The next improvement comes from the realization that by associativity we can re-group Equation (10) as follows.

$$\vec{u}_a(\vec{\sigma}) \leftarrow \sum_j \left(v_a\left(\vec{c}^j\right) \mathrm{Reps}\left(\vec{c}^j\right)\right) \prod_{a'} (\vec{\sigma}_{a'})^{\vec{c}^j_{a'}} \tag{11}$$

Which means that even though repetitions are logically part of the probability calculation, we can simplify our computations by storing them with the payoff values. Specifically, we can combine the repeats-array into the payoffs-array by multiplying each payoff value by that configuration's repetitions, so that the entry in row a, column j of the payoffs array stores the value:

$$\mathrm{Reps}\left(\vec{c}^j\right) v_a\left(\vec{c}^j\right)$$

This operation can be performed once in advance, saving space and speeding up all subsequent deviation payoff calculations.[5]

3.6 Log Transformation

The combined effect of all the improvements so far is sufficient to allow reasonably efficient deviation payoff calculations for games with over 100 players, as long as the number of actions is kept small, but this poses a new problem: $\binom{32}{6,6,6,7,7} > 2^{63}$, meaning that the repetitions overflow a 64-bit integer for some profiles with $P = 33$ and $A = 5$. We can solve this problem by working with log-probabilities, and this incidentally produces a slight computational speed-up by making use of arithmetic operations that consume fewer processor cycles when we transform exponents into multiplication and multiplication into addition. Specifically, we can store the natural (or other base) log of the repetition-weighted payoffs λ, and to calculate repetitions, we can use a log-gamma function to avoid overflows.

$$\lambda_a^j \leftarrow \log\left(\mathrm{Reps}\left(\vec{c}^j\right)\right) + \log\left(v_a(\vec{c}^j)\right)$$

This will not work if payoffs are negative, but since any positive affine transformation of utilities has no effect on incentives, we can transform any game into one with non-negative payoffs. In fact, we find it useful under all of our game representations to transform the payoffs into a standardized range, to simplify hyperparameter tuning for various equilibrium-computation algorithms. And of course these calculations can be vectorized as before with the exponential applied element-wise to an array with contribution-values for each configuration.

We can now compute deviation payoffs using this representation by first computing each configuration's contribution γ^j to the deviation payoff in log-space.

$$\gamma^j \leftarrow \exp\left(\lambda^j + \sum_{a'} \vec{c}^j_{a'} \log \vec{\sigma}_{a'}\right) \tag{12}$$

---

[6]Implementation shown in Appendix B: Julia Code. https://arxiv.org/abs/2302.13232

and then summing over all configurations.[5]

$$\vec{u}_a(\vec{\sigma}) \leftarrow \sum_j \gamma^j$$

3.7 GPU Acceleration

Now that our deviation-payoff calculations are done as a sequence of simple mathematical operations broadcast across large arrays of floating-point data, an obvious way to accelerate them is to move the computation to a graphics processor. Most modern programming languages have libraries that make GPU translation manageable for operations as simple as ours, and specifically in Julia the translation is trivial, requiring only that we choose an array data type that moves the data to the GPU; the code for CPU and GPU versions of our final data structure is otherwise identical.[6]

3.8 Batch Processing

The final improvement we implemented requires no change to the preceding data structure, but instead uses it more efficiently. If memory allows, we can take even greater advantage of SIMD operations on the GPU by computing deviation payoffs for a batch containing multiple mixtures.[6] This is useful because many of the algorithms we use to compute Nash equilibria are forms of local search where it can help to initialize them from many starting points. For example, in both replicator dynamics and gain-gradient descent, we generally restart the algorithm as many as 100 times from different initial mixtures, so by parallelizing deviation-payoff calculations, we can parallelize multiple restarts of the algorithm. This adds an extra dimension to the intermediate arrays constructed by the deviation payoff computation, giving them dimension actions × configurations × mixtures. This is mainly helpful with small games, since large games can occupy all GPU execution units with the deviation-payoff calculation for a single mixture. If batch processing is used on large games, the number of mixtures must be carefully chosen to not exceed available GPU memory.

