

Simple Search Methods for Finding a Nash Equilibrium

Ryan Porter and Eugene Nudelman and Yoav Shoham

Stanford University

{rwporter,eugnud,shoham}@stanford.edu

Abstract

We present two simple search methods for computing a sample Nash equilibrium in a normal-form game: one for 2-player games and one for n -player games. We test these algorithms on many classes of games, and show that they perform well against the state of the art—the Lemke-Howson algorithm for 2-player games, and Simplicial Subdivision and Govindan-Wilson for n -player games.

Introduction

Game theory has had a profound impact on multi-agent systems research, and indeed on computer science in general. The Nash equilibrium (NE) is arguably the most important concept in game theory, and yet remarkably little is known about the problem of computing a sample NE in a normal-form game. All evidence points to this being a hard problem, but its precise complexity is unknown (Papadimitriou 2001).

At the same time, several algorithms have been proposed over the years for the problem. In this paper, three previous algorithms will be of particular interest. For 2-player games, the Lemke-Howson algorithm (Lemke & Howson 1964) is still the state of the art, despite being 40 years old. For n -player normal form games, until recently the algorithm based on Simplicial Subdivision (van der Laan, Talman, & van der Heyden 1987) was the state of the art. Indeed, these two path-following algorithms are the default ones implemented in GAMBIT (McKelvey, McLennan, & Turocy), the best-known game theory software. Recently, Govindan and Wilson introduced an alternative, continuation-based method (Govindan & Wilson 2003). This algorithm, which we will refer to as Govindan-Wilson was extended, efficiently implemented, and tested by (Blum, Shelton, & Koller 2003).

In a long version of this paper we provide more intuition behind each these methods. Here we simply note that they have surfaced as the most competitive algorithms for the respective class of games, and refer the reader to two thorough surveys on the topic (von Stengel 2002; McKelvey & McLennan 1996). Our goal in this paper is to demonstrate that for both of these two classes of games – 2-player, and n -player for $n > 2$ – there exists a relatively sim-

ple, search-based method that performs very well in practice. For 2-player games our algorithm performs substantially better than Lemke-Howson. For n -player games our algorithm is better than Simplicial Subdivision; the comparison to the Govindan-Wilson algorithm is less conclusive. It is striking, however, that a relatively simple algorithm is not completely dominated by a such a sophisticated algorithm that is based on deep insight into the structure of games.

The basic idea behind our search algorithms is simple. Recall that, while the general problem of computing a NE is hard, computing whether there exists a NE with a *particular support* for each player¹ is easy (it simply requires solving a feasibility program, whereas the full problem is a complementarity program).

Each algorithms is equipped with a separate means for exploring the space of supports in a more efficient manner. Our algorithm for 2-player games instantiates the supports for each player separately, and prunes the search space by checking for actions in a support that are strictly dominated, given that the other agent will only play actions in its own support. Alternatively, the algorithm for n -player games considers full supports, but uses an easily-computed heuristic to first explore supports that are more promising according to this heuristic.

Common to both algorithms is the precedence they give to supports of small size. Since it turns out that games drawn from classes that researchers have focused on in the past tend to have (at least one) NE with a very small support, our algorithms are often able to find one quickly. Thus, this paper is as much about the properties of NE in games of interest as it is about novel algorithmic insights.

We emphasize, however, that we are not cheating in the selection of games on which we test. Past algorithms were tested almost exclusively on “random” games. We tested on these too (indeed, we will have more to say about how “random” games vary along at least one important dimension), but also on many other distributions (24 in total). To this end we use GAMUT, the recently introduced computational testbed for game theory.² Our results are quite robust across

¹The support consists of the pure strategies played with nonzero probability.

²To preserve author anonymity, we suppress the citation, which will appear in the final version of the paper.

all games tested.

The rest of the paper is organized as follows. After formulating the problem and the basis for searching over supports, we describe our two search algorithms. Then, we describe our experimental setup, and separately present our results for 2-player and n -player games. In the final section, we conclude and describe opportunities for future work.

