UNIVERSITY OF LIVERPOOL

DEPARTMENT OF COMPUTER SCIENCE

An Empirical Study on Computation of Exact and Approximate Equilibria

Thesis submitted in accordance with the requirements of
The University of Liverpool for
the degree of Doctor of Philosophy
by

Tobenna Peter, IGWE

January 2018

Abstract

The computation of Nash equilibria is one of the central topics in game theory, which has received much attention from a theoretical point of view. Studies have shown that the problem of finding a Nash equilibrium is PPAD-complete, which implies that we are unlikely to find a polynomial-time algorithm for this problem. Naturally, this has led to a line of work studying the complexity of finding approximate Nash equilibria. This thesis examines the computation of such approximate Nash equilibria within several classes of games from an empirical perspective.

In this thesis, we address the computation of approximate Nash equilibria in bimatrix and polymatrix games. For both of these game classes, we provide a library of implementations of algorithms for the computation of exact and approximate Nash equilibria, as well as a suite of game generators which were used as a base for our empirical analysis of the algorithms. We investigate the trade-off between quality of approximation produced by the algorithms and the expected runtime. We provide some insight into the inner workings of the state-of-the-art algorithm for computing ϵ-Nash equilibria, presenting worst-case examples found for our provided suite of game generators. We then show lower bounds on these algorithms. In the case of polymatrix games, we generate this lower bound from a real-world application of game theory. For bimatrix games, we provide a robust means of generating lower bounds for approximation algorithms with the use of genetic algorithms.

Acknowledgments

Special thanks to my supervisor, Rahul Savani, for his immense support over the years. Rahul first introduced me to game theory when I had just finished my first year as an undergraduate computer science student. From taking out of his time to teach me about game theory, giving me the opportunity to partake in research as a PhD student, to supporting me throughout the process, I cannot overstate how much help he has provided. For all the help provided, I am ever grateful.

I especially thank my second supervisor, Martin Gairing, and my academic advisors Paul Spirakis and Giorgos Christodoulou for their time, feedback, and advice given during my studies.

Many thanks to John Fearnley and Argyrios Deligkas, my collaborators on several research topics. Together with Rahul Savani, we tackled research questions and were my coauthors on published papers. The ideas they contributed and the knowledge which they shared are highly valued.

I also extend thanks to my friends, colleagues and officemates who were there throughout all these years, without whom my time as a PhD student would not have been so enjoyable.

Last but by no means least, special thanks to my parents and family, who have been a relentless provision of support and encouragement throughout.

Contents

Chapter 1

Introduction

In this thesis, we study the problem of computation of approximate Nash equilibria, more specifically, from an empirical perspective. This thesis is split into two main parts, the first of which is a comprehensive study on approximate Nash equilibria on both bimatrix and polymatrix games (consisting of Chapters 2 - 4). The second part (Chapter **??**) focuses on an analysis of Nash equilibria in symmetric combinatorial auctions where the bidders have a budget constraint. Before stating the contributions made, we give an overview of the problems and questions which we address.

1.1 Background

Non-cooperative game theory is one of the fields of mathematics which has recently been receiving an increased level of interest. It is a tool which is used to analyse situations whereby strategic decisions are being made by two or more rational and selfish individuals or agents trying to maximise their personal welfare. The concepts originating from the field of game theory have been studied theoretically not only within mathematical circles (such as Computer Science [22, 112]), but also within the social sciences (e.g. Economics [5, 89, 94, 110]), and even the physical sciences (especially Biology [45, 109]). Not only is it a topic with rich theoretical studies, but it also has a broad range of practical applications; from problems within artificial intelligence such as classification [49, 60] and clustering [57, 120], to issues within security, both cyber [74, 128] and physical [101]. Just based on the possible applications alone, one can easily understand why there is so much interest in game theory.

1.1.1 Games

Within the realm of game theory, a game refers to an interaction between two or more individuals, otherwise known as *players* in the game. One of the fundamental assumptions is the notion of rationality, which assumes that each player's objective is to maximise their personal welfare / happiness, usually measured using a *utility* or *payoff* function. This function maps the final outcome of a game, born out of the collective choices of actions performed by all players, to a numerical value representing their preference of the given outcome.

One of the most common yet simple representations of a game is a *normal form game*. A normal form game, sometimes known as the *strategic form representation* of a game, consists of a finite set of players each having a finite set of *pure strategies*. The game is played by all players choosing a single pure strategy simultaneously, and receiving a payoff based on the outcome resulting from the combination of strategies by all players. In general, every player is allowed to randomise their play based on a probability distribution over their set of pure strategies, defining a *mixed strategy* for the player.

In the base case, with two players, such a game can be represented using two matrices, (where it derives its name of *bimatrix games*) one for each player. The actions of the first player (or *row* player) are represented by the rows of the matrices, while the columns represent those of the second player (*column* player). Finally, each element of the matrices shows the payoff achieved by the player given the actions corresponding to the row and column of the given element.

When representing a normal form game one needs to show, for each possible combination of strategies, the payoff of each player in the game. This leads to the representation growing exponentially in the number of players. For example, for a game with n players who can each choose between 2 strategies, $n \cdot 2^n$ payoffs must be specified. However, many scenarios do not need the flexibility which the representation of normal form games provide. In particular, it is often the case that only the *pairwise* interactions between players are important. This is the issue which the use of polymatrix game representation tries to resolve.

A *Polymatrix game* is an n-player game which is defined by an n-vertex graph. Each vertex represents a player. Each edge e corresponds to a bimatrix game that will be played by the players that e connects. Hence a player with degree d plays d bimatrix games. More precisely, each player picks a strategy and plays that strategy in *all* of the bimatrix games that he is involved in. His expected payoff is given by the sum of the expected payoffs that

he obtains over all the bimatrix games that he participates in.

In practice, situations do occur where the players involved do not have a complete knowledge of the information pertaining to the other players of the game; such cases are modelled using *Bayesian games*. In a Bayesian game, each player has a private type (chosen from a publicly known joint distribution) which determines the action set of the given player as well as their utility function. Unlike the previously mentioned game types where each player submits a single strategy profile, in order to play a Bayesian game, each player must submit a mixed strategy profile for each one of their types.

1.1.2 Nash equilibria

Out of the solution concepts within game theory, one of the concepts which has received quite a lot of attention both from a theoretical and practical perspective is the notion of a *Nash equilibrium*. A Nash equilibrium is a strategy profile such that no player can unilaterally deviate from their chosen strategy and increase their expected payoff as a result, while keeping the strategies of the other players fixed. Another equivalent definition of a Nash equilibrium states that all strategies played with non-zero probability (also known as the *support* of a strategy) are best-responses to the strategy profile being played by the other players. This can also be referred to as a *mixed strategy Nash Equilibrium*. If all the players played a single one of their actions with probability of 1, it is referred to as a *pure strategy Nash equilibrium*.

As mentioned earlier, there are several practical applications of game theory, most of which make use of the concept of a Nash equilibrium. This has led to the problem of equilibrium computation receiving much attention from a theoretical point of view. One such method for finding equilibria of a game is the widely used Lemke-Howson algorithm (LH) [86], which finds at least one Nash equilibrium of a bimatrix game by complementarily pivoting on a pair of polytopes generated from the game. Although it is widely used, it is known to have, in the worst case, an exponential running time [116], and it is `PSPACE`-hard to compute the equilibrium that it finds [64].

A generalised version of the LH algorithm is the Lemke's algorithm for solving *linear complementarity problems* (LCP) [85]. Miller and Zucker have shown that finding a Nash equilibrium in a polymatrix game can be reduced in polynomial time to an LCP [92]. There are several other algorithms [66–68, 71, 105] which can be used to solve more general n-player normal form games. However, these algorithms have a running time which is exponential in the worst case scenario.

Although the existence of Nash equilibria in games was established in 1950 [95], only recently was the problem of computing Nash equilibria classified. More specifically, it was shown that the problem of finding a Nash equilibrium is PPAD-complete [22, 32, 99], even for games with only two players. While one of the most popular complexity classes, NP, captures decision problems, the complexity class PPAD captures problems which are reducable to the following problem:

- ANOTHER END OF THE LINE: Given a succinctly represented graph where each vertex has an indegree and outdegree of at most one with no isolated vertices and a vertex with indegree of zero, find a vertex with either an indegree of zero or an outdegree of zero.

It is generally assumed that it is unlikely for there to exist a polynomial time algorithm for PPAD-complete problems. Naturally, the difficulty involved in computing an exact Nash equilibrium has led to a line of work studying the complexity of finding *approximate* Nash equilibria.

In contrast to a mixed Nash equilibria, pure strategy equilibria are quite simple and easy to reason about as the players are not required to randomize. Analysis of pure Nash equilibria can also be easier to analyse due to its discrete nature [19, 113], although they are not guaranteed to exist in finite games unlike mixed Nash equilibria.

Certain classes of games for which a pure Nash equilibrium is guaranteed to exist have been studied [59, 93, 106]. These games are guaranteed to have a pure Nash equilibrium due to the existence of a global function (also known as a *potential function*) which can be used to express the incentives of players to change their strategies. For certain subclasses of games, such as congestion games with symmetric networks, algorithms for computing a pure Nash equilibria in polynomial time exist [52]. However, the problem of computing a pure Nash equilibrium in such potential games is known to be PLS-complete [52,75]. In the general case, deciding whether a game has a pure Nash equilibrium in NP-hard [27,65].

1.1.3 Approximate Nash equilibria

In most algorithmic studies of approximation algorithms to computational problems, a relative notion of approximation (i.e. a multiplicative factor to the optimal value of an objective function) is usually the standard approach. However, regarding Nash equilibria, primarily due to recent work showing computation of relative approximate equilibria to be PPAD-hard [31, 112], most of the work on computing approximate Nash equilibria has focused on additive approximations. There are two different notions of additive approximate

Nash equilibria which are being studied, both of which stem from the result of relaxing the two definitions of a Nash equilibrium.

By relaxing the first definition, we get the notion of an ϵ-*Nash equilibrium* and relaxing the second gives us the notion of an ϵ-*well supported Nash equilibrium*. An ϵ-*Nash equilibrium* (ϵ-NE) requires that both players achieve an expected payoff that is within ϵ of the best response. In other words, no player can unilaterally deviate from their current strategy and gain a payoff more than ϵ, whereas the stronger notion, an ϵ-*well supported Nash equilibrium* (ϵ-WSNE), requires that the players only place a non-zero probability on strategies that are within ϵ of the value achieved by a best response. For these notions of approximation to have a consistent meaning across all games, it is usually assumed that the utilities of the players are in the range $[0, 1]$, hence $\epsilon \in [0, 1]$.

Shortly after the hardness of computing a Nash equilibrium was confirmed, work on computing approximate Nash equilibria started, with the first results on ϵ-NE having a guarantee of 0.75 [78], 0.5 [34] and 0.38 [33] on bimatrix games. Following these algorithms, there have been several algorithms which make use of linear programming to get approximation guarantees of 0.38, 0.36 [17] and the current state of the art which is a descent algorithm that yields a guarantee of 0.3393 + δ [121], where $\delta > 0$. For polymatrix games, the bimatrix version of the descent algorithm was generalised where it yields an approximation guarantee of $0.5 + \delta$ [43], and a recent QPTAS for polymatrix games on trees [11].

With regards to ϵ-WSNE, the several algorithms available for this task make use of linear programming which gives an approximation of 0.6667 [80], 0.6608 [53] and 0.6528 [28] on bimatrix games. By the nature of its definition, any ϵ-WSNE is also an ϵ-NE. It has also been shown that for any given $\frac{\epsilon^2}{8}$-NE for a game, one can in polynomial time find an ϵ-WSNE [22].

A recent piece of research by Czumaj, Fasoulakis and Jurdziński [29] studied the concept of approximation algorithms from a slightly different perspective. Their algorithm makes use of similar concepts from several other algorithms by forming and solving several zero-sum games [17,28,53,80]. In contrast to these algorithms, their proposed method finds either a $\frac{1}{3}$-NE or a $\frac{1}{2}$-WSNE. Although this method does not guarantee which will be the result of its computation, it is able to produce either an ϵ-NE or ϵ-WSNE which, when computed, gives better results than the current state-of-the-art algorithms.

Although we have algorithms which have a constant approximation guarantee, the problem of computing an ϵ-NE has been shown to be PPAD-complete for games with three or more players [32]. This result was then extended to show PPAD-completeness

for games with two players [22]. The result implies that finding a fully polynomial-time approximation scheme (FPTAS) for computing an approximate Nash equilibrium is highly unlikely unless PPAD is in P. More recently, it was shown that when computing an ϵ-NE in an $n \times n$ bimatrix game, there exists a constant $\epsilon > 0$ which requires time $n^{\log^{1-o(1)} n}$ [112]. This result assumes the *Exponential Time Hypotheses* for PPAD [9], which conjectures that one of the fundamental problems END OF THE LINE requires $2^{\tilde{\Omega}(n)}$ time to solve.

Despite the hardness results for both exact and approximate Nash equilibrium, there exist several algorithms which take into consideration the structure and properties of certain classes of games, and run in polynomial time. One of the more popular examples of these algorithms is the use of linear programming to solve zero-sum bimatrix games (where for every outcome of the game, the payoffs of both players sum to zero), with this method being a fundamental part of some of the currently known approximation algorithms. Several other algorithms for bimatrix games include a polynomial-time approximation scheme (PTAS) for constant rank games [76], small-probability games [36], and others [10, 39, 79].

1.1.4 Item-Bidding Auctions

Amidst the wide variety of games within the field, one of the most comprehensively studied are auctions. Auctions have primarily served as a tool for allocating and selling goods and services, and with the dawn of the internet, they have become more accessible through sites like eBay and Google. In general, the problem which auctions tend to address is how to allocate a set of items to a group of individuals. One method of modelling such a problem is with the use of combinatorial auctions.

In a combinatorial auction, a set of bidders compete over how a set of items is to be allocated. Each player has a valuation function which maps from a subset of the items to a numerical value representing the value the player has for the given subset. Upon allocation, each player gains a utility which is the difference between one's valuation for the allocated subset and the price according to the pricing rules of the auction.

When formulating such auctions, the mechanism designer tends to aim for a mechanism which: maximizes the social welfare (the sum of utilities achieved by all players); is truthful (each player gains nothing by revealing a false valuation to the auctioneer); and is computationally-tractable. Each one of these goals pose a significant hurdle to the designer, especially welfare maximization which is known to be computationally intractable. The welfare maximization problem in general is NP-hard even to approximate [56].