3.9 Deviation Derivatives

For any of the data structure variants we describe, it is possible to calculate deviation derivatives, but here we describe the approach only for the final log-transformed variants.

Computing the Jacobian with respect to the mixture probability $\vec{\sigma}_s$ only requires a small addition to the deviation payoff contributions computed in equation (12); we multiply by $\frac{\vec{c}_s}{\vec{\sigma}_s}$ over a new dimension s, before summing across configurations.[6]

$$\frac{\partial \vec{u}_a(\vec{\sigma})}{\partial \vec{\sigma}_s} \leftarrow \sum_j \frac{\vec{c}_s^j}{\vec{\sigma}_s} \gamma^j$$

Table 1: For a given number of actions A, the largest number of players P before *repetitions* overflows a 64-bit integer.

A	2	3	4	5	6	7	8	9	10
max P	67	44	36	32	29	27	26	25	25

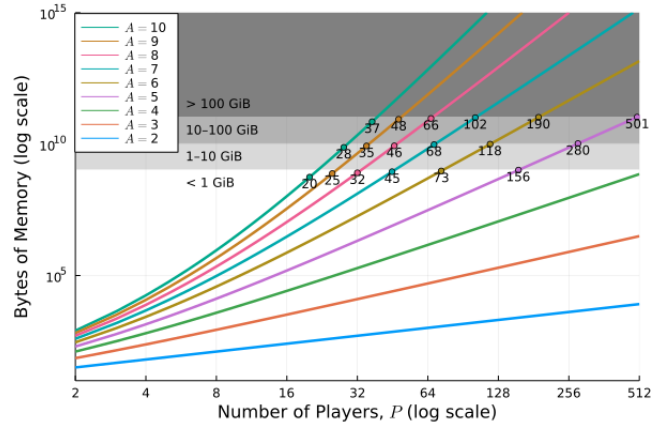| A	11	12	13	14	15	16	17	18	≥ 19
max P	24	23	23	23	23	22	22	22	21

Figure 3: GPU memory required to store the final version of our data structure. Batch-computation of deviation payoffs increases memory footprint by a linear factor.

This calculation can be fully vectorized using similar operations but adds a dimension since we have a partial derivative for each *pair* of actions. This extra dimension is the main reason gradient descent runs slower than replicator dynamics in our experiments.

4 VALIDATION AND EXPERIMENTS

4.1 Size Limits

The first constraint on the size of the data structures comes from integer overflows when calculating repetitions. Table 1 shows the upper limit in terms of P and A beyond which at least one profile will overflow a signed 64-bit integer. For example, if we want to represent a 5-action game with more than 32 players, we must use the log-transform to ensure that we calculate probabilities correctly. Note that using unsigned integers usually makes no difference and never allows more than one extra player.

This raises the question of whether we sacrifice precision when using the log-transform. To test this, we generated 10 random bipartite action-graph games for every combination of $2 \leq P \leq 512$ and $2 \leq A \leq 20$ for which the size of our data structures would be below 1GiB. We then represented these games using the CPU (64-bit) and GPU (32-bit) versions of our data structure, and calculated deviation payoffs for 1000 mixtures per game and recorded the largest error in the calculated deviation payoffs. These errors were tiny, except in games with hundreds of players in the 32-bit representation, and even then the scale of the errors was at least 7 orders of magnitude smaller than the payoff range of the game. Full results are shown in the Supplementary Figures Appendix.[7]

With these results, the question of what games we can represent comes down to what can fit in (main or GPU) memory. Figure 3 shows the size of the GPU data structure as P increases for various values of A. A similar figure for 64-bit arrays appears in Appendix D.[7] Note that calculating deviation payoffs requires storing intermediate arrays of similar size to the data structure, so game