Notation

In this section we formally define the necessary notation. We consider finite, n -player, normal-form games $G = \langle N, (A_i), (u_i) \rangle$:

- $N = \{1, \dots, n\}$ is the set of players.
- $A_i = \{a_{i1}, \dots, a_{im_i}\}$ is the set of actions available to player i , where m_i is the number of available actions for that player. We will use a_i as a variable that takes on the value of a particular action a_{ij} of player i , and $a = (a_1, \dots, a_n)$ to denote a profile of actions, one for each player. Also, let $a_{-i} = (a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$ denote this same profile excluding the action of player i , so that (a_i, a_{-i}) forms a complete profile of actions. We will use similar notation for any profile that contains an element for each player.
- $u_i : A_1 \times \dots \times A_n \rightarrow \mathfrak{R}$ is the utility function for each player i . It maps a profile of actions to a value.

Each player i selects a mixed strategy from the set $\mathcal{P}_i = \{p_i : A_i \rightarrow [0, 1] \mid \sum_{a_i \in A_i} p_i(a_i) = 1\}$. A mixed strategy for a player specifies the probability distribution used to select the action that the player will play in the game. We will sometimes use an action a_i in place of a mixed strategy p_i , which will denote the pure strategy in which $p_i(a_i) = 1$. The support of a mixed strategy s_i is the set of all actions $a_i \in A_i$ such that $p_i(a_i) > 0$. We will use $x = (x_1, \dots, x_n)$ to denote a profile of support sizes.

Because agents use mixed strategies, u_i is extended to also denote the expected utility for player i for a strategy profile $p = (p_1, \dots, p_n)$: $u_i(p) = \sum_{a \in A} p(a) u_i(a)$, where $p(a) = \prod_{i \in N} p_i(a_i)$.

Definition 1 A strategy profile $p^* \in \mathcal{P}$ is a Nash equilibrium if:

$$\forall i \in N, a_i \in A_i, \quad u_i(a_i, p_{-i}^*) \leq u_i(p_i^*, p_{-i}^*)$$

Searching Over Supports

The basis of our two algorithms is to consider space of instantiations of the support $S_i \subseteq A_i$ for each player i . A mixed strategy profile p consistent with S is then a equilibrium if each player is indifferent between all actions within his support, and does not prefer an action outside of his support. Figure 1 gives the formal description of the feasibility program for finding such an equilibrium p (if it exists).

In Feasibility Program 1, c_i corresponds to the expected utility of player i in an equilibrium. Because $p(a_{-i}) = \prod_{j \neq i} p_j(a_j)$, this program is linear for $n = 2$ and nonlinear for all $n > 2$. Note that, strictly speaking, we do not require that each action $a_i \in S_i$ be in the support, because it is allowed to be played with zero probability. However, the

$$\begin{aligned} \text{variables : } & \forall i, a_i \in S_i, \quad p_i(a_i) \\ & \forall i, \quad c_i \\ \text{constraints : } & \forall i, a_i \in S_i, \quad \sum_{a_{-i} \in S_{-i}} p(a_{-i}) u_i(a_i, a_{-i}) = c_i \\ & \forall i, a_i \notin S_i, \quad \sum_{a_{-i} \in S_{-i}} p(a_{-i}) u_i(a_i, a_{-i}) \leq c_i \\ & \forall i, \quad \sum_{a_i \in S_i} p_i(a_i) = 1 \\ & \forall i, a_i \in S_i, \quad p_i(a_i) \geq 0 \end{aligned}$$

Figure 1: Feasibility Program 1

point is that player i must still be indifferent between action a_i and all other actions $a'_i \in S_i$.