Truthfulness is a desired property to have in a mechanism, especially when tied together with individual rationality (i.e. by bidding truthfully, the players are guaranteed not to have a negative payoff). Having these two properties, eases the decision process for the players involved, as there is a clear dominant strategy which they can always default. Having these properties within a mechanism allows for the designer to reason about the behaviour of the players involved. However, efficient computation and truthfulness have been found to be difficult to achieve. Even the renowned VCG mechanism [25, 69, 125] while being a truthful mechanism, maximizes the social welfare has been found to take an exponential amount of time [96, 97]. Despite the benefits provided by truthfulness, studies have shown that non-truthful auctions are capable of having equivalent equilibria [77, 124], as well as equilibria with more preferable properties [46] in comparison to the truthful outcomes of such mechanisms

In order to address some of the issues when using combinatorial auctions, a simpler and more practical format was introduced and has been studied widely [24, 56, 70]. In combinatorial auctions with item-bidding (sometimes referred to as simultaneous auctions), each player places a bid for each item, and each item is sold in an independent single-item auction. This simplicity results in the ease of communication between the bidders and auctioneer as the bidders only need to send over a single bid for each item, as opposed to their full valuation function, which is exponential in the number of items. These auctions are also computationally tractable as the auctioneer does not need to worry about running multiple single-item auctions. In exchange for this simplicity, the bidders are not capable of fully expressing their true valuations which kills all hope of having a truthful item-bidding auction.

1.2 Contributions

So far, most of the work on approximate equilibria has been theoretical in nature. This thesis aims to investigate, from an empirical perspective, the relevance of approximate Nash equilibria in practice. In this section, we address the contributions which we make in this thesis towards the problem at hand.

Implementations of Algorithms and Game Generators

Currently, the Gambit library [91] remains one of the largest open source libraries which provides implementations of several algorithms for solving normal-form games. Some

of the algorithms implemented include: a linear programming method for solving zero-sum two-player games; the global Newton method [66]; the simplicial subdivision algorithm [123], amongst others. Despite the collection of several algorithms, there is a lack of any implementation of algorithms for solving polymatrix games while maintaining their compact nature. This library also lacks any implementation of algorithms for computing approximate Nash equilibria.

When performing empirical analysis of algorithms, besides the implementation of the algorithms, another important portion is having a method to generate a wide range of inputs on which to evaluate the algorithms. To fulfil this role, a library known as *GAMUT* [98] is usually used in empirical studies of game theoretical algorithms [26, 61, 104, 114]. However, one of the short-comings of this library is its lack of compact representation of polymatrix games.

We address the aforementioned problems by making our implementations of several algorithms for computing approximate Nash equilibrium on both bimatrix [1] and polymatrix [2] games open-source. For both classes of games, we also provide a suite of game generators which can be used for benchmarking and testing algorithms related to computing Nash equilibria. Implementation details and documentation of the tools can be found in the Appendix.

Comprehensive Empirical Analysis of Algorithms

Making use of the implementations of algorithms and game generators which we make available, we perform an extensive and thorough empirical analysis on the performance of algorithms regarding approximation of Nash equilibria. In order to do this, we pose and address several relevant questions.

Efficiency of Exact Algorithms. One of the first questions which we need to address is the issue of finding exact Nash equilibria in practice. As mentioned earlier, the problem of computing an exact Nash equilibrium is `PPAD`-complete. However, we have to evaluate to what degree this difficulty affects the ability to make use of these algorithms in practice. If these algorithms are capable of handling and scaling-up with the size of the games, it would imply the futility of studying notions of approximation (besides the instances which prove to be hard to solve by the exact algorithms).

Another point on efficiency, is on the performance of approximation algorithms. We

[1] `https://bimatrix-games.github.io`
[2] `https://polymatrix-games.github.io`

need to be able to compare the speed advantage brought upon by using an approximation in contrast to an exact algorithm.

Approximation Guarantee. An important portion of studying approximation algorithms is the approximation guarantees which they provide. So far, the best approximation algorithms give a guarantee of 0.3393-NE and 0.5-NE for bimatrix and polymatrix games respectively. However, these guarantees are merely upper-bounds, with the possibility of achieving much better approximations in practice. If these theoretical guarantees cannot be beaten in practice, it is unlikely to be useful.

Approximation vs Time Tradeoff. From the theoretical perspective, we are already aware of the tradeoff between computing exact Nash equilibria and approximation of Nash equilibria. In bimatrix games, we can easily achieve a 1-NE guarantee in constant time by simply selecting an arbitrary strategy profile for the player. We can also achieve a 0.5-NE guarantee using the DMP algorithm in $O(n)$ for an $n \times n$ bimatrix game. On the opposite end of the spectrum, we have algorithms for computing exact Nash equilibria which do not have a polynomial time complexity.

We would like to study the degree to which this tradeoff transfers when looking at it from an empirical point-of-view. Do fast algorithms generally produce worse approximate equilibria? Should our desired quality of equilibrium impact our choice of algorithms?

Lower-bounds on Approximation Algorithms

In the first empirical study of the current best algorithm for computing approximate Nash equilibria (which produces a 0.3393-NE) amongst randomly generated games the worst ϵ-NE ever found was a 0.015-NE. Upon further studies we performed using other classes of games, we discovered a game where the algorithm produces a 0.1544-NE. Furthermore, we also discovered a polymatrix game where the current state-of-the-art algorithm (which has an upper-bound of 0.5-NE) produces a 0.1065-NE. These results beg the question as to whether these algorithms are actually tight.

We present near tight lower-bounds for both of these algorithms. For the bimatrix games, we approached the construction of this game using both a genetic algorithm and a Bayesian optimization method. This resulted in a bimatrix game where the algorithm found a 0.3385-NE. However, for the polymatrix game, we constructed a game which resulted in a 0.4835-NE by making use of an application of game theory to solve classification problems.

1.3 Structure of this Thesis

In Chapter 2, we present results from an empirical study of a range of algorithms for bimatrix games on a wide range of game classes. The chapter begins with an introduction of the problem being faced, highlighting areas which have been explored in previous studies. Next, the algorithms and game classes which we investigate are introduced, with details regarding the implementations of the algorithms and game generators being presented. Finally, we discuss the results of the experiments which we carried out. The results of this chapter have been published in SEA'15 with the title "An Empirical Study of Finding Approximate Equilibria in Bimatrix Games" [54].

In Chapter 3, we explore, through empirical analysis, algorithms for computing both exact and approximate Nash equilibria in several classes of polymatrix games. We start with a brief introduction to the area, followed by a summary of related work done. We then discuss the algorithms which we study, followed by a description of the 2-player Bayesian games and polymatrix games used in our study. Before giving an analysis of the algorithms, we briefly highlight some implementation issues and limitations which we faced. This chapter is based on the paper "An Empirical study on Computing Equilibria in Polymatrix Games" [42] published in AAMAS'16.

In Chapter 4, we direct our focus towards finding lower bounds for the current state-of-the-art approximation algorithms for bimatrix and polymatrix games. After a brief overview of the problem and summary of results are presented, we show a construction of a *graph transduction game* (which stems from applying game theory towards the problem of classification [49]) that yields a polymatrix game with a high approximation guarantee for the approximation algorithm. With regards to the bimatrix games, we look at two methods of heuristic search: *Tree-structured Parzan Estimator* (a method of performing Bayesian optimizations) and *genetic algorithms*.

Chapter 2

Bimatrix Games

2.1 Introduction

Most of the work on approximation algorithms have focused on additive approximations. Here, we study two different notions of additive approximate Nash equilibrium on bimatrix games. An *ϵ-Nash equilibrium* requires that both players achieve an expected payoff that is within ϵ of the best response, while the stronger notion of *ϵ-well supported Nash equilibrium* (ϵ-WSNE) requires that both players only place probability on strategies that are within ϵ of the best response. The current state of the art for ϵ-Nash equilibria is the algorithm of Tsaknakis and Spirakis [121], which finds a 0.3393-Nash equilibrium in polynomial time, and the current state of the art of ϵ-WSNE is the algorithm of Czumaj et al. [28], which finds a 0.6528-WSNE in polynomial time.

So far, most of the work on approximate equilibria have been theoretical in nature. This chapter aims to investigate if *approximate equilibria are relevant to the problem of solving bimatrix games in practice?* To answer this, we must study several related questions.

- Firstly, how good are the algorithms for finding exact Nash equilibria in practice? If they are good enough, then there is no need for approximation. Otherwise, how much faster are the approximation algorithms?

- Secondly, what quality of approximation do the approximation algorithms provide in practice? If the best theoretical guarantee of a 0.3393-Nash equilibrium is not trounced in practice, it is unlikely to be useful.

- Finally, is there a trade off between running time and approximation? We have a wide variety of approximation algorithms — from those that solve a single linear

program, to those that perform complicated gradient descent procedures. Do fast algorithms generally produce worse approximate equilibria? Should our desired quality of equilibrium impact our choice of algorithm?

2.2 Preliminaries

A *bimatrix game* is a pair (R, C) of two $m \times n$ matrices: R which gives the payoffs for the *row* player, and C which gives the payoff for the *column* player. The row player has m *pure strategies* represented by the rows, while the column player has n *pure strategies* represented by the columns. To play the game, both players simultaneously select a pure strategy: the row player selects a row i, and the column player selects a column j. This results in both the row and column player receiving a payoff of $R_{i,j}$ and $C_{i,j}$ respectively.

A *mixed strategy* is a probability distribution over a players pure strategies. We denote the mixed strategy of the row player by a vector x of size m, such that the value of x_i is the probability that the row player will play the pure strategy i. Using the same interpretation, the column players mixed strategy is denoted by the vector y of size n. Given the mixed strategies x and y for the row and column players respectively, then the row player receives a payoff of $x^T Ry$ and the column player receives a payoff of $x^T Cy$. For a given mixed strategy x for either player, the *support* ($supp(x)$) of the mixed strategy x is defined to be the set of pure strategies which are played with a non-zero probability in x (i.e. $supp(x) = \{i : x_i > 0\}$).

Nash Equilibria. Let y be a mixed strategy for the column player. The set of *pure best responses* for the row player against y, is the set of pure strategies which maximize the payoff against y. Put formally, a pure strategy $i \in [m]$ is a best response against y if, for all pure strategies $i' \in [m]$ we have: $(Ry)_i \geq (Ry)_{i'}$. In a similar fashion, we can derive the definition of the column players best responses.

The pair of mixed strategies (x, y), also known as a *mixed strategy profile*, is a *(mixed) Nash equilibrium* if every pure strategy in the support of the row players strategy x, is a best response against the column players strategy y, and vice versa. This results in a situation where the expected payoff gotten by playing a mixed strategy equals the pure best response against the opponents strategy.

Approximate Equilibria. There are two main notions of approximate Nash equilibria gotten by relaxing the requirements of a mixed Nash equilibria, both of which are studied in this chapter. The first notion is that of an *ϵ-approximate Nash equilibria* (ϵ-NE) which

relaxes the requirements that a players expected payoff should equal the payoff gotten by playing their best response. A mixed strategy profile (x, y) is said to be an ϵ-NE if for both players, the difference between the maximal payoff and expected payoff gotten against the opponents strategy is at most ϵ.

The other notion of approximation is that of *ϵ-approximate well-supported Nash equilibrium* (ϵ-WSNE), relaxes the requirement that players only play with non-zero probability strategies that are best responses, by allowing the players to play strategies which are ϵ-best responses with non-zero probability. A pure strategy i is an ϵ-best response for the row player if: $\max_{i' \in [n]} ((Ry)_{i'}) - (Ry)_i \leq \epsilon$. A mixed strategy profile (x, y) is said to be an ϵ-WSNE if every pure strategy in the support for each player is an ϵ-best response against the opponents strategy.

2.3 Contribution

While there have been several empirical studies on computing exact equilibria [26,61,104, 114], the empirical work for approximate equilibria has so far been limited to a paper [122] that evaluates the algorithm of Tsaknakis and Spirakis (TS) [121], and one that looks exclusively at symmetric games [81]. We address this by performing a comprehensive study of approximation algorithms. In particular, we compare the performance of five different algorithms for finding approximate equilibria on 15 different types of games. Moreover, we include two algorithms for finding exact equilibria: the Lemke-Howson algorithm and support enumeration.

This allows us to answer the three questions that we posed earlier. Firstly, we find that approximation algorithms can tackle (non-pathological) instances that exact algorithms cannot. With a timeout of 15 minutes, we find that exact algorithms were mostly unable to solve instances of size 1000×1000; whereas approximation algorithms were easily able to tackle instances up to size 2000×2000. Secondly, we find that approximation algorithms often find much better approximations than their theoretical worst case might suggest. In particular, in agreement with the experimental study of Tsaknakis et al. [122], we find that the TS algorithm often finds 0.01-Nash equilibria or better. In answer to our third question, we observe that the TS algorithm clearly wins in terms of the quality of approximation, although it is usually the slowest. Also, if we only require weaker approximate equilibria, then other algorithms can find one faster. An analysis of this trade off can be found in Section 2.8.

To obtain our results, we tested the algorithms on a wide variety of games. Previous

work on exact equilibria have typically used the GAMUT library [98]. However, we find that almost all games provided by GAMUT have exact pure Nash equilibria, so using the games in GAMUT alone would skew our results. For example, the work of Porter et al. [104] concluded that support enumeration typically outperforms the Lemke-Howson algorithm, based largely on the fact that support enumeration can quickly find the pure equilibria in the games provided by the GAMUT library.

There are many practical applications of game theory where players cannot use pure strategy profiles in equilibrium. Our second contribution is to define several natural classes of games that do not have pure strategy Nash equilibria. We have made all our game generators and algorithm implementations open source and freely available[1]. Our results show that our algorithms perform very differently on these types of games, so they should be included in any future study of the practical aspects of computing equilibria.

2.4 Related work

GAMUT provides a suite of game generators that have been used in previous studies [98]. Porter et al. [104] use GAMUT to compare two algorithms for finding exact Nash equilibria: support enumeration (SE) with the Lemke-Howson (LH) algorithm. They showed that SE performs well when compared to LH, because many of the games in the library have small support equilibria. They also highlight random games and covariant games as the most challenging GAMUT games for SE and LH, which our results also confirm. Most of their experiments considered games of size 600×600, although random games up to size 1000×1000 were also considered.

Sandholm at al. [114] describe four ways of solving games via mixed integer programming (MIP). Experiments were carried out on games of size 150×150 provided by GAMUT. It was found that MIP performed better than LH but was outclassed by SE.

Codenotti et al. [26] performed an experimental study of LH algorithm, on random and covariant games, where it was found that LH has a running time of roughly $O(n^7)$ for $n \times n$ covariant games. They presented a heuristic which involves running different LH paths (for different dropped pure strategies) with limits on the number of allowed pivots for a given path, unless no equilibrium is found by any path. This heuristic showed a significant improvement over LH for random games, but did not have any advantage over LH in covariant games. They looked at games of size up to 1000×1000.