---

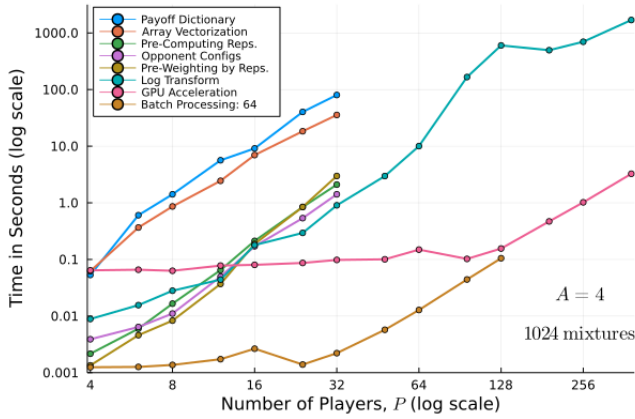[7]See Appendix D: supplementary figures. https://arxiv.org/abs/2302.13232

Figure 4: Time to required to compute deviation payoffs for 1024 mixtures in 4-action games using each data structure variant. Lines stop when either an integer overflow is encountered or when more than 1GB of memory is required. Similar plots for $A = 6$ and $A = 8$ appear in Appendix D.
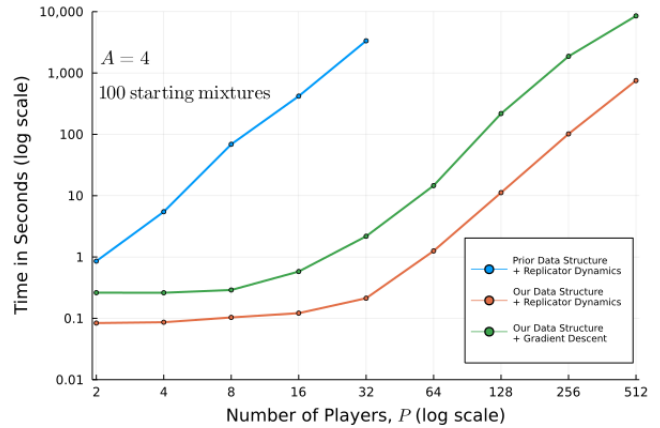

Figure 5: Time to compute Nash equilibria in 4-action games with 100 starting mixtures and 1000 iterations. Replicator dynamics is considerably faster than gradient descent (and could get away with fewer iterations), but it is often worthwhile to run both. Our data structure lets us represent and solve much larger games than the previous state-of-the-art. Similar plots for $A = 6$ and $A = 8$ appear in Appendix D.

size should be restricted to below half of available memory (or smaller if we want to calculate deviation payoffs in batches). Note that these sizes mean that it is entirely reasonable to store a 100-player, 6-action game.

4.2 Timing Comparisons

Figures 4 and 5 show our main results: our data structures produce significant speedup in computing deviation payoffs and therefore running Nash-finding algorithms. Timing comparisons were run using an AMD Threadripper 2990WX CPU and NVIDIA RTX 2080-Ti GPUs. Figure 4 shows the time to compute deviation payoffs using each of the data structures described in the previous section. The first several lines end at $P = 32$, since this is near the int-overflow limit for $A = 4$. The batch-processing line ends at $P = 128$, because operating on a batch of 64 mixtures requires much more memory. Note that our proposed data structure can compute deviation payoffs in a game with 384 players faster than the baseline data structure can handle $P = 12$. Many more deviation-payoff timing comparisons including 6- and 8-action games appear in Appendix D.[7]

Figure 5 shows a similar result for Nash-finding, but focusing only on our best data structure, and using batch sizes adapted to available memory. First, note that replicator dynamics using our data structure outperforms the previous implementation by four orders of magnitude. Gradient descent is consistently an order of magnitude slower than replicator dynamics, making it impractical and not generally implemented with older data structures. But with our data structure that slowdown can often be acceptable because gradient descent and replicator dynamics frequently identify some distinct equilibria (as illustrated in Figure 1), and so in cases where replicator dynamics fails or more equilibria are desired it is entirely reasonable to run both algorithms. For example, with $A = 4$ and

$P = 512$, we can perform 100 runs of replicator dynamics or gradient descent in a reasonable time-frame. Timing results shown in Appendix D for $A = 6$ and $A = 8$ are broadly similar.[7]

5 EXTENSIONS

5.1 Many Strategies

The size of our data structures and therefore the time to compute deviation payoffs scales reasonably similarly with respect to the number of players P and the number of actions A. The configurations and payoffs arrays both have size $A \times \binom{P+A-2}{P-1}$, which is dominated by the binomial that grows exponentially in the smaller of P and A. As a result, it is generally possible to represent games with a large number of players but a small number of actions or a large number of actions but a small number of players, with the caveat that actions contribute an extra linear factor.