Algorithm for Two-Player Games

Our first algorithm exhaustively searches the space of supports. It orders the search space by considering every possible support size profile separately, favoring support sizes that are balanced and small. For a particular support size profile, it considers all possible instantiations of both player's support, making use of what we will call "conditional (strict) dominance" to prune the search space. Informally, an action is conditionally dominated (with the condition being a restricted set of available actions for each agent) if there exists another action for that player which always yields a higher utility, given that the remaining agents will only play actions within their respective restricted sets of available actions. When all other agents' supports are instantiated, this constraint is formally expressed as: $\nexists a_i \in S_i, a'_i \in A_i, a_{-i} \in S_{-i}, u_i(a_i, a_{-i}) < u_i(a'_i, a_{-i})$. Pseudocode for Algorithm 1 is as follows.

Algorithm 1

for all support size profiles $x = (x_1, x_2)$, sorted in increasing order of, first, $|x_1 - x_2|$ and, second, $\min(x_1, x_2)$ **do**
for all $S_1 \in 2^{A_1}$ s.t. $|S_1| = x_1$ **do**
 $A'_2 \leftarrow \{a_2 \in A_2 \text{ not conditionally dominated}\}$
if $\nexists a_1 \in S_1$ that is conditionally dominated **then**
for all $S_2 \in 2^{A'_2}$ s.t. $|S_2| = x_2$ **do**
if $\nexists a_1 \in S_1$ that is conditionally dominated **then**
if $\exists (p, c)$ that satisfies Feasibility Program 1 for (S_1, S_2) **then**
Return the found equilibrium profile p

This algorithm can be interpreted as solving a constraint satisfaction problem in which the variables are the supports S_i , and the domain of each S_i is the set of supports of size x_i . The single constraint is that there must exist a solution to Feasibility Program 1. However, it is useful to add the redundant constraints that no agent plays a conditionally dominated action when all agents are restricted to their respective supports. Algorithm 1 is then an instance of the

general backtracking algorithm, and the removal of conditionally strictly dominated strategies is similar to using the AC-1 to enforce arc-consistency with respect to the redundant constraints.

The keys to this algorithm are the ordering of the support size profiles and the removal of dominated strategies, which often provides a significant speedup. For example, after instantiating a support of size two for the first player, it is often the case that many of the second player’s actions are pruned, because only two inequalities must hold for one action to dominate another. Then, we only need to execute Feasibility Program 1 a relatively small number of times each support of the first player.

Algorithm for N-Player Games

While Algorithm 1 is able to benefit from a large amount of pruning in two-player games, and it can be easily generalized to the n -player case based on the CSP formulation, the benefits from pruning tend to decrease as the number of players increases. Consider, for example, a 6-player game for which we have instantiated supports of size 2 for 5 of the players. In general, there will be a relatively small amount of pruning of actions for the remaining player, since 32 inequalities must be satisfied for one action to dominate another. For this reason, we introduce a second search method (see the pseudo-code for Algorithm 2 below) that, for each support size profile, uses a heuristic to order the examination of support profiles.

The heuristic is based on the function $rank_i(a)$, which returns the order statistic for $u_i(a_i, a_{-i})$ out of the set of all possible utilities $u_i(a'_i, a_{-i})$ that player 1 can achieve by choosing a pure strategy, given that the other agents plays a_{-i} . This function is used in the following scoring function for a support: $score(S) = \sum_{a \in S} \sum_i rank_i(a)$, which captures the intuition that we should favor support profiles that give rise to action profiles in which each agent is playing an action that is close to “optimal” for it.

Algorithm 2 also prefers support size profiles $x = (x_1, \dots, x_n)$ which are closer to uniform, and we capture this preference by sorting the profiles according to the difference function: $diff(x) = \sum_i (x_i - \min_j x_j)$. However, instead of next sorting by $\min_i x_i$, it uses a support size profile x in which at least one $x_i = 0$ as a starting point, and then explores all support size profiles obtained by successively incrementing each x_i by one. For example, for a 3-player, 3-action game, the algorithm begins by using $x = (0, 0, 0)$ as a basis, from which it moves on to $(1, 1, 1)$ and then $(2, 2, 2)$.