[1] `http://bimatrix-games.github.io/`

Tsaknakis et al. [122] performed an experimental analysis of their algorithm (TS) for finding a 0.3393-Nash equilibrium in polynomial time. They studied games of size 100×100, and they also constructed games with no small support equilibria, to prevent easy solutions. Among the games they studied, it was found that TS always finds a 0.015-Nash equilibrium or better. We confirm this result that in general TS performs very well; however among some of our games TS only finds a 0.1544-Nash equilibrium.

More recently, Gatti et al. [61] evaluated the performance of LH, MIP and SE for different classes of games. They found that none of the methods was superior for all games. They introduced a number of heuristics, and compared their performance on the games from GAMUT. They did look at the quality of approximation achieved by their heuristics and algorithms, but the largest games they looked at were of size 150×150.

2.5 Algorithms

Now, we describe the algorithms that are studied in this paper. For the computation of exact equilibria, we study the following two algorithms.

- **LH:** The Lemke-Howson algorithm is a pivoting algorithm [86]. The algorithm works by having for each player, a representation of the best responses against the opponents strategies in the form of a best response polytope. Starting from the artificial equilibrium, where both players strategies are the 0 vector, it has the free choice of a starting pivot that corresponds to a pure strategy in the game. Once this pivot has been made, subsequent pivots follow in a complementary manner, uniquely defined by its current state. Pivoting continues until an equilibrium is found.

  It is widely used, despite its exponential worst case behaviour [116], and the fact that it is PSPACE-hard to compute the equilibrium that it finds [64]. We make use of the implementation from [26], with modifications so that degeneracy resolution uses the lexicographic minimum ratio test.

- **SE:** Support enumeration is a brute force algorithm. It goes through every possible pair of support profile, and solves a system of linear equations to check whether there exists a Nash equilibrium with the given pair of support profiles. Our implementation goes through supports from small to large cardinality.

There is a wide variety of algorithms for finding ϵ-Nash equilibria, and we have implemented the following (polynomial-time) algorithms:

- **PURE:** This algorithm checks all pure strategy profiles and returns one that gives the best ϵ-Nash equilibrium.

- **DMP:** This algorithm was given by Daskalakis et al. [34]. It finds a 0.5-Nash equilibrium using a very simple approach that starts with an arbitrary pure strategy, and then makes two best response queries. The algorithm operates as follows: one player fixes an arbitrary pure strategy, his opponent finds a pure best response y, and then the first player finds a best response against y. A 0.5-Nash equilibrium is obtained when the first player mixes uniformly over his two pure strategies, and the second player plays y probability 1.

- **BBM1:** This is the first of two algorithms given by Bosse et al. [17]. It finds a 0.3819-Nash equilibrium. Given a bimatrix game (R, C), BBM1 solves the zero-sum game $(R - C, C - R)$ using linear programming. Then it proceeds in a similar manner to DMP, but uses the LP solution as the initial strategy. Given the solution to the above mentioned zero-sum game is (x^*, y^*), if this solution yields an approximation of 0.3819-NE it gets returned as the solution. Otherwise, assuming the regret of the column player is larger than that of the row player, the column player selects their best response to x^*, c_j. Finally, the algorithm arrives at a solution by letting the row player mix between x^* and the best response to c_j, while the column player plays c_j as a pure strategy.

- **BBM2:** This is the second of the two algorithms given by Bosse et al. [17]. It finds a 0.3639-Nash equilibrium. It is an adaptation of BBM1 that contains some extra steps to deal with cases where the first algorithm performs poorly. In contrast to the BBM1, this algorithm gets the column player to play a mixed strategy y where he mixes between y^* and c_j (the best response against x^*). In return, the row player then plays a mixed strategy x, gotten by mixing x^* and r_i (the best response against y).

- **TS:** This algorithm was given by Tsaknakis and Spirakis [121]. It finds a $(0.3393 + \delta)$-Nash equilibrium, where δ is an arbitrary positive constant. The algorithm uses gradient descent over the space of mixed strategy profiles, with its objective function being the quality of approximate Nash equilibrium.

  1. Initialize the algorithm with an arbitrary point (x, y).
  2. If the regrets of the players are unequal, then an LP is solved to equate it.

3. Compute the gradient (x', y')
4. If it is (x, y) corresponds to a stationary point or a $(0.3393 + \delta)$-NE can be achieved, then stop
5. Restart from step 2 with the points $(x + \alpha(x' - x), y + \alpha(y' - y))$, where $\alpha = \frac{\delta}{1+\delta}$.

The algorithm finds a stationary point. If the stationary point is not a $(0.3393 + \delta)$-Nash equilibrium, then it can be used to find a second point that is a $(0.3393 + \delta)$-Nash equilibrium.

Our implementation differs from the original specification of the algorithm in two ways. The first of which is the value of α. In a similar manner to an earlier analysis of the TS algorithm [122], we make use of a line search algorithm, which finds the value of α that would yield the greatest decrease in the maximum regret of both players. The second difference, is in the stopping criteria. By allowing the algorithm to stop as soon as a $0.3393+\delta$-NE is achieved, the algorithm ends up terminating in a lot of cases, within a single iteration. In order to properly analyse the quality of Nash equilibrium which can be gotten at the stationary points, we forgo this condition. In other words, the implementation used within this study, always reaches a δ stationary point before any computation of the extra points are made. After which the point with the best approximation guarantee is returned.

To investigate the dependence of the algorithm on δ, we use two versions with $\delta = 0.2$ and $\delta = 0.001$. We refer to these as **TS2** and **TS001**, respectively. At the end of Section 2.8.3, we analyze the effect of the choice of δ.

There has been comparatively less study of algorithms to find approximate well-supported Nash equilibria. We implemented the following two algorithms:

- **KS:** This algorithm was given by Kontogiannis and Spirakis [80], which finds a $\frac{2}{3}$-WSNE. The algorithm first checks all pure strategy profiles in order to determine if there is a pure $\frac{2}{3}$-WSNE. If not, then the algorithm solves the same zero-sum game as BBM1/BBM2, $(R - C, C - R)$, and the equilibrium returned from this computation is guaranteed to be a $\frac{2}{3}$-WSNE in the original game.

- **KS+:** This algorithm gives an improved approximation guarantee compared to KS [53] of $(\frac{2}{3} - 0.00591)$. It combines the KS algorithm with two extra procedures: one that finds the best WSNE with 2×2 support, and the other which finds the best WSNE on the supports from an equilibrium to the KS zero-sum game.

2.6 Game classes

We now describe the classes of games used in our study. In order to have a consistent meaning of approximation guarantees, all games are scaled to have payoffs in $[0, 1]$. Firstly, we use games provided by the GAMUT library. We made use of classes of games in GAMUT that could be scaled indefinitely. We eliminate the classes of games that have a fixed size, and we were also forced to eliminate some classes of games because their generators either crashed or produced invalid games when asked to produce games with more than 1000 strategies per player, which include BidirectionalLEG and RandomLEG . With the exception of parameters which we explicitely declare, all other parameters are chosen arbitrarily from the options available in the GAMUT library. A slight modification to the GAMUT library was made so as to avoid using exponential functions whenever a function is needed as a parameter, in order to avoid generating payoffs too large to be computed or stored by the GAMUT library especially when generating large games. We were left with the games shown below.

- **BertrandOligopoly:** All players in this game are producing exactly the same item and are expected to set a price at which to sell the item. The player with the lowest price gets all the demand for the item and produces enough items to meet the demand. If both players have the same price, then they split the demand equally. This results in a payoff of $\frac{p(D(p))}{m} - C(\frac{D(p)}{m})$ for the players with the lowest price where p is the lowest price, m is the number of players with the lowest price, C and D are the cost and demand functions given the price respectively. While the other player gets a payoff of 0. Both the cost and demand functions are chosen arbitrarily.

- **CournotDuopoly:** In contrast to Bertrand's Oligopoly, in this setting, each player choses a quantity of the same item to be produced which would be sold at the same price determined by the total quantity produced. This results in a payoff of $q_i P(Q) - C_i(q_i)$ for player i, where q_i is the quantity being produced by player i, Q is the total quantity being produced by all players, C_i is the cost incured by player i producing q_i items and P is the inverse demand function which determines the price of each item given the quantity produced.

- **CovariantGame:** These are games where the payoffs for both players for each pure strategy profile are drawn from a multivariate normal distribution with covariance ρ, are particularly notable. When $\rho = 1$ we have a coordination game and when $\rho = -1$ we have a zero-sum game. Previous work [26, 104] indicates that

these games are easy to solve when $\rho > 0$, with the hardest games in the range $[-0.9, -0.5]$. We study 5 classes of games CovariantGame-p for $p = 1, 3, 5, 7, 9$ where $\rho = -0.1, -0.3, -0.5, -0.7, -0.9$ respectively.

- **GrabTheDollar:** In this game, there is a price (a dollar) which is up for grabs and both players have to choose when to grab the price. If both players choose to grab the money at the exact same time, then the price gets ripped and both players get a payoff of 0, otherwise, the player who attempted to grab the price at an earlier time gets a payoff of 1 and the other player gets a payoff of c where $0 < c < 1$.

- **GuessTwoThirdsAve:** Each player in this game chooses a number, trying to come as close as possible to two thirds of the average of all the numbers guessed by all players. The player closest to two thirds of the average receives a payoff of 1, while the others receive a payoff of 0. In the case of a tie, the players split the maximum payoff equally.

- **MinimumEffortGame:** In this game, the payoff of each player is determined by a formula $a + bM - cE$ given that $-1 \leq a \leq 1$, $0 \leq c < b \leq 1$ where a, b and c are constants with M being the minimum effort of the players and E is the effort being put in by the given player.

- **TravelersDilemma:** In this game, each player simultaneously requests an amount of money and receives the lowest of the requests submitted. If there was a tie, no player gets an extra reward, otherwise, the player with the lowest value gets a reward of $r > 0$.

- **RandomGame:** These are games where by the value of each individual payoff is independently selected uniformly at random from the range $[0, 1]$.

- **WarOfAttrition:** In this game, both players are competing for a single object which each player i values at v_i and chooses a time in which to concede and give the item to the opponent. If both players concede at the exact same time, the item gets shared between them. However, at each point in time, the players incur a cost (or a loss of valuation), resulting in the following payoff function for player i

$$U_i(t_i, t_j) = \begin{cases} v_i - t_i c_i & \text{if } t_i > t_j \\ \frac{v_i}{2} - t_i c_i & \text{if } t_i = t_j \\ -t_i c_i & \text{if } t_i < t_j \end{cases},$$

where v_i is player i's valuation of the item, c_i is the cost incured in every time step, t_i and t_j are the times in which players i and j would concede respectively.

With the exception of covariant and random games, the other bimatrix game classes provided by GAMUT have pure equilibria, and are therefore easily solved by support enumeration.

2.6.1 Game Generators

In order to broaden our study, we chose to implement generators for the following games, which generally do not have pure equilibria.

- **Non-zero sum colonel Blotto Games:** The players have an equal number of soldiers T that must be assigned simultaneously across n different hills. Each player has a personal valuation for each hill, and he receives this valuation if he assigns strictly more soldiers to the hill than his opponent, and in case of ties, the hill goes to a player chosen uniformly at random. Each player's overall payoff is the sum of the valuations of the hills won by that player. In order to avoid pure equilibria, we draw the valuations of both players from a multivariate normal distribution with a covariance of $\rho > 0$. In our experiments, we study families of games with $n = 3, 4$ and $\rho = 0.5, 0.7, 0.9$, which we denote by Blotto-n-p for $p = 5, 7, 9$, respectively. The number of soldiers T was varied in order to generate a scalable family of games.

- **Competitiveness-based Ranking Games:** [63] Each player chooses an effort level with an associated cost and score. A prize is given to the player with the higher score, or it is split in the case of a tie. The payoff of a player is the value of the prize received minus the cost of the effort expended. We generated the score and cost as strictly increasing step functions of effort, with random step sizes. We denote these games by Ranking.

- **SGC games:** Sandholm et al. [114], in their experimental study, also noted that most GAMUT games have small support equilibria. To tackle this problem, they introduced a family of games where all equilibria have both players placing positive probability on half of their actions. We denote their games as SGC . In these game the only equilibrium in a $(2k-1) \times (k-1)$ game has support sizes k for both players, which makes these games hard for support enumeration.

- **Tournament Games:** [3] Starting with a random tournament, an asymmetric bipartite graph is constructed where nodes on one side correspond to the nodes of the tournament, while nodes on the other side correspond to subsets of nodes in the tournament. The bipartite graph is transformed into a win-lose game where the actions of each player are the nodes on their side of the graph. The size of the subset of nodes changes the number of strategies the column player has in relationship to the row player. We study games which are constructed with subsets of size 2 or 3. We denote these games as Tournament-2 and Tournament-3 respectively, or Tournament in general.

- **No-pure random Unit Vector Games (UVG):** [117] The payoffs for the column player are chosen uniformly at random from the range $[0, 1]$, but in the row player's matrix, each column j contains exactly one 1 payoff with the rest being 0. In order to avoid pure equilibria, we generated these games by placing the 1s uniformly at random in the rows that do not generate a pure equilibrium. We denote these games as Unit.

Unlike other studies [61, 114], we do not include the exponential-time examples for LH devised by Savani and von Stengel [116]. They are not suitable for an experimental study on approximate equilibria because the games have a number of very large payoffs, so when they are normalised to the range $[0, 1]$, almost all payoffs in the game are close to 0. Hence these games are very easy to approximate. For example, the 30×30 instance has a pure 1.63×10^{-10}-WSNE. Furthermore, for instances larger than 16×16, these games exhaust the precision of floating point, making it impossible for a floating point implementation of the LH algorithm to solve the game.

2.7 Implementation Details

Implementations of the algorithms are done in C, and we used CPLEX to solve the linear programs used in some of the algorithms. The CPLEX library was chosen due to its popularity within the academic community. It provides implementations of several algorithms for the purpose of solving linear programs including the simplex and barrier methods as well as a heuristic based approach for choosing a solver based on the nature of the LP which it is solving. However, we opted to use the Simplex algorithm in order to be able to have a fair comparison across the board for all solvers. Also, according to the documentation for the CPLEX library, the simplex method is the preferred method especially for LPs

having less than 100,000 rows or columns. For the sizes of games which we consider in this study, the largest LP is approximately 4000×4000, hence the decision to make use of the simplex method.

For our runtime results, we only measured the amount of time spent by the solver, and discarded the time taken to read the game from its input file. Our experiments were carried out on a cluster of 8 identical machines running Scientific Linux 6.6, which each have an Intel Core i7-2600k processor clocked at 3.40GHz with 16GB of RAM.

To verify our results, we implemented three programs that compute the quality of exact, approximate, and well-supported Nash equilibria, respectively. All of these programs carry out their calculations in *exact arithmetic*. Our exact equilibrium checker takes a pair of supports, and checks whether there exists a Nash equilibrium on these supports. Both of the approximate checkers take a mixed strategy profile, and output the value of ϵ that this profile achieves.