Unfortunately, equilibrium computation does not scale equally well in players and actions. Identifying symmetric mixed-strategy Nash equilibria means searching for specific points in the mixed-strategy simplex, whose dimension grows with the number of actions. The various complete algorithms are generally hopelessly slow with large numbers of actions, and the local search methods we prefer are much less likely to identify an equilibrium on any given run. This means that solving symmetric games with many players is far more feasible than solving ones with many actions.

To handle games with many actions, we can take inspiration from Porter et al. [20], and search for equilibria with small support sets. This requires solving a subgame defined by restricting the game to a subset of the actions, and then checking whether those subgame solutions have beneficial deviations in the full action set. To facilitate this, we can use a variant of our data structure where we include additional rows in the payoffs-array representing actions outside the support set, but make no change to the

configurations-array. Then for a mixture in the support set, we can compute deviation payoffs for all actions against that mixture by expanding the step where we broadcast the multiplication of configuration probabilities and payoffs. If we determine that the current subgame set does not contain a Nash equilibrium, we can update the support set replacing weaker strategies with stronger ones, as measured by deviation gain against the subgame's candidate solutions (or other metrics). This will iteratively search the space of supports and eventually identify a small-support equilibrium if any exist. This approach of constructing complete subgames with extra payoff information for unilateral deviations is already widely used in empirical game settings [29] where determining payoffs for a single configuration is expensive.

5.2 Role-Symmetric Games

Outside of fully-symmetric games, many large normal-form games exhibit role-symmetry, where players can be grouped into R roles—like buyers and sellers or attackers and defenders—and within each role, all players are indistinguishable. To solve role symmetric games, analysts typically search for role-symmetric Nash equilibria, where all players in a given role play the same mixed strategy. Role symmetric mixed-strategy profiles can be represented by a vector that concatenates the mixed strategy played by each role, and the deviation payoff vector of the same dimension gives the expected utility of a unilateral deviator selecting each action. These deviation payoffs and their derivatives remain central to computing role-symmetric equilibria, and almost all of the techniques we propose for improving representations of symmetric games also apply under role symmetry.

Unfortunately, representing opponent configurations gets trickier with more than one role, because the set of opponents is different for players belonging to different roles. For example, in a game with 10 buyers and 10 sellers, when computing deviation payoffs for buyer-actions, opponent configurations include 9 buyers and 10 sellers, whereas for seller actions, there are 10 buyer-opponents and 9 seller-opponents. We could resolve this by building separate opponent-configuration tables for each role r, where the arrays for role r are based on configurations with $P_r - 1$ opponents in the same role and $P_{r'}$ opponents in other roles. This gives us R pairs of configuration- and payoff-arrays where the arrays for role r have $\binom{P_r + A_r - 2}{P_r - 1} \left(\prod_{r' \neq r} \binom{P_{r'} + A_{r'} - 1}{P_{r'}} \right)$ columns, by $\left(\sum_{r=1}^{R} A_r \right)$ rows for configurations and A_r rows for payoffs.

We could also dispense with the opponent-configuration approach and instead store full P-player profiles. This results in one profile-array and one payoff-array that each have size $\left(\sum_{r=1}^{R} A_r \right) \times \left(\prod_{r=1}^{R} \binom{P_r + A_r - 1}{P_r} \right)$, but requires us to return to masking-based deviation-payoff computations. Thus for multi-role games, we have a choice between slightly smaller storage requirements for a profile-based data structure or slightly simpler computations for a configuration-based representation; the gameanalysis.py library[2] employs the former option. Under either of these approaches, all of our other optimizations can still apply, and as long as the number of roles is small both options provide for reasonably efficient deviation-payoff calculations.