For each support size profile encountered in such a path, a queue of supports is created, sorted in increasing order of $score(S)$. Each element of the queue is then expanded, and its children are all of the supports created by the addition of an action profile. Before placing a child in the next queue, we first check that it is not already present, due to the different orders in which a support profile can be created. If it is not, then we check that no action is dominated, before finally calling our nonlinear solver to attempt to solve Feasibility Program 1. Continuing with the example above, there exists a single (null) support consistent with $x = (0, 0, 0)$. It is then expanded, and all 27 of its children are considered. If

none of these supports satisfy Feasibility Program 1, then we move onto the support size profile $x = (2, 2, 2)$. The support profile of size 1 that had the lowest score is expanded first, and it has 8 children (two choices of an action to add for each agent). After considering these children, we advance to the support profile of size 1 with the second lowest score.

The use of sorted queues provides somewhat useful, but often insufficient, guidance to the search, due to the fact that supports of size k are often scored based on substantially fewer action profiles than supports of size $k + 1$. For this reason, we iterate through each queue multiple times, making use of cutoffs to restrict the search. On each iteration, we only consider child supports whose score lies below the cutoff for the current iteration, and above the cutoff of the previous iteration. To preserve completeness, the cutoff for the final pass was set ∞ , and the cutoff for the pass preceding the first pass was defined to be 0. We created 3 intermediate cutoffs by sampling and scoring a large number of supports consistent with this support size profile in the current game, and setting the cutoffs to be the scores that were less than that of all but 0.1%, 1% and 10% of the samples.

Algorithm 2

```

for all support size profiles  $x$  s.t.  $(\exists x_i, x_i = 0)$ , sorted by
increasing order of  $diff(x)$  do
   $queue[0] \leftarrow \emptyset$ 
  for all  $S$  s.t.  $\forall i, |S_i| = x_i$  do
    Insert  $S$  into  $queue[0]$ 
  for  $y = 1$  to  $\min_i (m_i - x_i)$  do
    Create  $cutoff[c]$  for  $0 \leq c \leq numCutoffs$ 
    for  $c = 1$  to  $numCutoffs$  do
       $queue[y] \leftarrow \emptyset$ 
      while  $S \leftarrow NextElement(queue[y - 1])$  do
        for all  $a \in A$  s.t.  $\forall i, a_i \notin S_i$  do
           $S' \leftarrow (S_1 \cup \{a_1\}, \dots, S_n \cup \{a_n\})$ 
          if  $(score(S') \leq cutoff[c]) \wedge$ 
 $(score(S') > cutoff[c - 1])$  then
            if  $S' \notin queue[y + 1]$  then
              Insert  $S'$  into  $queue[y + 1]$ 
              if  $\nexists i, a_i \in S_i$  s.t.  $a_i$  is strictly condition-
ally dominated then
                if  $\exists(p, c)$  that satisfies Feasibility Pro-
gram 1 for  $S'$  then
                  return the found equilibrium  $p$ 

```

Experimental Results

To evaluate the performance of our algorithms we ran several sets of experiments. All games were generated by GAMUT, a test-suite that is capable of generating games from a wide variety of classes of games found in the literature. Table 1 provides a brief description of the distributions on which we tested.

A distribution of particular importance is the one most commonly tested on in previous work: D18, the “Uniformly Random Game”, in which every payoff in the game is drawn independently from an identical uniform distribution. Also important are distributions D5, D6, and D7, which fall under

D1	Bertrand Oligopoly	D2	Bidirectional LEG, Complete Graph
D3	Bidirectional LEG, Random Graph	D4	Bidirectional LEG Star Graph
D5	Random; payoff correlation 0.9	D6	Random; correlation uniform in $[-1, 1]$
D7	Random, no correlation	D8	Dispersion Game
D9	Graphical Game, Random Graph	D10	Graphical Game, Road Graph
D11	Graphical Game, Star Graph	D12	Graphical Game, Small-World
D13	Minimum Effort Game	D14	Polymatrix Game, Complete Graph
D15	Polymatrix Game, Random Graph	D16	Polymatrix Game, Road Graph
D17	PolymatrixGame, Small-World Graph	D18	Uniformly Random Game
D19	Travelers Dilemma	D20	Uniform LEG, Complete Graph
D21	Uniform LEG, Random Graph	D22	Uniform LEG, Star Graph
D23	Location Game	D24	War Of Attrition