2.7.1 Issues with implementation

One of the main issues faced during the implementation of the algorithms was with regards to floating point errors, most especially in the implementation of the LH algorithm. In most implementations of algorithms, floating point arithmetic is the norm. The reason lies not only not in the fact that it is a widely supported data type across most programming and scripting languages, but are considerably faster and tend to use less memory in comparison to the alternative of exact representation using rationals.

Due to the pivoting nature of the LH algorithm, the elements of a given row are scaled (by dividing each value in the row by the pivot element), and multiples of the row are subtracted from the other rows in the tableau. This results in the round-off errors propagating throughout the tableau at each step of the algorithm. In order to ensure the algorithm was correctly implemented, we also made an implementation using exact arithmetic.

Although there is no method of completely avoiding floating point errors (without the use of exact arithmetic) when making use of floating points, one method of minimizing its effects is to make use of a *machine epsilon*. Strictly speaking, this term refers to a value ϵ_d which upper bounds the relative error caused as a result of rounding. In the implementations of all the algorithms, we made use of the default value provided by the *C* compiler ($\epsilon_d = 2.220446 \times 10^{-16}$, as defined in `float.h`).

A Quick Note on Size

Due to the nature of some of the games which we use in this study, it is considerably difficult to make the sizes of the games match exactly. Some of the game generators allow us to be able to create games with great flexibility regarding the size of the instance to be generated (for instance RandomGame, Unit, and so on). Whereas, other game classes are extremely restrictive in regards to the range of sizes which they can generate (such as SGC, Colonel-Blotto and Tournament games).

In order to address this, we use instances that have roughly the same number of payoffs as the corresponding square games. For example, we compare 100×100 RandomGame to a 105×105 Colonel-Blotto game.

2.8 Experimental Results

In the following presentation of our results, we divide the games into three main classes. The first which contains games from the GAMUT library which have pure Nash equilibria, the second class of games are also from the GAMUT library, but do not always posses a pure Nash equilibrium and have been found to be hard cases for exact algorithms in previous research work [61, 114]. The final class of games are the games which we proposed in Section 2.6.1.

2.8.1 Floating Point Stability of Lemke-Howson

Although we took steps towards avoiding floating point errors by making use of the default machine epsilon which is available within the C compiler, we were still able to notice certain floating point errors. The instances which resulted in floating point errors were flagged by checking if the probability distributions do not sum up to one, if there exists a strategy played with probability greater than 1 or significantly less than 0 (-1×10^{-10}), and finally if the ϵ-NE computed by our checker is above 5×10^{-6}. Due to the nature of this method of verification, we are unable to detect all instances which had floating point errors, especially those which still terminated with a Nash equilibrium although they followed a different path from that which would have been generated by an implementation making use of exact arithmetic.

From the range of games which we study, instances of the Tournament games are more likely to result in a floating point error than any other game. Following these games are TravelersDilemma and Colonel Blotto games.

Out of the three values of epsilon used with regards to floating point implementation of LH, as shown in Table 2.1, we find that a value of 1×10^{-20} was the most stable of them. As a result, for the remainder of this chapter we make use of a machine epsilon of 1×10^{-20}.

Games	2.2×10^{-16}			1×10^{-17}			1×10^{-20}		
	100	300	1000	100	300	1000	100	300	1000
BertrandOligopoly	5.43	15.7	20	5.43	9.43	16.7	5.43	9.26	14.3
CournotDuopoly	4	3	5	1	1	2	2	1	2
GrabTheDollar	0	5	5.05	3.03	8	4	3	5.05	6.06
GuessTwoThirdsAve	0	0	0	0	0	0	0	0	0
LocationGame	0	0	0	0	0	0	0	0	0
MinimumEffortGame	0	0	0	0	0	0	0	0	0
TravelersDilemma	36.8	42.2	47.6	20.3	25.9	20	29.2	33.7	36.4
WarOfAttrition	0	0	0	0	0	0	0	0	0
CovariantGame-1	0	0	0	0	0	0	0	0	0
CovariantGame-3	0	0	-	0	0	-	0	0	-
CovariantGame-5	0	-	-	0	-	-	0	-	-
CovariantGame-7	0	-	-	0	-	-	0	-	-
CovariantGame-9	0	0	-	0	0	-	0	0	-
RandomGame	0	0	0	0	0	0	0	0	0
Blotto-3-5	13.8	10.5	23	10.1	0	24.6	8.79	10.2	25
Blotto-3-7	13.5	17.4	33.8	16.4	0	26	12.5	19.3	27.4
Blotto-3-9	17.2	23.2	28.2	17.9	4.35	26.3	19.1	17.5	20.8
Blotto-4-5	12.5	27.8	28.1	26.2	27.7	35.8	16.5	23.3	30.8
Blotto-4-7	17.4	31.5	25	23.1	30.6	34	15.4	27	31.3
Blotto-4-9	17.9	23.9	26.4	19.1	23.1	26.5	22	21.9	21.9
Ranking	3	11	38	2	9	29	0	1	1
Tournament-2	92.8	100	100	92.9	100	100	82.4	100	100
Tournament-3	30.8	81.7	100	28.1	79	98	22.6	73.5	100
Unit	0	0	0	0	0	0	0	0	0

Table 2.1: Showing the percentage of instances out of those which did not timeout, where there were detectable point errors. Each set of three columns represents an implementation of LH with a given value for the machine epsilon on instances of games of various sizes (we used sizes 105, 300 and 1035 for Blotto-3 games, 120 364 and 969 for Blotto-4 games, 27, 57 and 126 rows for Tournament-2, 16, 28 and 50 rows for Tournament-3 and 100, 300 and 1000 for all other games.)

2.8.2 Exact Algorithms

We start by testing the limits of algorithms for exact equilibria. To do this, we tested LH and SE against our library of games with a timeout of 15 minutes. Table 2.2 shows the percentage of games that were solved by these two algorithms for various game sizes, along with the average running time. We have divided the games into three classes. Firstly we have the games from GAMUT that always have pure equilibria. As we would expect, SE performs very well on these games, while LH is also able to tackle the majority of instances. Secondly, we have the GAMUT games that do not always have pure equilibria. Here we can see that both algorithms perform very poorly for covariant games, which is in agreement with previous studies. Finally, we have the games that we have proposed. It can be seen that both algorithms significantly struggle with the games in this class, which supports the idea that GAMUT's existing library does not give a comprehensive picture of possible games. Ranking games provide an interesting case that differentiates between LH and SE : these games only had equilibria with medium sized support so SE was hopeless, however LH was able to solve these games using a linear number of pivots. The conclusion of these experiments is that exact methods tend to be inadequate for the 2nd and 3rd classes of games, so it is for these games that we are particularly interested in the performance of approximation methods.

2.8.3 Approximation algorithms

Next we tested our suite of approximation algorithms against the library of games. Tables 2.3, 2.4 and 2.5 show the running time and quality of approximation, respectively, when the algorithms were tested on games of size 2000×2000. Again, there is a clear split in the data between the "easy" games from GAMUT and the more challenging game classes. Note that, with only a handful of exceptions, the approximation algorithms were easily able to deal with games of this size. Only TS001 was observed to time out, and it timed out on 0.4% of the instances that it was tested on. The timed out instances consisted of 1 instance of LocationGame, 2 instances of BertrandOligopoly and 4 instances of TravelersDilemma. This indicates that approximation algorithms can indeed be applied to games that cannot be tackled with exact methods. We summarize the performance of the algorithms as follows.

- **DMP.** As one might expect for such a simple algorithm, it runs very quickly and usually gives a very poor approximation. In terms of quality, it is clearly outmatched

Games	SE						LH					
	100		300		1000		100		300		1000	
	%T	Time	%T	Time	%T	Time	%T	Time	%T	Time	%T	Time
BertrandOligopoly	0	3.7e-05	0	0.000261	0	0.00813	8	12	46	70.8	58	129
CournotDuopoly	0	0.00269	0	0.079	0	4.6	0	8.01e-05	0	0.00215	0	0.0436
GrabTheDollar	1	3.4e-05	0	5.12e-05	0	8.19e-05	0	0.00742	1	0.251	1	16
GuessTwoThirdsAve	0	3.33e-05	0	4.49e-05	0	8.22e-05	74	13.7	89	87	97	224
LocationGame	0	0.00633	1	0.192	2	11.6	0	7.93e-05	0	0.00727	0	1.66
MinimumEffortGame	0	3.25e-05	0	4.98e-05	0	8.4e-05	0	7.79e-05	0	0.00146	0	0.0274
TravelersDilemma	0	3.2e-05	0	5.02e-05	0	8.16e-05	4	0.0922	14	12.1	23	107
WarOfAttrition	0	3.37e-05	0	5.76e-05	0	0.000111	0	9.7e-05	0	0.00229	0	0.0442
CovariantGame-1	64	0.00278	64	0.083	73	6.28	0	0.478	30	110	95	173
CovariantGame-3	91	0.00252	94	0.131	99	3.43	0	2.15	84	335	100	-
CovariantGame-5	100	-	100	-	100	-	0	4.33	100	-	100	-
CovariantGame-7	100	-	100	-	100	-	0	2.3	100	-	100	-
CovariantGame-9	100	-	100	-	100	-	0	0.158	35	549	100	-
RandomGame	37	0.00241	37	0.0853	36.3	4.71	0	0.127	14	81	81.4	125
Blotto-3-5	41.3	0.00146	100	-	36	1.15	9	0.00298	12	3.22	24	4.73
Blotto-3-7	37.3	0.00202	100	-	41.3	2.15	12	0.00959	12	0.0655	27	4.78
Blotto-3-9	50.7	0.00231	100	-	48	2.84	11	0.0187	20	0.0715	28	7.42
Blotto-4-5	42.7	0.00239	56	0.0776	45.3	1.75	9	0.0418	27	0.0868	35	1.12
Blotto-4-7	48	0.00288	54.7	0.0964	38.7	1.79	9	0.00328	26	3.33	33	3.5
Blotto-4-9	60	0.00323	65.3	0.118	62.7	2.77	9	0.0378	27	0.12	36	0.942
Ranking	100	-	100	-	100	-	0	0.00785	0	0.281	0	17.9
Tournament-2	78	0.00326	100	-	100	-	49	0.101	59	4.55	59	31.2
Tournament-3	9	5.45e-05	67	0.00139	75	1.15	7	3.09	32	0.679	51	57
Unit	100	-	100	-	100	-	0	0.0174	0	16.8	49	147

Table 2.2: Showing the percentage of instances which timed out (%T) and the average time it took to solve each of the remaining instances.

Games	PURE			DMP		
	Time	ϵ-NE	ϵ-WSNE	Time	ϵ-NE	ϵ-WSNE
BertrandOligopoly	0.0317	0	0	5.33e-05	0.372	0.745
CournotDuopoly	39.9	0	0	4.17e-05	0.24	0.481
GrabTheDollar	9.61e-05	0	0	5.56e-05	0.268	0.537
GuessTwoThirdsAve	7.42e-05	0	0	5.64e-05	0.5	1
LocationGame	94.8	0.00372	0.00372	4.51e-05	0.216	0.427
MinimumEffortGame	7.43e-05	0	0	5.3e-05	0	0
TravelersDilemma	7.58e-05	0	0	5.47e-05	0.161	0.321
WarOfAttrition	0.00038	0	0	4.24e-05	0	0
CovariantGame-1	86.8	0.0138	0.0138	5.4e-05	0.193	0.38
CovariantGame-3	95.7	0.0451	0.0451	5.48e-05	0.219	0.434
CovariantGame-5	94.5	0.0943	0.0943	5.61e-05	0.261	0.517
CovariantGame-7	96.1	0.138	0.138	5.43e-05	0.295	0.589
CovariantGame-9	95.8	0.207	0.207	5.47e-05	0.317	0.634
RandomGame	59.8	6.01e-05	6.01e-05	5.29e-05	0.323	0.531
Blotto-3-5	42.5	0.0426	0.0426	4.38e-05	0.356	0.671
Blotto-3-7	45.5	0.0593	0.0593	4.29e-05	0.352	0.678
Blotto-3-9	57.2	0.0799	0.0799	4.3e-05	0.416	0.819
Blotto-4-5	42.1	0.0446	0.0446	4.25e-05	0.299	0.564
Blotto-4-7	60.3	0.0836	0.0836	4.18e-05	0.328	0.632
Blotto-4-9	78.8	0.0922	0.0922	4.34e-05	0.352	0.681
Ranking	94.8	0.171	0.171	5.43e-05	0.328	0.641
Tournament-2	249	1	1	8.06e-05	0.5	1
Tournament-3	14.4	0.75	0.75	6.41e-05	0.5	1
Unit	95.3	0.000591	0.000591	5.43e-05	0.344	0.689

Table 2.3: Showing experimental results of PURE and DMP algorithms on games of size 2000×2000 (games of sizes 2016 for Blotto-3, 2024 for Blotto-4, 200 rows for Tournament-2 and 71 rows for Tournament-3)

Games	BBM1			BBM2		
	Time	ϵ-NE	ϵ-WSNE	Time	ϵ-NE	ϵ-WSNE
BertrandOligopoly	1.09	0.000167	0.000167	1.1	0.000167	0.000167
CournotDuopoly	0.742	2.57e-09	2.57e-09	0.75	2.57e-09	2.57e-09
GrabTheDollar	0.757	0.0959	0.0959	0.745	0.0926	0.108
GuessTwoThirdsAve	0.736	0	0	0.73	0	0
LocationGame	0.848	0.0094	0.0113	0.837	0.0094	0.0113
MinimumEffortGame	0.757	0	0	0.746	0	0
TravelersDilemma	0.734	0	0	0.739	0	0
WarOfAttrition	0.808	0	0	0.799	0	0
CovariantGame-1	86.6	0.00892	0.0176	87.4	0.00892	0.0176
CovariantGame-3	86.8	0.00798	0.0155	87.8	0.00798	0.0155
CovariantGame-5	88.2	0.00645	0.0127	87.3	0.00645	0.0127
CovariantGame-7	87.7	0.00512	0.0101	86.4	0.00512	0.0101
CovariantGame-9	86.6	0.00297	0.00584	86.1	0.00297	0.00584
RandomGame	89.1	0.028	0.0547	87.9	0.028	0.0547
Blotto-3-5	1.06	0.0845	0.155	1.06	0.0852	0.168
Blotto-3-7	1.03	0.0654	0.117	1.05	0.065	0.122
Blotto-3-9	1	0.0477	0.0739	1.03	0.0477	0.0739
Blotto-4-5	0.976	0.111	0.179	0.955	0.112	0.185
Blotto-4-7	0.99	0.101	0.163	0.992	0.0998	0.162
Blotto-4-9	1	0.0603	0.0912	0.967	0.0603	0.0912
Ranking	4.25	0.148	0.173	4.24	0.148	0.173
Tournament-2	1.03	0.0557	0.0686	1.03	0.0557	0.0686
Tournament-3	0.126	0.118	0.144	0.123	0.118	0.144
Unit	87.4	0.00605	0.00657	87.7	0.00605	0.00657