5.3 Action-Graph Games

Action-graph games [14] represent certain games compactly by storing, for each node, a mapping from neighborhood configurations to payoffs. Each action (for any player) corresponds to a node in a graph, and an action's payoff depends only on the number of players choosing each of the adjacent actions. This means that action-graph games are role-symmetric, with any group of players who share an action set belonging to the same role; when all players have the same action set, action-graph games are symmetric.

In the symmetric case, we can extend our data structures to efficiently compute deviation payoffs in action-graph games with a pair of arrays for each action storing that action's neighborhood configurations and repetition-weighted payoffs. The key augmentation is to include, as part of the representation of an opponent-configuration, an extra "action" capturing the number of players choosing any action outside the neighborhood. Then when computing the deviation payoff for an action, we can sum the probabilities for each non-adjacent action to get the out-of-neighborhood probability. This means that all of our data structure improvements can be applied to symmetric AGG-$\emptyset$ games.

For role-symmetric AGG-$\emptyset$ games, our role-symmetric data structure variants could be applied after splitting up action nodes shared by more than one role. For action-graph games with contribution-independent function nodes, many of our data-structure improvements can be applied, but the representation of these games tends to be sufficiently compact that the potential gains may be small. The gameanalysis.jl library[1] implements symmetric action-graph games with a bipartite graph between actions and contribution-independent functions as one of the tools for generating interesting random symmetric game instances. This implementation makes use of vectorization, opponent configurations, pre-computed repetitions, and the log transformation.

5.4 Multi-Threading

The next extension we would like to explore would focus on improving the CPU version of our data structure to capture some of the parallelism achievable on the GPU. At present, our deviation-payoff calculations are single-threaded, and while we might not expect to outperform SIMD array operations, there should still be room for significant speedup on multi-core systems from parallelizing a batch of deviation payoff computations across threads, especially for games in the multi-gigabyte size-range that can stress GPU memory but still fit in system RAM.

6 CONCLUSION

Our validation experiments show that the data structures we propose are capable of storing (Section 4.1) and efficiently computing (Section 4.2) deviation payoffs in symmetric games with very large numbers of players. By using incomplete-but-effective search algorithms we are consistently able to identify symmetric Nash equilibria. In combination with the iterative exploration approach described in Section 5.1 we can also find small-support equilibria in games with many actions. These results dramatically out-class existing solvers, enabling analysis of a much wider range of symmetric games, and closing the gap between micro- and macro-scale symmetric games.

REFERENCES

[1] Ben Blum, Daphne Koller, and Christian Shelton. 2006. Game Theory: GameTracer. http://dags.stanford.edu/Games/gametracer.html. Accessed: 2022-08-08.

[2] Enric Boix-Adserà, Benjamin L Edelman, and Siddhartha Jayanti. 2020. The multiplayer colonel blotto game. In *Proceedings of the 21st ACM Conference on Economics and Computation*. 47–48.

[3] George W. Brown. 1951. Iterative solution of games by fictitious play. *Activity analysis of production and allocation* 13, 1 (1951), 374–376.

[4] Peter E Caines. 2021. Mean field games. In *Encyclopedia of Systems and Control*. Springer, 1197–1202.

[5] Ben-Alexander Cassell and Michael P Wellman. 2012. EGTAOnline: An experiment manager for simulation-based game studies. In *International Workshop on Multi-Agent Systems and Agent-Based Simulation*. Springer, 85–100.

[6] Shih-Fen Cheng, Daniel M Reeves, Yevgeniy Vorobeychik, and Michael P Wellman. 2004. Notes on equilibria in symmetric games. In *AAMAS-04 Workshop on Game-Theoretic and Decision-Theoretic Agents*. GTDT.

[7] Vincent Conitzer and Tuomas Sandholm. 2002. Complexity results about Nash equilibria. *arXiv preprint cs/0205074* (2002).

[8] Constantinos Daskalakis, Paul W Goldberg, and Christos H Papadimitriou. 2009. The complexity of computing a Nash equilibrium. *SIAM J. Comput.* 39, 1 (2009), 195–259.

[9] Sevan G. Ficici, David C. Parkes, and Avi Pfeffer. 2008. Learning and Solving Many-Player Games through a Cluster-Based Representation *(UAI'08)*. AUAI Press, Arlington, Virginia, USA, 188–195.