Table 1: Descriptions of GAMUT distributions.

a model studied by (Rinott & Scarsini 2000) (which we will refer to as a ‘‘Covariant Game’’), in which the payoffs for the n agents for each action profile are drawn from a multivariate normal distribution in which the covariance ρ between the payoffs of each pair of agents is identical. When $\rho = 1$, the game is common-payoff, while $\rho = \frac{-1}{N-1}$ yields minimal correlation, which occurs in zero-sum games. Thus, by altering ρ , we can smoothly transition between these two extreme classes of games.

Our experiments were executed on a cluster of 12 dual-processor, 2.4GHz Pentium machines, running Linux 2.4.20. We capped runs for all algorithms at 1800 seconds. When $n = 2$, we solved Feasibility Program 1 using CPLEX 8.0’s callable library. For $n > 2$, because the program is nonlinear, we instead solved each instance of the program by executing AMPL, using MINOS as the underlying optimization package. Obviously, we could substitute in any nonlinear solver; and, since a large fraction of our running time is spent on AMPL and MINOS, doing so would greatly affect the overall running time.

Before presenting the empirical results, we note that an comparison of the worst-case running times of our two algorithms and the three we tested against does not distinguish between them, there exist inputs for each which lead to exponential time. It is important to note, though, that while Algorithm 1 is extremely space efficient, Algorithm 2 requires an exponential amount of memory to maintain the queues.

Results for Two-Player Games

In the first set of experiments, we compared the performance of Algorithm 1 to that of Lemke-Howson (implemented in Gambit, which added the preprocessing step of iterated removal of weakly dominated strategies) on 2-player 300-action games drawn from 24 different GAMUT distributions. Both algorithms were executed on 100 games drawn from each distribution. The time is measured in seconds and plotted on a logarithmic scale.

Figure 2(a) compares the median runtimes of the two algorithms, and show that Algorithm 1 performs better on all distributions.³ However, this does not tell the whole story. For many distributions, it simply reflects the fact that there is a greater than 50% chance that the distribution will generate

³Obviously, the lines connecting data points across distributions for a particular algorithm are meaningless— they were only added to make the graph easier to read.

a game with a pure strategy NE, which our algorithm will then find quickly. Two other important statistics are the percentage of instances solved (Figure 2(b)), and the average runtime conditional on solving the instance (Figure 2(c)). Here, we see that Algorithm 1 completes far more instances on several distributions, and solves fewer on just a single distribution (6 fewer, on D23). Additionally, even on distributions for which we solve far more games, our conditional average runtime is 1 to 2 orders of magnitude faster.

Clearly, the hardest distribution for our algorithm is D6, in which the covariance ρ is drawn uniformly at random from the range $[-1, 1]$. In fact, neither Algorithm 1 nor Lemke-Howson solved any of the games in another ‘‘Covariant Game’’ distribution in which $\rho = -0.9$, and these results were omitted from the graph, because the median and conditional average are undefined for these results. On the other hand, for the distribution ‘‘CovariantGame-Pos’’ (D5), in which $\rho = 0.9$, both algorithms perform well.

To further investigate this continuum, we sampled 300 values for ρ in the range $[-1, 1]$, with heavier sampling in the transition region and at zero. For each such game, we plotted a point for the runtime of both Algorithm 1 and Lemke-Howson in Figure 2(d). The theoretical results of (Rinott & Scarsini 2000) suggest that the games with lower covariance should be more difficult for Algorithm 1, because they are less likely to have a pure strategy Nash equilibrium.⁴ Nevertheless, it is interesting to note the sharpness of the transition that occurs in the $[-0.3, 0]$ interval. More surprisingly, a similarly sharp transition also occurs for Lemke-Howson, despite the fact that the two searches operate in completely unrelated ways. Finally, it is important to note that the transition region for Lemke-Howson is shifted to the right by approximately 0.3, and that there does not exist a value of ρ such that it performs better than Algorithm 1. Moreover, on instances in the easy region for both algorithms, Algorithm 1 is still an order of magnitude faster.