Table 2.4: Showing experimental results of BBM1 and BBM2 algorithms on games of size 2000×2000 (games of sizes 2016 for Blotto-3, 2024 for Blotto-4, 200 rows for Tournament-2 and 71 rows for Tournament-3)

Games	TS001			TS2		
	Time	ϵ-NE	ϵ-WSNE	Time	ϵ-NE	ϵ-WSNE
BertrandOligopoly	356	0.000953	0.0404	24.7	0.0598	0.192
CournotDuopoly	2.18	2.43e-06	0.169	2.08	0.00229	0.295
GrabTheDollar	4.29	6.75e-06	0.045	4.59	2.8e-06	0.000503
GuessTwoThirdsAve	0.567	0	0	0.74	0	0
LocationGame	7.15	6.2e-05	0.0375	3.73	0.000973	0.537
MinimumEffortGame	1.39	2.46e-06	0.00128	2.21	0.00019	0.000748
TravelersDilemma	154	8.89e-05	0.00266	50.5	0.00262	0.0176
WarOfAttrition	2.27	3.2e-07	0.00606	2.46	9.3e-08	0.00465
CovariantGame-1	184	0.00038	0.0194	310	0.000711	0.0127
CovariantGame-3	177	0.0004	0.0184	311	0.000716	0.0121
CovariantGame-5	195	0.000429	0.0162	318	0.000755	0.0113
CovariantGame-7	230	0.000422	0.0133	324	0.000803	0.0109
CovariantGame-9	392	0.000249	0.0083	333	0.000874	0.0118
RandomGame	319	0.000392	0.0311	328	0.00211	0.0403
Blotto-3-5	8.87	0.00532	0.215	4.29	0.0219	0.298
Blotto-3-7	7.62	0.00335	0.221	3.73	0.0165	0.266
Blotto-3-9	7.69	0.000646	0.193	3.77	0.0146	0.224
Blotto-4-5	6.09	0.00375	0.129	2.71	0.0185	0.175
Blotto-4-7	5.74	0.00209	0.143	2.5	0.022	0.191
Blotto-4-9	5.79	0.000659	0.0829	2.67	0.0152	0.109
Ranking	46.1	0.000127	0.0277	4.43	0.0603	0.257
Tournament-2	8.8	5.58e-05	0.00945	41.2	0.00745	0.0596
Tournament-3	7.6	5.24e-05	0.00844	0.314	0.0512	0.0909
Unit	185	0.000386	0.0209	65.4	0.00376	0.01

Table 2.5: Showing experimental results of TS001 and TS2 algorithms on games of size 2000×2000 (games of sizes 2016 for Blotto-3, 2024 for Blotto-4, 200 rows for Tournament-2 and 71 rows for Tournament-3)

Games	KS		
	Time	ϵ-NE	ϵ-WSNE
BertrandOligopoly	0.0314	0	0
CournotDuopoly	40.3	0	0
GrabTheDollar	9.94e-05	0	0
GuessTwoThirdsAve	7.77e-05	0	0
LocationGame	94.7	0.00223	0.00412
MinimumEffortGame	7.63e-05	0	0
TravelersDilemma	7.33e-05	0	0
WarOfAttrition	0.000392	0	0
CovariantGame-1	148	0.00614	0.0113
CovariantGame-3	174	0.00783	0.0152
CovariantGame-5	175	0.00645	0.0127
CovariantGame-7	177	0.00512	0.0101
CovariantGame-9	175	0.00297	0.00584
RandomGame	86.3	6.01e-05	6.01e-05
Blotto-3-5	43.4	0.017	0.024
Blotto-3-7	47.3	0.0178	0.0438
Blotto-3-9	58.1	0.0185	0.0331
Blotto-4-5	44.1	0.0251	0.0344
Blotto-4-7	59.6	0.0399	0.0613
Blotto-4-9	78.4	0.0329	0.0523
Ranking	99.3	0.148	0.173
Tournament-2	247	0.0557	0.0686
Tournament-3	14.3	0.0849	0.108
Unit	177	0.000591	0.000591

Table 2.6: Showing experimental results of KS algorithm on games of size 2000×2000 (games of sizes 2016 for Blotto-3, 2024 for Blotto-4, 200 rows for Tournament-2 and 71 rows for Tournament-3)

by all of the other algorithms.

- **BBM1 and BBM2.** These algorithms were typically in the middle in terms of both approximation quality and running time. Only a handful of Blotto instances triggered the extra steps in BBM2, so these two algorithms are mostly identical.

- **PURE.** Whenever the game has a pure NE, this algorithm performs well, because it terminates once a pure NE has been found. Otherwise, it tends to be among the slowest of the algorithms, because its running time is never faster than n^3, where n is the number of strategies. The quality of approximation results confirm that our new games have succeeded in avoiding pure strategy profiles that are close to being Nash equilibria.

- **TS2 and TS001.** The TS algorithm was the clear winner in terms of quality of approximation. The results show that the choice of δ can have a significant effect on the algorithm's characteristics. TS2 often terminates in a reasonable running time when compared to BBM, and it usually beats BBM significantly on quality of approximation. However, TS001 always beats TS2 in quality of approximation, and always provides the best approximations among all of the algorithms that we studied. This accuracy comes at the cost of speed, as there are many games upon which TS001 is much slower than TS2, and there are three classes of games upon which TS001 timed out for all instances.

Table 2.6 shows the results for the WSNE algorithms on the same set of games. Due to its running time, we were not too surprised to find that KS+ timed out on all instances, so we do not display results for this algorithm. The KS algorithm uses a linear programming based approach and also searches for the best WSNE that uses pure strategy profiles, and then outputs the best approximation provided by these two methods. We found that the pure WSNE part of the algorithm almost always provides the better approximation, and that the search over pure strategy profiles is the dominant component of KS's running time. Hence there is a fairly significant cost for targeting ϵ-WSNE over ϵ-Nash equilibria, since PURE is often among the slower algorithms for ϵ-Nash equilibria, and KS is never faster than PURE.

Finally, we comment on the trade off between running time and quality of approximation. In Figure 2.1 we plot these two metrics against each other for CovariantGame-9 and Ranking, which provide a fairly representative sample of the results that we observed

across the dataset. The points towards the lower left of the diagrams are those that minimize the running time for a given approximation guarantee. In order of accuracy, we typically see points from DMP, BBM1, TS2, and TS001 along this frontier.

The TS algorithm

Since our results indicate that the TS algorithm gives the best quality of approximation, it is worth spending more time analysing this algorithm. We have seen that both the quality of approximation and the running time of the algorithm are strongly affected by the choice of the parameter δ. In this section, we give more detailed results on how this parameter affects the characteristics of the algorithm. To test the dependence on δ, we ran the TS algorithm on one hundred 200×200 and 400×400 instances of the various games which we study, for various values of δ in the range $(0, 0.14]$. The results of these experiments are displayed in Figures 2.2 and 2.3. The left side of the figures shows the results for a single game, while the right side of the figures shows the average results over all instances.

The first two rows show the runtime and quality of approximation, respectively. It can be seen that the algorithm does not scale smoothly with respect to δ, and instead there are discontinuities in both running time and quality of approximation. The explanation for these discontinuities can be found in the third and fourth rows, which show the number of rows and the size of the LP that is solved in each iteration, respectively. The third row shows that, as we would expect, the number of iterations increases as δ decreases. However, the data in the fourth row shows that the story is more complicated. The size of the LP that is formulated in each iteration increases as δ increases. Thus, although the number of iterations falls, the time per iteration gets larger.

This pattern was not only spotted on RandomGame, but also for the different games which we study. Although these games have a similar pattern, the discontinuities occur at different values of δ and the duration between the discontinuities are not always the same either, with these discontinuities occurring outside the range shown above for several classes of games including GuessTwoThirdsAve and LocationGame.

2.8.4 Worst case examples

The best theoretical upper bound on the performance of the TS algorithm is that it produces a 0.3393-Nash equilibrium, but previous work has not been able to show a matching lower bound. As we have seen, the algorithm performs very well in practice, and usually finds a very good approximation. The Blotto games were the only classes of game that were

able to challenge the algorithm, and even then the approximations delivered were usually good. Figure 2.5 shows box and whisker plots for the quality of approximation of TS001 on the classes of Blotto games that we considered, the plots for the other games can be found in Figure 2.4. It can be seen that almost all points were close to 0.01, and only a handful of instances were larger. The worst approximation that we found was a 0.1544-Nash equilibrium, which is still far from the worst-case guarantee.

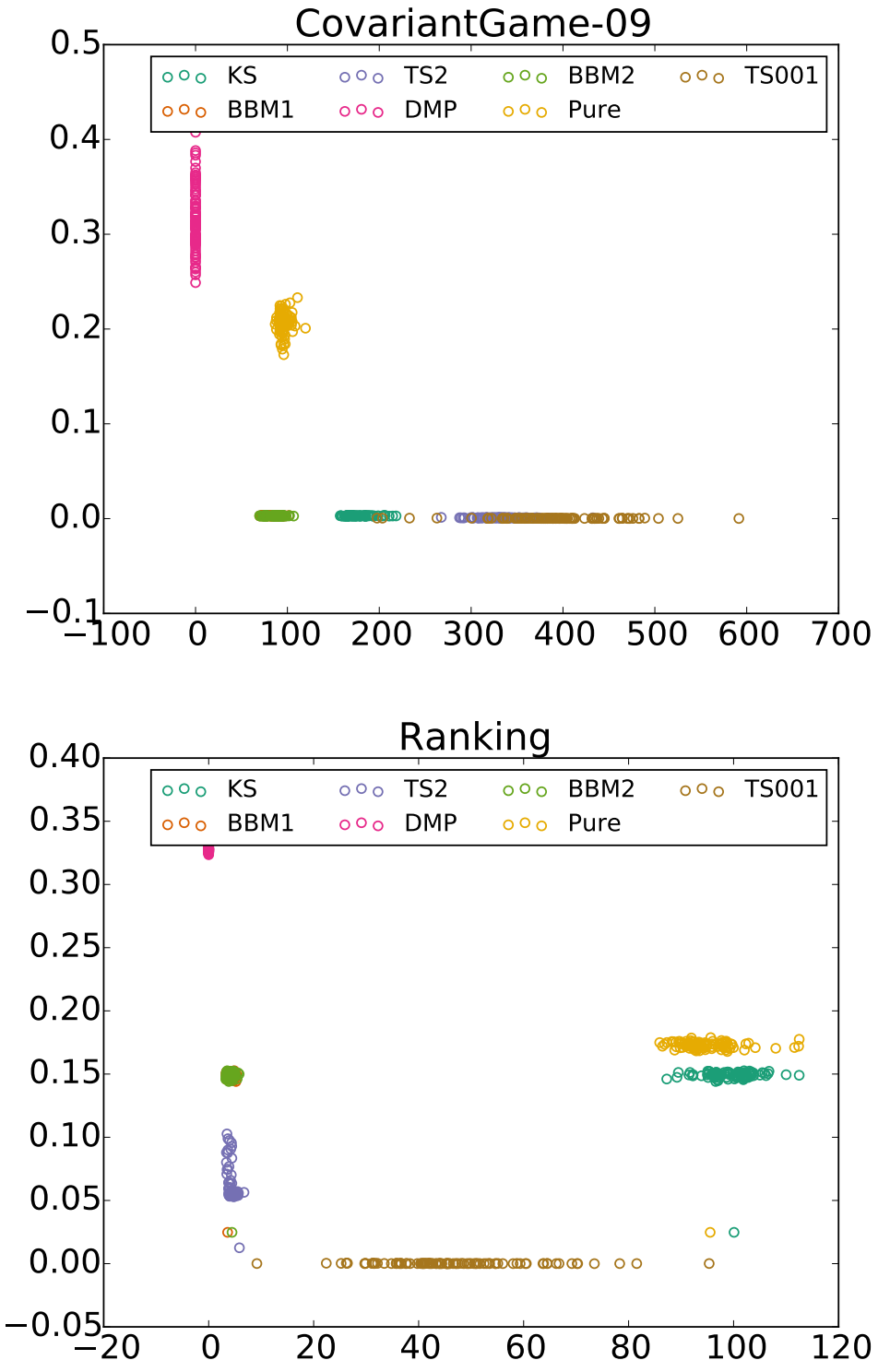


Figure 2.1: Runtime vs Quality of approximation for CovariantGame-9 for instances of size 2000×2000.

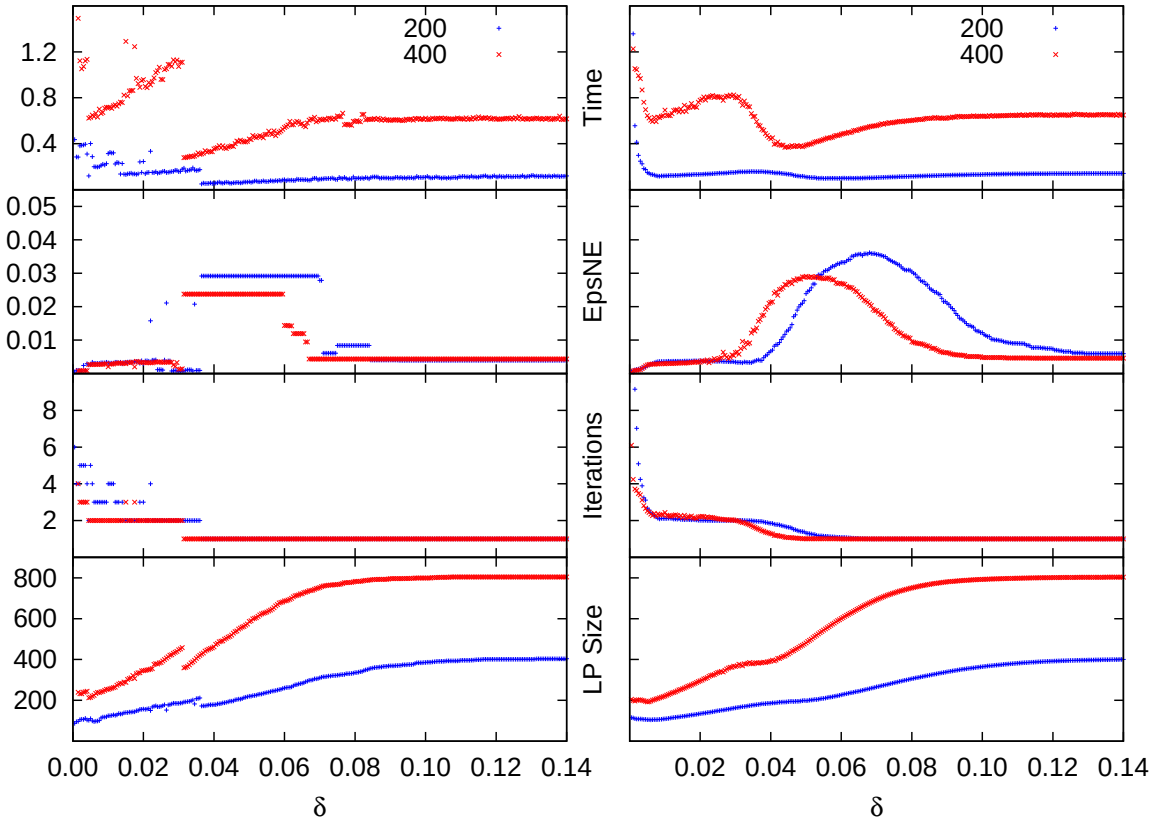


Figure 2.2: TS performance plots of runtime (row 1), quality of approximation (row 2), number of iterations (row 3,) and average LP-Size (row 4) against the value of δ. Left diagrams shows results for a single instance of RandomGame (400×400 in red and 200×200 in green), while right diagrams show averages over one hundred instances of RandomGame.