[10] I Gemp, R Savani, M Lanctot, Y Bachrach, T Anthony, R Everett, A Tacchetti, T Eccles, and J Kramár. 2022. Sample-based Approximation of Nash in Large Many-Player Games via Gradient Descent. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, Vol. 1. 507–515.

[11] Srihari Govindan and Robert Wilson. 2003. A global Newton method to compute Nash equilibria. *Journal of Economic Theory* 110, 1 (2003), 65–86.

[12] Albert Xin Jiang, Hau Chan, and Kevin Leyton-Brown. 2017. Resource Graph Games: A Compact Representation for Games with Structured Strategy Spaces.. In *AAAI*. 572–578.

[13] Albert Xin Jiang and Kevin Leyton-Brown. 2011. Polynomial-time computation of exact correlated equilibrium in compact games. In *Proceedings of the 12th ACM conference on Electronic commerce*. 119–126.

[14] Albert Xin Jiang, Kevin Leyton-Brown, and Navin a R Bhat. 2011. Action-Graph Games. *Games and Economic Behavior* 71, 1 (2011), 141–173.

[15] Kevin Leyton-Brown and Yoav Shoham. 2008. Essentials of game theory: A concise multidisciplinary introduction. *Synthesis lectures on artificial intelligence and machine learning* 2, 1 (2008), 1–88.

[16] Zun Li and Michael Wellman. 2020. Structure Learning for Approximate Solution of Many-Player Games. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 02 (Apr. 2020), 2119–2127.

[17] Nicholas A Loehr. 2017. *Combinatorics: Discrete mathematics and its applications* (second ed.). Chapman and Hall/CRC.

[18] Richard D. McKelvey, Andrew M. McLennan, and Theodore L. Turocy. 2006. *Gambit: Software Tools for Game Theory*. Technical Report. Version 0.2006.01.20. http://econweb.tamu.edu/gambit/

[19] John Nash. 1951. Non-cooperative games. *Annals of Mathematics* (1951), 286–295.

[20] Ryan Porter, Eugene Nudelman, and Yoav Shoham. 2008. Simple search methods for finding a Nash equilibrium. *Games and Economic Behavior* 63, 2 (2008), 642–662.

[21] quantecon.org 2022. QuantEcon. https://quantecon.org/. Accessed: 2022-08-08.

[22] Herbert Scarf. 1967. The approximation of fixed points of a continuous mapping. *SIAM J. Appl. Math.* 15, 5 (1967), 1328–1343.

[23] Martin D Schatz, Tze Meng Low, Robert A van de Geijn, and Tamara G Kolda. 2014. Exploiting symmetry in tensors for high performance: Multiplication with symmetric tensors. *SIAM Journal on Scientific Computing* 36, 5 (2014), C453–C479.

[24] Yoav Shoham and Kevin Leyton-Brown. 2008. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press.

[25] Sam Sokota, Caleb Ho, and Bryce Wiedenbeck. 2019. Learning deviation payoffs in simulation-based games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 2173–2180.

[26] Peter D. Taylor and Leo B. Jonker. 1978. Evolutionary stable strategies and game dynamics. *Mathematical biosciences* 40, 1-2 (1978), 145–156.

[27] Weiran Wang and Miguel A Carreira-Perpinán. 2013. Projection onto the probability simplex: An efficient algorithm with a simple proof, and an application. *arXiv preprint arXiv:1309.1541* (2013).

[28] Xintong Wang, Christopher Hoang, Yevgeniy Vorobeychik, and Michael P Wellman. 2021. Spoofing the limit order book: A strategic agent-based analysis. *Games* 12, 2 (2021), 46.

[29] Michael P Wellman. 2020. Economic Reasoning from Simulation-Based Game Models. *Œconomia. History, Methodology, Philosophy* 2, 10 (2020), 257–278.

[30] Bryce Wiedenbeck and Michael P Wellman. 2012. Scaling simulation-based game analysis through deviation-preserving reduction.. In *AAMAS*. 931–938.