In the third set of experiments we explore the scaling behavior of both algorithms on the ‘‘Uniformly Random Game’’ distribution (D18), as the number of actions increases from 100 to 1000. For each multiple of 100, we generated 20 games. Because space constraints preclude an analysis similar to that of Figures 2(a) through 2(c), we instead plot in Figure 2(e) the *unconditional* average of runtimes over 20 instances for each data size, with a timeout counted as 1800s. While Lemke-Howson failed to solve any game with more than 600 actions and timed out on some 100-action games, Algorithm 1 solved all instances and, without the help of cutoff times, still had an advantage of 2 orders of magnitude at 1000 actions.

Results for N-Player Games

In the next set of experiments we compare Algorithm 2 to Govindan-Wilson and Simplicial Subdivision (also implemented in Gambit, and thus combined with iterated removal of weakly dominated strategies). First, to compare perfor-

⁴We presume, although it is not addressed in (Rinott & Scarsini 2000), that a lower covariance also causes the game to be less likely to have a NE of support size k , for small values of k

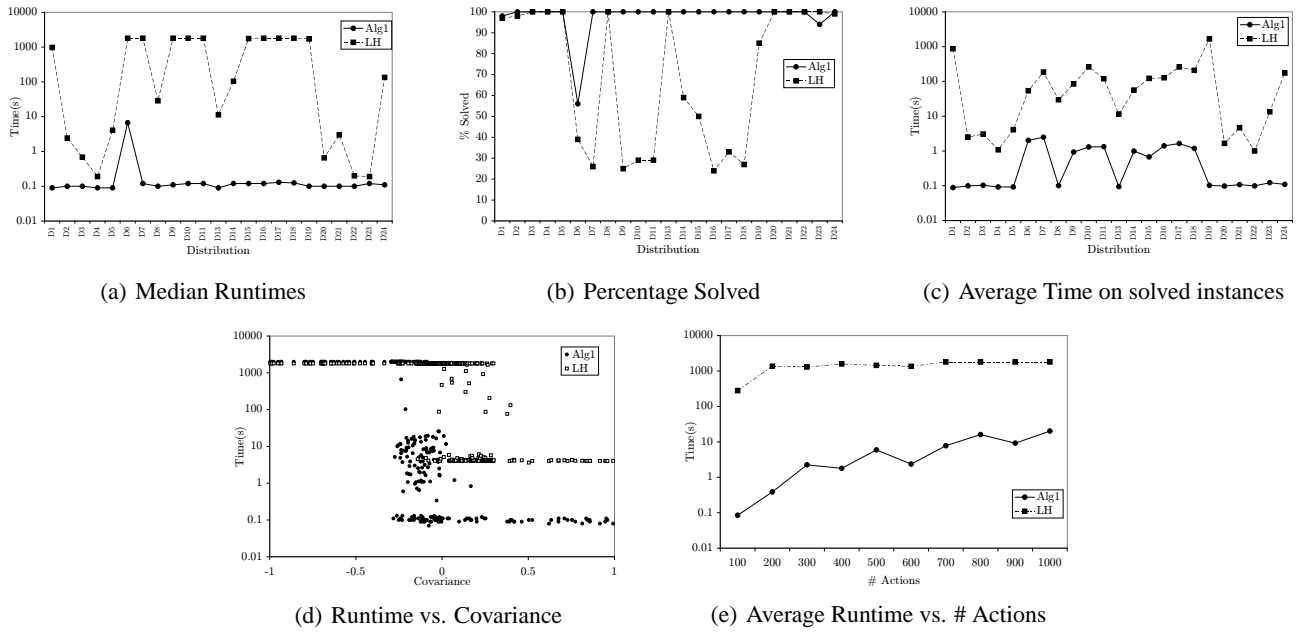


Figure 2: Comparison of Algorithm 1 and Lemke-Howson on 2-player games. Subfigures (a)-(d) are for 300-action games.