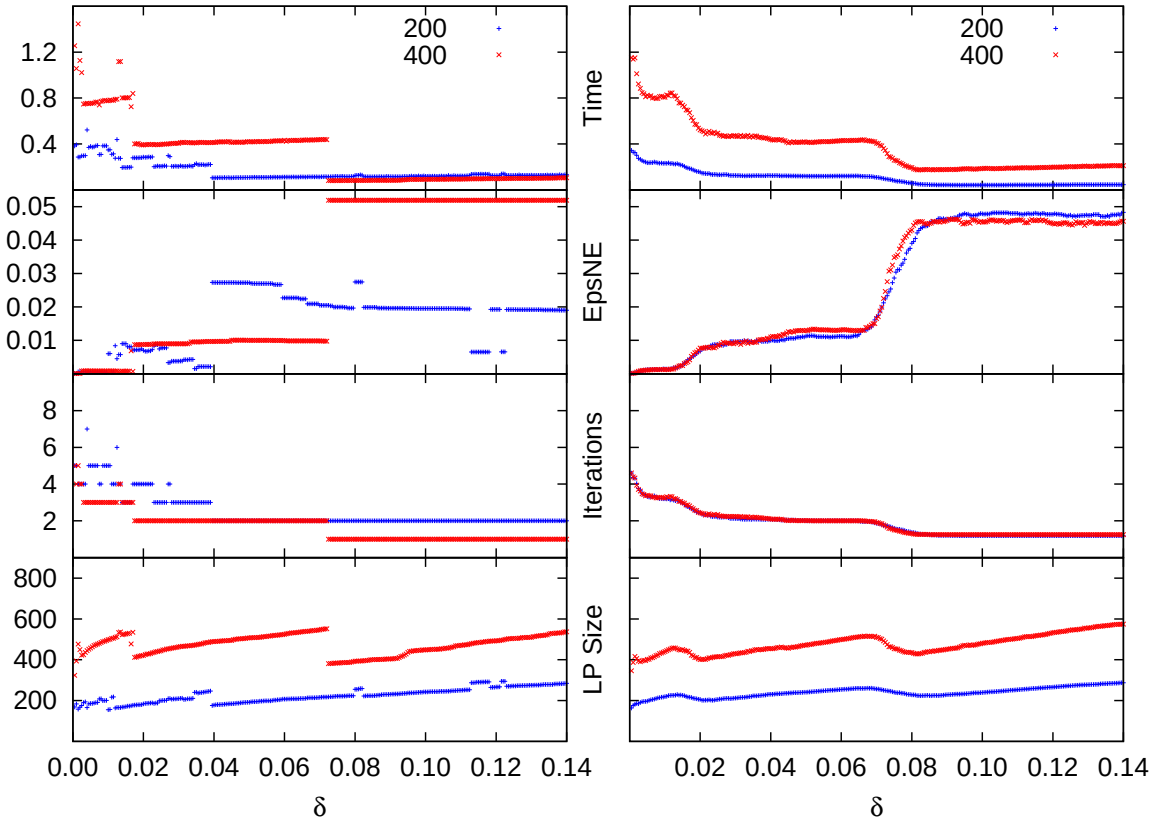


Figure 2.3: TS performance plots of runtime (row 1), quality of approximation (row 2), number of iterations (row 3,) and average LP-Size (row 4) against the value of δ. Left diagrams shows results for a single instance of Ranking, while right diagrams show averages over one hundred instances of Ranking.

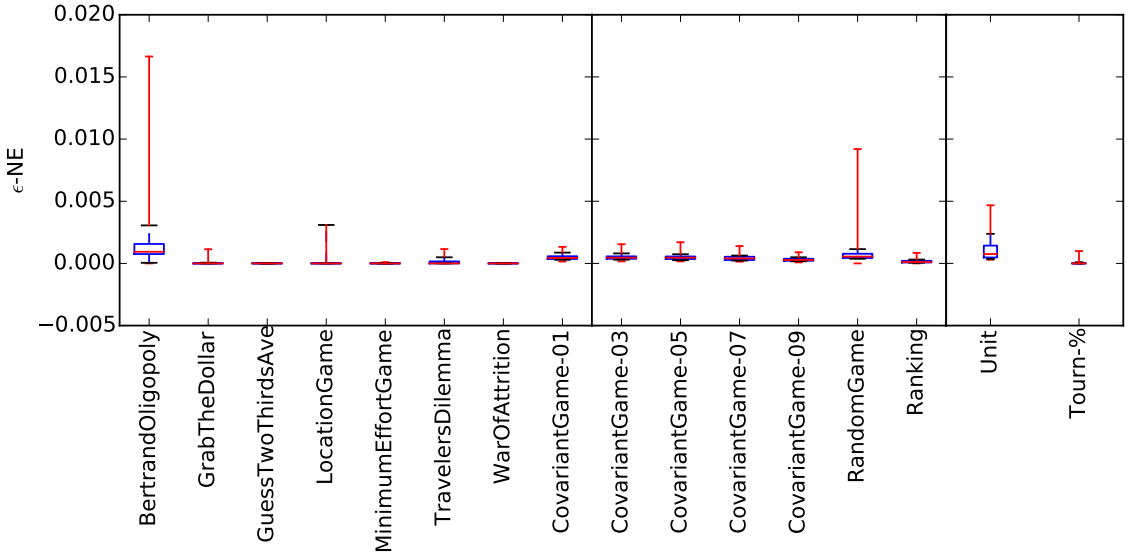


Figure 2.4: Box and whisker plots for the quality of ϵ-NE found by TS001.

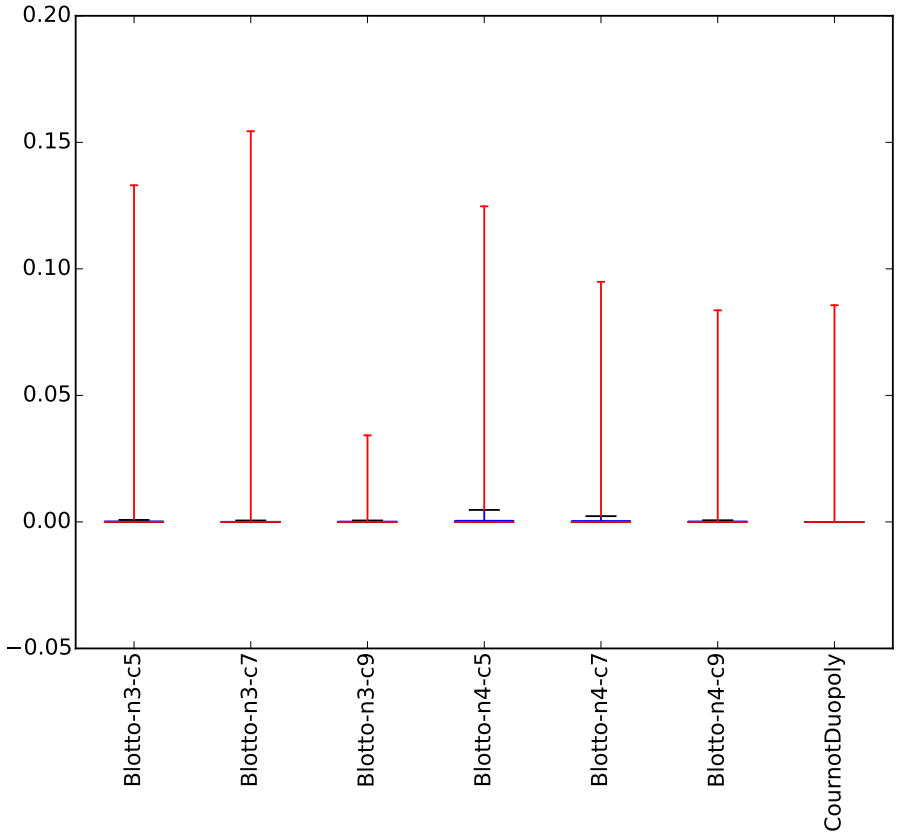


Figure 2.5: Box and whisker plots for the quality of ϵ-NE found by TS001 on Blotto games.

Chapter 3

Polymatrix Games

3.1 Introduction

In Chapter 2, we investigated bimatrix games, which are 2-player games represented in *strategic-form*. The strategic-form representation is one of the well-known game representations due to its ability to represent all possible outcomes of a game, by having a numerical payoff for each player, in each combination of strategy choices. However, as a result of the method of representation, the size of the representation grows *exponentially* in the number of players. For example, for a game with n players who can each choose between 2 strategies, $n \cdot 2^n$ payoffs must be specified. Hence, the strategic-form is typically unsuitable for certain instances which might occur.

Many realistic scenarios do not need the flexibility that strategic-form games provide. In particular, it is often the case that only *pairwise* interactions between players are important. In this chapter, we study *polymatrix games* which model this setting. In these games, the interaction between the players is specified as a graph. Each player plays an independent two-player game against each player that he is connected to, and the same strategy must be played in all of his games. A player's payoff is then the sum of the payoffs from each of the games. Crucially, the representations of these games grow *quadratically* in the number of players, which makes them more suitable for representing some large real-world scenarios.

While there has been a large amount of theoretical work on polymatrix games [22, 23, 32, 51, 55, 67], the practical aspects of computing equilibria in polymatrix games have yet to be studied. We provide in this chapter, an empirical study of two prominent methods for computing equilibria in these games. Firstly, we study *Lemke's algorithm*. The

Lemke-Howson algorithm is a famous algorithm for finding Nash equilibria in bimatrix games [86], and Lemke's algorithm is a more general technique for solving *linear complementarity problems* (LCPs). Miller and Zucker [92] have shown that the problem of finding a Nash equilibrium in a polymatrix game can be reduced to the problem of solving an LCP, which can then be tackled by Lemke's algorithm.

Secondly, we study a recently proposed gradient descent-like algorithm for finding approximate equilibria in polymatrix games [43], which is the only known approximation technique for (general) polymatrix games. It generalizes the algorithm of Tsaknakis and Spirakis (TS) for bimatrix games [121], and has an approximation guarantee of $0.5+\delta$, for a $\delta \in (0, 0.5]$. As stated in the previous chapter, we found that the TS algorithm typically finds high-quality approximate equilibria in practice [54], much better that its theoretical worst-case performance.

3.2 Preliminaries

Polymatrix games. An n-player polymatrix game is defined by an n-vertex graph. Each vertex represents a player. Each edge e corresponds to a bimatrix game that will be played by the players that e connects. Hence, a player with degree d plays d bimatrix games. More precisely, each player picks a strategy x_i and plays that strategy in *all* of the bimatrix games that he is involved in. His expected payoff is given by the sum of the expected payoffs that he obtains over all the bimatrix games that he participates in. We use $\mathbf{x} = (x_1, \ldots, x_n)$ to denote a strategy profile of an n-player game, where x_i denotes the mixed strategy of player $i \in [n]$.

Solution concepts. The standard solution concept for strategic-form games is the *Nash equilibrium* (NE). A relaxed version of this concept is the approximate NE, or ϵ-NE. Intuitively, a strategy profile is an ϵ-NE in an n-player game, if no player can increase his utility more than ϵ by unilaterally changing his strategy. To put it formally, let $\mathbf{x}$ denote a strategy profile for the players and let $u_i(z, \mathbf{x}_{-i})$ denote the utility of player i when he plays the strategy z and the rest of the players play according to $\mathbf{x}$. We say that $\mathbf{x}$ is an ϵ-NE if for every player i it holds that $u_i(x_i, \mathbf{x}_{-i}) \geq u_i(z, \mathbf{x}_{-i}) - \epsilon$ for all possible z. If $\epsilon = 0$, we have an exact Nash equilibrium.

3.3 Contribution

We provide a thorough empirical study of finding exact and approximate Nash equilibria in polymatrix games. We develop an extensive library of game classes that cover a number of applications of polymatrix games including cooperation games, strictly competitive games, and group-wise zero-sum games. We also study *Bayesian two-player games*, which can be modelled as polymatrix games. In particular, we focus on various forms of *Bayesian auctions* (e.g. item bidding combinatorial auctions) and Bayesian variants of Colonel Blotto games, which have applications in task allocation and resource allocation problems [118]. All our algorithm implementations and game generators are open source and publicly available[1], so that any new algorithms developed for polymatrix games can be tested against our test suite.

We study Lemke's algorithm and the descent method on all of the problems that we consider. In total we applied Lemke's algorithm to 188,000 instances using 26 months of CPU time, while we applied descent to 213,000 instances using 2.7 months of CPU time. We found that Lemke's algorithm can compute exact equilibria in relatively large games in a reasonable amount of time. Furthermore, we observe that the descent method is much more scalable and can be used to compute approximate equilibria for instances that are an order of magnitude bigger. Moreover, in contrast to its theoretical worst-case performance guarantee, the descent method typically finds very high quality approximate equilibria.

3.4 Related work

Although work on approximate equilibrium for polymatrix games is recieving some traction as of late, it has been studied a lot less in comparison to bimatrix games. There is the recent descent procedure introduced by Deligkas et. al. [43], and a recent QPTAS for polymatrix games on trees [11].

There are empirical studies on equilibrium computation both for exact equilibria [6, 7, 54, 104, 114] and approximate equilibria [54], but none of them focused on polymatrix games. Instead, these studies mainly focused on games created by GAMUT [98], the most famous suite of game generators. GAMUT has a generator for some simple polymatrix games, but it converts them to strategic-form games which blows up the representation exponentially.