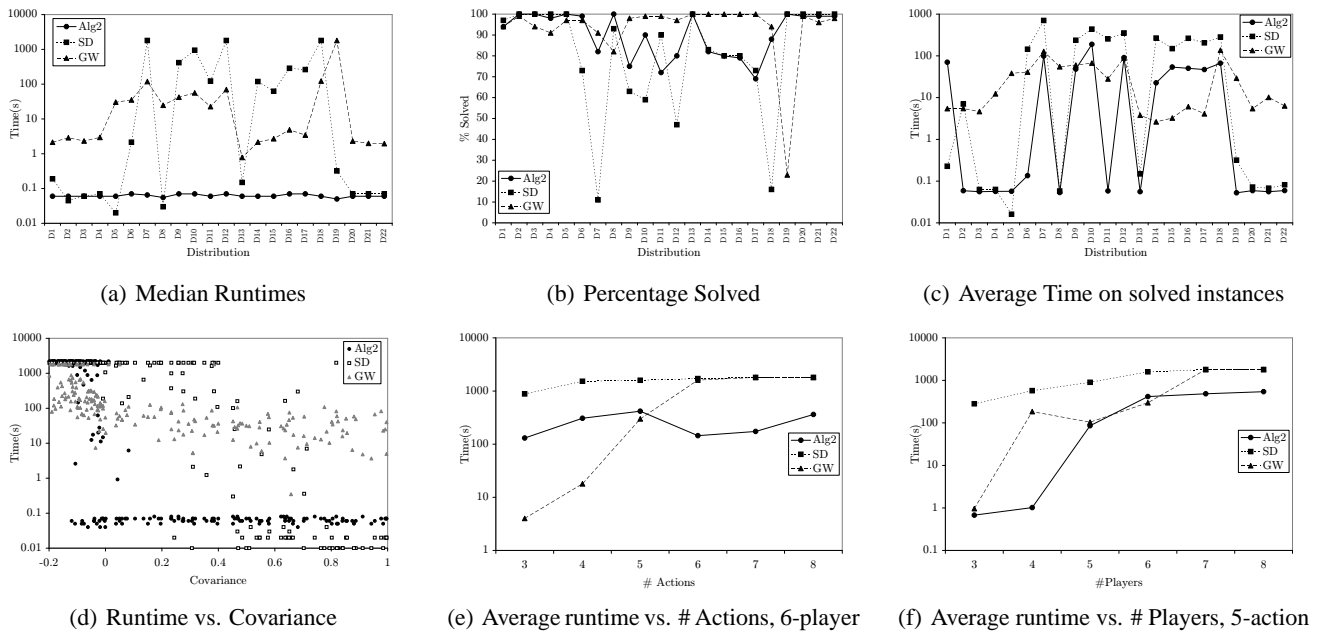


Figure 3: Comparison of Algorithm 1, Simplicial Subdivision, and Govindan-Wilson. Subfigures (a)-(d) are for 6-player, 5-action games.

mance on a fixed problem size we tested on 100 games from 22 GAMUT n -player distributions with 6 players and 5 actions each. While those numbers appear small, note that these games have 15625 outcomes and 93750 payoffs. Note that two distributions from the tests of 2-player games are missing here, due to the fact that they do not naturally generalize to more than 2 players.

Once again, Figures 3(a), 3(b), and 3(c) show median runtimes, number of completed instances, and conditional average runtime (on the instances solved before the cutoff), respectively. Algorithm 2 has a very low and consistent median runtime, for the same reason that Algorithm 1 did for two-player games. It solves more instances than Simplicial Subdivision, and has a superior conditional average runtime (with a correlation between the two algorithms that is striking). The comparison with Govindan-Wilson is much more ambiguous, since Algorithm 2 solves fewer total instances, but holds a significant advantage in conditional average runtime on more distributions than Govindan-Wilson does.