Polymatrix games have received a lot of attention recently. Computing a Nash equi-

[1] `http://polymatrix-games.github.io/`

librium in a polymatrix game is PPAD-hard even when all the bimatrix games are either zero-sum or coordination games [20]. Recently, it was proven that there is a constant $\epsilon > 0$ such that it is PPAD-hard to compute an ϵ-Nash equilibrium of a polymatrix game [111]. Govindan and Wilson proposed a (non-polynomial-time) algorithm for computing Nash equilibria of an n-player strategic-form game, by approximating the game with a sequence of polymatrix games [67]. Later, they presented a (non-polynomial) reduction that reduces n-player games to polymatrix games while preserving approximate Nash equilibria [68].

Many papers have derived bounds on the Price of Anarchy [16,56,107] in item bidding auctions [8, 24, 47]. Only recently, Cai and Papadimitriou [21] and Dobzinski, Fu and Kleinberg [44] studied the complexity of the equilibrium computation problem in this setting. In situations where computation of equilibrium proves to be a significant challenge to handle, no-regret learning can be used to approximately compute learning outcomes in polynomial time [108]. However, it has been shown that there are no polynomial time no-regret learning alogrithms for second price item bidding auctions [41]. Best response dynamics on the other hand, have recently been shown to reach near optimal welfare within a short period of time [48]. Blotto games are a basic model of resource allocation, and have therefore been studied in the agents community [2, 103]. There have also been several papers that study Blotto games with incomplete information as well, see for example [1, 83].

Polymatrix games are an example of *graphical games*, which are succinct representations of games where interactions between players are encoded in a graph. A related succinct representation is that of Action Graph Games (AGGs); introduced by Bhat and Leyton-Brown, AGGs capture local dependencies as in graphical games, and partial indifference to other agents' identities as in anonymous games [15, 40, 73].

3.5 Algorithms

Lemke's algorithm is a complementary pivoting algorithm for the Linear Complementarity Problem (LCP) [85]. An LCP is given by an $n \times n$ matrix M and a vector q of size n, where the problem is to find a pair of vectors w, z such that:

$$w = q + Mz, \qquad w \geq \mathbf{0}, \qquad z \geq \mathbf{0}, \qquad z^{\mathrm{T}}w = 0.$$

Miller and Zucker have shown that finding a Nash equilibrium in a polymatrix game can be reduced in polynomial time to an LCP [92]. So, we first turn the polymatrix game into

an LCP, and then apply Lemke's algorithm. We shall refer to this algorithm as LEMKE.

LEMKE's algorithm is very similar to the Lemke-Howson algorithm, using the same complementary pivoting method. It has known that the Lemke-Howson algorithm is a special case of LEMKE's algorithm [115].

One drawback of this method, is that the reduction assumes a complete interaction graph even if the actual interaction graph is not complete (by padding with constant 0 payoffs games). Hence, for sparse polymatrix games the reduction introduces a significant blowup, which affects the performance of the algorithm. To make this blowup clear, in our results we report the number of payoffs in the original polymatrix game and the number of matrix entries in the resulting LCP.

Descent is a gradient descent-like algorithm, which was proposed in [43]. It aims to minimize the *regret* that a player suffers, which is the difference between the utility he gets by playing a best response and the actual utility he gets. The algorithm starts from an arbitrary strategy profile $\mathbf{x}$ and in each iteration it computes a new profile in which the maximum regret (over the players) has been reduced. The algorithm takes a parameter δ that controls how accurate the resulting approximate Nash equilibrium is. The theoretical results state that the algorithm finds a $0.5 + \delta$-NE after $O(\delta^{-2})$ iterations (for a fixed size game). We test the cases where δ is either 0.1 or 0.001, and we provide full results for both cases. We shall refer to this algorithm as DESCENT in our results.

For the strategy profile $\mathbf{x}$, the algorithm first computes a direction vector $(\mathbf{x}' - \mathbf{x})$, and then moves a certain distance in that direction. In other words, the algorithm moves to a new strategy profile $\mathbf{x} + \alpha(\mathbf{x}' - \mathbf{x})$, where α is some constant in the range $(0, 1]$. The theoretical analysis in [43] uses $\alpha = \frac{\delta}{\delta+2}$ in the proof of polynomial-time convergence. In practice we found that using larger step sizes greatly increases the speed of the algorithm. Hence, we adopt a *line search* technique, which we adapt from the two-player setting [122]. It checks a number of equally spaced points for α between 0 and 1, and selects the best improvement that is found. We also include $\alpha = \frac{\delta}{\delta+2}$ as an extra point in this check, so that the theoretical worst-case running time of the algorithm is unchanged. In our results, we check 200 different points for α in each iteration. For a justification for the efficacy of this choice, see Section 3.9.1.

3.6 Game classes

3.6.1 Bayesian Two-player Games

A two-player Bayesian game is played between a row player and a column player. Each player has a set of possible types. At the start of the game, each player is assigned a type by drawing from a publicly known joint probability distribution. Each player learns his type, but not the type of his opponent. Rosenthal and Howson showed that the problem of finding an exact equilibrium in a two-player Bayesian game can be reduced to finding an exact equilibrium in a polymatrix game [72], and this was extended to approximate equilibria in [43]. The underlying graph in the resulting polymatrix game is a complete bipartite graph where the vertices of each side represent the types of a player. More specifically, if the row player has n types and the column player has m types, the corresponding polymatrix game has $n + m$ vertices and the payoff matrix for edge (uv) corresponds with the payoff matrix of the Bayesian game where the row player has type u and the column player has type v. We study the following Bayesian two-player games.

Combinatorial auctions

In a combinatorial auction, m items are auctioned to n bidders. Each bidder has a valuation function that assigns a non-negative real number to every subset of the items. Notice that in this setting, in general, the size required to represent the valuation function is exponential in m. In combinatorial auctions with *item bidding*, each player bids for every item separately and all the items are auctioned simultaneously. A bidder wins an item if he submitted the highest bid for that item. The bidders pay according to a predefined payment rule. We study three popular payment rules. In a *first price* auction the winner of an item has to pay his bid for that item, in a *second price* auction the winner of an item has to pay the second highest bid submitted for the item, and in an *all pay* auction every bidder has to pay his bid *irrespective* of whether he won the item or not. If more than one bidder has the highest bid for some item, we resolve this tie according to a predefined publicly known rule. We study two tie-breaking rules: either we always favor one of the players, or we choose the winner for each item independently and uniformly at random.

We say that a combinatorial auction allows *overbidding* if a bidder is allowed to make a bid for a item greater than his value for it. A common assumption in the literature is that overbidding is not allowed, since allowing overbidding leads to the existence of trivial equilibria. Therefore, in our experiments we do not allow overbidding.

In a *Bayesian* combinatorial auction the valuation function for every player is chosen according to a commonly known joint probability distribution, which in this paper is always discrete. The different valuation functions that may be drawn for a player are known as his *types*.

Item bidding auctions

We create Bayesian combinatorial auctions with item bidding with two bidders, 2 to 4 items for sale and 2 to 5 different types per player. Each player's type (valuation function) is chosen uniformly at random. We study several well known valuation functions:

- *Additive*: the value of each bundle of items is the sum of the values of the items contained in the bundle.
- *Budget additive*: the value of each bundle is the minimum of a budget parameter and the sum of the values of the items contained in the bundle.
- *Single minded*: each bidder has positive value for a specific bundle of items (and the same value for any other bundle of items containing that bundle) and zero otherwise.
- *Unit demand*: the value of each bundle is the maximum value the bidder has for any single item contained in the bundle.
- *AND-OR*: the first bidder has positive value only for the grand bundle of items and zero otherwise, while the second one has a unit demand valuation.

To create the valuations, we set a maximum value $M \in \mathbb{N}$ that a player can have for any item, and a minimum value $m \in \mathbb{N}$ such that in every valuation there must be an item with a value of at least m. Bids are restricted to $\mathbb{N}$, so M and m define the number of pure strategies a player has, and consequently the size of the game. For a player with a single-minded valuation function, we choose a random subset of items and a random value for that subset in the range $[m, M]$. For additive and unit demand valuations, we randomly select a value for each item from the set of allowed valuations. The same procedure is extended for budget additive bidders: first we draw values for items as for additive bidders; then we draw the budget as a random number between M and N where N is the sum of the values for the items.

Multi-unit auctions

In these auctions all the items being sold are identical. When there are n items for sale, a valuation is given by an n-tuple $(v_1, \ldots, v_n)$, where v_j represents the player's marginal value for receiving a j-th copy of the item. Hence, the valuation for a bidder when he wins k items is the sum of the values v_1 up to v_k.

Again we study the three most common payment rules: the first price rule, a.k.a. the *discriminatory auction* [84], where a player that won k items has to pay the sum of his k highest bids; the second price rule, a.k.a. the *uniform-price auction* [84], where the price for every item is the market-clearing price, i.e. the highest losing bid; and the all-pay rule, where a player has to pay the sum of his bids.

We consider two well known valuation functions: additive, where $v_j = v_1$ for all $j > 1$ and submodular, where $v_j \geq v_{j+1}$ for all $j \in [n-1]$. We create games with 2 to 4 items and 2 to 5 different types per player. The sampling of non-additive sub-modular valuation functions is not described here; we refer the reader to the source code for further details[2].

Colonel Blotto games

In Colonel Blotto games, each player has a number of soldiers m that are simultaneously assigned to n hills. Each player has a value for each hill that he receives if he assigns strictly more soldiers to the hill than his opponent and any ties are resolved by choosing a winner uniformly at random. The value that a player has for a hill is generated independently uniformly at random and the payoff a player gets under a strategy profile is given by the sum of the value of the hills won by that player. We consider games with 3 and 4 hills and 3 to 15 soldiers per player. We study two different Bayesian parameters: the valuations of the players over the hills and the number of soldiers that each player has. In the game we looked at, only one of these two parameters was used (i.e. for the other there was complete information). For both cases we study games with 2 to 4 types per player and the probability that a particular type occurs is uniform over the number of types. When the types correspond to different valuations for hills, for every type, the valuations for each hill were drawn from i.i.d. uniform distributions on $[0, 1]$. When the types correspond to the number of soldiers, we drew independently from $\{3, \ldots, 15\}$ for each player and each type.

[2] `http://polymatrix-games.github.io/`

Adjusted Winner games

The adjusted winner procedure fractionally allocates a set of n divisible items to two players [18]. Under the procedure, $n-1$ items stay whole and at most one is split between the players. Each player has a non-negative value for each item, and these values sum to 1. Both players have additive valuations over bundles of items. For a split item that a player has value v for, if a player receives $w \in [0,1]$ of the item, he gets gets $w \cdot v$ value from this part of the item.

The players simultaneously assign m points to the items. Suppose player 1 assigns α_i points for the items $i = 1, \ldots, n$ with $\sum_i \alpha_i = m$ for player 1, and similarly player 2's assignment is $(\beta_1, \ldots, \beta_n)$. The procedure starts with an initial allocation in which each item goes to one of the players that assigned most points to it. If the players get equal utilities it stops. Otherwise it next determines which player gets higher utility under this allocation (say player 1). In the next step, it finds the item i that is currently allocated to player 1 and has the smallest ratio α_i/β_i. If possible it splits this item in a such a way as to equalize the total utilities of the two players, or, if not, completely reallocates this item to player 2, and repeats this step until the utilities of the two players are equalized. Thus, at most one item is actually split.

We study the cases of 2 to 4 items and the players with between 3 and 15 points to assign. For every type, the valuations for each item were drawn from i.i.d. uniform distributions on $[0,1]$, and then normalized to sum up to 1.

3.6.2 Multi-player Polymatrix Games

We study several types of game, and for each, we study a range of underlying graphs: complete graphs, cycles, stars, and grid graphs. In each case the entries of the payoff matrices are chosen independently and uniformly at random from $[0,1]$.

- **Net coordination games.** In these games, every edge e corresponds to a bimatrix game (A_e, A_e). These games possess a pure NE, which is PLS-complete to compute and the complexity of finding a (possibly non-pure) exact equilibrium is in PLS $\cap$ PPAD [20].

- **Coordination/zero-sum games.** Here each edge s either a coordination or zero-sum game, i.e. on edge e the bimatrix game is (A_e, A_e), or $(A_e, -A_e)$. These games are PPAD-complete [20] to solve. We create games having a proportion p

of coordination games for $p \in \{0, 0.25, 0.5, 0.75, 1\}$. We study how p affects the running time of the algorithms.

- **Group-wise zero-sum games.** The players are partitioned into groups so that the edges going between groups are zero-sum while those within the same group are coordination games. In other words, players inside a group are "friends" who want to coordinate their actions, while players in different groups are competitors. These games are PPAD-complete [20] to solve even when there are 3 groups. We create games with 2 and 5 groups, all played on complete graphs. In every case, each group is approximately the same size, and each player is assigned to a group at random.

- **Strictly competitive games.** A bimatrix game is strictly competitive if for every pair of mixed strategy profiles s and s' we have that: if the payoff of one player is better in s than in s', then the payoff of the other player is worse in s than in s'. We study polymatrix games with strictly competitive games on the edges. These games are PPAD-complete [20].

- **Weighted cooperation games.** The unweighted version of these games was introduced in [4]. There each player chooses a colour from a set of available colours. The payoff of a player is the number of neighbours who choose the same colour. These games have a pure NE that can be computed in polynomial time. We study the more general case where each edge has a positive weight. The complexity for the weighted case is unknown [38]. We create games where all players have the same number of pure strategies (i.e. available colours) k, where k is in $\{15, \ldots, 45\}$. For every player, his available colours are chosen uniformly at random from all k-sized subset from a universe of colours that is of size either $2k$ or $5k$.

3.7 File Format

The closest generator for polymatrix games with the exception of this library is the GAMUT library, which has a generator for random polymatrix games. However, due to a lack of a standard file format for polymatrix games, the game gets converted to an n-player normal form game before being output. In this section we present the file format used for both the generators and algorithm implementations.

Consider the polymatrix game represented in Figure 3.1; this is represented as a matrix of the form shown in Figure 3.2. As can be seen from the representation, a significant portion of the matrix is covered by entries of 0 due to there not being an edge between two nodes.

The file representation of a polymatrix game is split into 2 main sections separated by whitespaces. Although not currently implemented, in the future, we plan on including comments (which would be lines starting with a # symbol) into the file format to aid with readability.

1. The number of players, n.
2. An adjacency matrix representing the graph G.
3. For all pairs of players (i, j) in lexicographic order
   (a) The number of strategies for both players, $|s_i||s_j|$
   (b) The payoff matrices for both players $A_{ij}A_{ji}$

The full file representation of the game shown in figure 3.1 can be seen in Figure B.3.