Since the results of (Rinott & Scarsini 2000) for the probability of the existence of a pure strategy NE also hold for n -player games, we again investigate the relationship between ρ and the hardness of games. For general n -player games, minimal correlation under the ‘‘Covariant Game’’ distribution occurs when $\rho = -\frac{1}{n-1}$. Thus, we can only study the range $[-0.2, 1]$ for 6-player games. Figure 3(d) shows the results for 6-player 5-action games. Algorithm 2, over the range $[-0.1, 0]$ experiences a transition in hardness that is even sharper than that of Algorithm 1. Simplicial Subdivision also undergoes a transition, which is not as sharp, that begins at a much larger value of ρ (around 0.4). On the other hand, the running time of Govindan-Wilson is only slightly affected by the covariance, as it neither suffers as much for small values of ρ nor benefits as much from large values.

Finally, Figures 3(e) and 3(f) show the scaling behavior of the three algorithms: the former holds the number of players constant at 6 and varies the number of actions from 3 to 8, while the latter holds the number of action constant at 5, and varies the number of players from 3 to 8. As before, for each (player,action) pair we sample 20 games from the ‘‘Uniformly Random Game’’ distribution, and plot unconditional averages. In both graphs, both Simplicial Subdivision and Govindan-Wilson solve no instances for the rightmost two sizes, while Algorithm 2 still solves a significant fraction. Also interesting is the fact that, over the range considered, our algorithm scales very well with number of actions, while Simplicial Subdivision transitions from being much faster than Algorithm 2 to being significantly slower.

Conclusion and Future Work

In this paper, we presented two search methods for finding a sample Nash equilibrium, both of which favored supports that were small and balanced. The first uses removal of dominated actions to efficiently solve 2-player games, and it strictly outperforms the Lemke-Howson algorithm. The second, which employs a heuristic to order supports of a particular size, compares well against Simplicial Subdivision. Against Govindan-Wilson, the results are more ambiguous,

but many favor our algorithm.

The most difficult games we encountered came from the covariant game model, as the covariance approaches its minimal value, and this is a natural target for future algorithm development. We expect these games to be hard in general, because, empirically, we found that as the covariance decreases, the number of equilibria decreases, and the equilibria that do exist are more likely to have support sizes near one half of the number of actions, which is the support size with the largest number of supports.

One direction for future work is to apply more sophisticated CSP techniques to the formulation described above. Another promising direction to explore is local search, in which the state space is the set of all possible supports, and the available moves are to add or delete an action from the support of a player. While the fact that no equilibrium exists for a particular support does not give any guidance as to which neighboring support to explore next, one could use the scoring metric defined for Algorithm 2 or a relaxation of Feasibility Program 1 that penalizes infeasibility through an objective function. More generally, our results show that AI techniques can be successfully applied to this problem, and we have only scratched the surface of possibilities along this direction.

References

- Blum, B.; Shelton, C. R.; and Koller, D. 2003. A continuation method for nash equilibria in structured games. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*.
- Govindan, S., and Wilson, R. 2003. A global newton method to compute nash equilibria. In *Journal of Economic Theory*.
- Lemke, C., and Howson, J. 1964. Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics* 12:413–423.
- McKelvey, R., and McLennan, A. 1996. Computation of equilibria in finite games. In H. Amman, D. Kendrick, J. R., ed., *Handbook of Computational Economics*, volume I. Elsevier. 87–142.
- McKelvey, R.; McLennan, A.; and Turocy, T. *Gambit: Software tools for game theory*. Available at <http://econweb.tamu.edu/gambit/>.
- Papadimitriou, C. 2001. Algorithms, games, and the internet. In *STOC-01*, 749–753.
- Rinott, Y., and Scarsini, M. 2000. On the number of pure strategy nash equilibria in random games. *Games and Economic Behavior* 33:274–293.
- van der Laan, G.; Talman, A.; 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*.
- von Stengel, B. 2002. Computing equilibria for two-person games. In Aumann, R., and Hart, S., eds., *Handbook of Game Theory*, volume 3. North-Holland. chapter 45, 1723–1759.