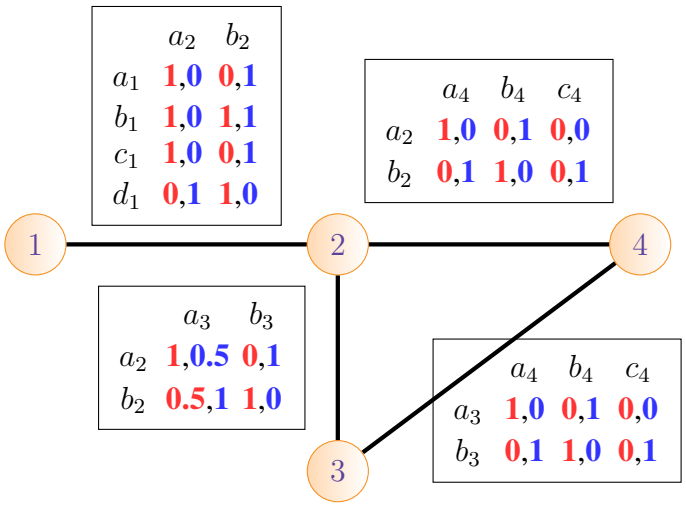


Figure 3.1: An example of a 4 player polymatrix game

3.8 Experimental setup

The algorithms and game generators were implemented in C. The CPLEX library was used for solving LPs in the implementation of DESCENT. Our implementation of LEMKE uses

	a_1	b_1	c_1	d_1	a_2	b_2	a_3	b_3	a_4	b_4	c_4
a_1	0	0	0	0	1	0	0	0	0	0	0
b_1	0	0	0	0	1	1	0	0	0	0	0
c_1	0	0	0	0	1	0	0	0	0	0	0
d_1	0	0	0	0	0	1	0	0	0	0	0
a_2	0	0	0	1	0	0	1	0	1	0	0
b_2	1	1	1	0	0	0	0.5	1	0	1	0
a_3	0	0	0	0	0.5	1	0	0	1	0	0
b_3	0	0	0	0	1	0	0	0	0	1	0
a_4	0	0	0	0	0	1	0	0	0	0	0
b_4	0	0	0	0	1	0	0	0	0	0	0
c_4	0	0	0	0	0	1	0	0	0	0	0

Figure 3.2: Representation of the polymatrix game in figure 3.1

integer pivoting in exact arithmetic using the GMP library; we were unable to produce a numerically stable floating point implementation (generally our attempts would start to fail on LCP instances of dimension 60).

In our results, the average runtime of the algorithms, as well as the approximation guarantee found, include the instances which timed out. The experiments were carried out on a cluster of 8 machines which had Intel Core i7-2600 CPU's clocked at 3.40GHz, running Scientific Linux 6.6 with Linux kernel version 2.6.32.

3.9 Results

We ran both LEMKE and DESCENT on all of our input instances. Table 3.1 shows the results for auctions, Table 3.2 shows the results for other Bayesian two-player games, and Table 3.5 shows the results for multi-player polymatrix games. While we tested games of many different sizes, for the purposes of exposition, the tables display the largest instances that LEMKE can solve without timing out.

One general feature of our results is that DESCENT is much faster than LEMKE. To illustrate this, Figure 3.3 shows the performance of the two algorithms on the three types of additive item-bidding auctions included in the study. It can be seen that on the hard instances (first price and all pay), LEMKE starts to struggle when there are around five million payoffs in the game, whereas even the slower and more accurate of the two DESCENT variants ($\delta = 0.001$) can handle games with 30 million payoffs in under a minute. Indeed, a runtime regression for Bayesian Blotto games found that LEMKE has rouglhy quadratic running time (with an R-squared of 0.75 for the regression), while DESCENT

Games				LEMKE			DESCENT 0.1 LS		DESCENT 0.001 LS	
	Valuation	Avg. Size	Auc	Time	% Timeout	% Pure	Time	ϵ	Time	ϵ
Itembidding	Additive	623990	FP	35.005	0.0	0.0	0.200	1.133e-02	1.287	2.630e-04
			SP	0.764	0.0	100.0	0.233	7.103e-03	0.232	2.312e-05
			AP	214.049	12.0	0.0	0.289	6.163e-03	1.610	1.943e-04
	Unit	650417	FP	46.351	0.0	34.0	0.199	6.249e-02	3.336	3.063e-02
			SP	203.107	19.0	58.0	0.459	1.442e-02	1.831	2.647e-04
			AP	14.709	0.0	21.0	0.168	4.182e-02	2.757	1.113e-02
	AndOr	519055	FP	280.322	23.0	0.0	0.194	1.927e-02	1.732	1.594e-04
			SP	1.269	0.0	100.0	0.279	7.026e-03	0.595	1.019e-04
			AP	166.011	9.1	0.0	0.171	8.662e-03	1.328	2.842e-04
	Budget	647583	FP	100.444	5.0	3.16	0.206	3.055e-02	2.003	4.064e-03
			SP	9.438	1.0	84.85	0.348	2.543e-02	0.915	2.186e-04
			AP	248.739	24.0	0.0	0.271	1.990e-02	2.199	2.233e-03
	SingleMinded	606511	FP	82.166	7.37	5.0	0.139	2.775e-02	1.551	1.822e-03
			SP	110.341	17.0	48.19	0.475	1.045e-02	1.359	1.669e-04
			AP	59.942	3.0	1.0	0.162	1.856e-02	1.544	1.038e-03
Multiunit	Additive	836465	D	9.504	0.0	30.0	0.295	1.452e-02	1.481	4.411e-04
			U	0.954	0.0	100.0	0.380	1.321e-02	1.870	3.579e-04
			AP	512.564	64.0	0.0	0.417	4.080e-03	2.182	3.170e-04
	SubModular	878491	D	29.054	0.0	5.0	0.270	1.326e-02	1.954	3.209e-04
			U	5.818	0.0	83.0	0.272	2.636e-02	1.741	8.645e-04
			AP	210.290	12.0	0.0	0.323	1.115e-02	2.192	3.273e-04

Table 3.1: Results for item bidding and multi-unit auctions, with 3 items and 3 types per player. Ties are broken by favouring the second player in item bidding, and by allocating the item uniformly at random in the multi-unit case. For LEMKE, we report the average running time, the percentage of instances that exceeded our timeout of 10 minutes, and the percentage of instances for which the algorithm finds a pure equilibrium. For DESCENT, we report the average running time and the quality of the approximation that was found.

Games			LEMKE		DESCENT 0.1 LS		DESCENT 0.001 LS		
Game	# Types	# Troops	Time	% Timeout	Time	ϵ	Time	ϵ	% Timeout
AdjWinner	30	5	281.407	21.0	0.939	4.450e-02	3.065	7.469e-03	0.0
	3	60	233.963	10.0	138.584	2.683e-02	220.254	1.827e-03	10.0
Blotto	3	8	71.814	0.0	0.032	9.181e-03	0.480	5.281e-04	0.0
		10	382.497	23.0	0.063	8.859e-03	0.845	5.408e-04	0.0
		12	573.663	91.0	0.118	8.590e-03	1.392	6.268e-04	0.0

Table 3.2: Results for Adjusted winner and Blotto games. The two rows for Adjusted winner show similar running times but actually correspond to very different input sizes, with the second row having being much bigger. The underlying reason is that the number of players in the polymatrix (i.e.m number of types in the Bayesian game) affects the running time much more that the number of actions (i.e. number of items/troops). See Section 3.9.2 for further illustration of this point.

Auc	Player 1		Random	
	Time	% Timeout	Time	% Timeout
FP	63.411	0.0	408.223	43.0
SP	3.663	0.0	39.727	3.0
AP	100.949	4.0	248.665	19.0

Table 3.3: Results for LEMKE showing the impact of the tie-breaking rule. We report on first price (FP), second price (SP) and all-pay (AP) auctions with budget additive valuations, and 3 items and 5 types per player. In the first two columns, all tied items are allocated to player 1, while in the last two, tied items are allocated uniformly at random.

	LEMKE	DESCENT 0.001		DESCENT LS 0.001	
	Time	Time	ϵ	Time	ϵ
FP	229.4	508.0	5.4e-03	4.894	6.768e-04
SP	1.6	470.8	3.7e-03	0.491	1.013e-05
AP	547.3	496.0	6.5e-03	5.551	3.283e-04

Table 3.4: Results showing the impact of line search for DESCENT. We report results for first price (FP), second price (SP), and all-pay (AP) auctions for additive bidders. LEMKE timed out on 13% and 44.5% of FP and AP auctions respectively, while DESCENT without line search times out on 61.5%, 56 % and 94% of instances on the respective auctions.

has linear running time (with an R-squared of 0.88 for the δ values of 0.1 and 0.001 respectively).

However, good runtime performance for the approximation algorithm would be useless if it found poor quality approximate equilibria. Fortunately, our results show that this is not the case. In almost all experiments DESCENT found high quality approximate equilibria. The variant with $\delta = 0.1$ typically found an ϵ-NE with $\epsilon \leq 0.05$, while the variant with $\delta = 0.001$ typically found an ϵ-NE with $\epsilon \leq 0.002$.

Figures 3.4 and 3.5 shows a box and whisker plot for the quality of approximate Nash equilibrium found by the two variants of DESCENT which we study. It can be seen that even the worst case performance of the algorithm is relatively good for several classes of game. The overall worst approximation was a 0.1065-NE that was found on a weighted cooperation game. While this is far larger than the average performance, this is still much smaller than the theoretical worst case of 0.5.

We now make more detailed observations about the specific classes of games that we tested. For auctions, one interesting observation is that on certain classes of auctions LEMKE will often find a pure Nash equilibrium. This is shown in the % Pure column of Table 3.1. This phenomenon is particularly prevalent for second-price auctions, where in some cases we found that LEMKE always finds a pure NE, and in these cases it does so in a very small amount of time.

Another interesting thing that we discovered is that the tie-breaking rule used in the

Games					LEMKE		DESCENT 0.1 LS		DESCENT 0.001 LS		
Game	Graph	# Payoff	LCP	p	Time	% T	Time	ϵ	Time	ϵ	% T
Coord-Zero	Complete	26010	32400	0	1.270	0.0	0.034	2.103e-02	0.760	9.951e-04	0.0
				0.25	63.407	4.0	0.033	2.115e-02	0.748	1.026e-03	0.0
				0.5	337.443	45.0	0.034	1.859e-02	0.750	1.070e-03	0.0
				0.75	522.207	74.0	0.033	1.604e-02	0.725	1.076e-03	0.0
				1	116.354	0.0	0.034	4.844e-03	0.598	5.087e-04	0.0
	Cycle	25920	136900	0	18.430	2.0	0.103	3.352e-02	3.612	1.093e-03	0.0
				0.25	184.451	21.0	0.103	3.157e-02	3.534	1.167e-03	0.0
				0.5	412.947	55.0	0.105	2.859e-02	3.430	1.136e-03	0.0
				0.75	593.414	96.0	0.103	2.626e-02	3.206	1.121e-03	0.0
				1	600.097	100.0	0.107	1.906e-02	2.712	6.557e-04	0.0
	Grid	26136	93636	0	35.447	3.0	0.072	3.143e-02	2.257	1.023e-03	0.0
				0.25	260.455	35.0	0.069	3.239e-02	2.233	1.137e-03	0.0
				0.5	451.699	61.0	0.072	2.955e-02	2.254	1.170e-03	0.0
				0.75	552.286	82.0	0.072	2.786e-02	2.106	1.186e-03	0.0
				1	599.349	99.0	0.070	2.159e-02	1.802	6.489e-04	0.0
	Tree	25992	152100	0	0.276	0.0	0.060	1.012e-02	0.818	1.175e-03	0.0
				0.25	0.542	0.0	0.062	1.997e-02	0.806	1.220e-03	0.0
				0.5	73.443	5.0	0.062	2.139e-02	0.814	1.246e-03	0.0
				0.75	165.418	4.0	0.061	2.150e-02	0.796	1.084e-03	0.0
				1	162.420	0.0	0.063	1.469e-03	0.778	7.686e-04	0.0
Group Zero	Complete	20250	25600	2	368.032	36.0	0.025	1.976e-02	0.564	1.093e-03	0.0
				3	495.919	66.0	0.025	1.762e-02	0.550	1.129e-03	0.0
				5	435.207	29.0	0.025	1.308e-02	0.525	9.926e-04	0.0
		26010	32400	2	438.650	59.0	0.034	1.855e-02	0.760	1.068e-03	0.0
				3	576.439	91.0	0.034	1.583e-02	0.731	1.120e-03	0.0
				5	582.924	88.0	0.033	1.186e-02	0.677	9.738e-04	0.0
		36000	44100	2	545.997	84.0	0.052	1.564e-02	1.073	1.049e-03	0.0
				3	598.616	99.0	0.051	1.396e-02	1.037	1.110e-03	0.0
				5	600.088	100.0	0.051	1.101e-02	0.969	9.721e-04	0.0
Strict	Complete	20250	25600	5	356.009	17.0	0.024	1.878e-02	0.552	1.054e-03	0.0
	Cycle	20480	108900	5	580.891	85.0	0.087	1.729e-02	2.102	1.068e-03	0.0
	Grid	20184	72900	5	551.795	77.0	0.066	1.612e-02	1.428	1.108e-03	0.0
	Tree	20808	122500	5	79.560	0.0	0.048	2.571e-03	0.664	8.111e-04	0.0
Weighted Cooperation	Complete	995000	1440000	2	194.233	8.0	8.455	1.446e-02	86.116	9.603e-04	0.0
				3	410.118	38.0	6.551	1.909e-02	73.485	1.047e-03	0.0
				5	552.583	81.0	4.957	2.585e-02	64.676	1.130e-03	0.0
	Cycle	17500	4410000	2	103.403	0.0	0.227	1.334e-01	577.984	3.094e-02	73.0
				3	90.062	0.0	0.172	1.412e-01	529.581	4.199e-02	48.0
				5	78.883	0.0	0.156	1.427e-01	438.707	3.950e-02	28.0
	Grid	27200	3006756	2	116.157	0.0	0.864	8.298e-02	275.839	5.461e-03	1.0
				3	81.933	0.0	0.464	1.110e-01	480.608	1.368e-02	15.0
				5	58.054	0.0	0.131	1.384e-01	358.662	3.028e-02	2.0
	Tree	24950	9000000	2	240.215	0.0	0.750	0.000e+00	2.768	3.253e-04	0.0
				3	220.510	0.0	0.709	0.000e+00	2.533	1.194e-04	0.0
				5	204.919	0.0	0.653	0.000e+00	2.345	8.947e-05	0.0

Table 3.5: Results for non-Bayesian polymatrix games, conducted on complete, cycles, grids and star graphs. $\%T$ is the proportion of the timed out instances. On Cooperation-Zerosum games, the value of p represents the proportion of games which are coordination games, for group zero-sum games, it represents the number of groups, and for weighted cooperation games, it represents the multiplier dictating the total number of colours available, i.e. if there are k colours per player, then there are $k \cdot p$ total colours available.

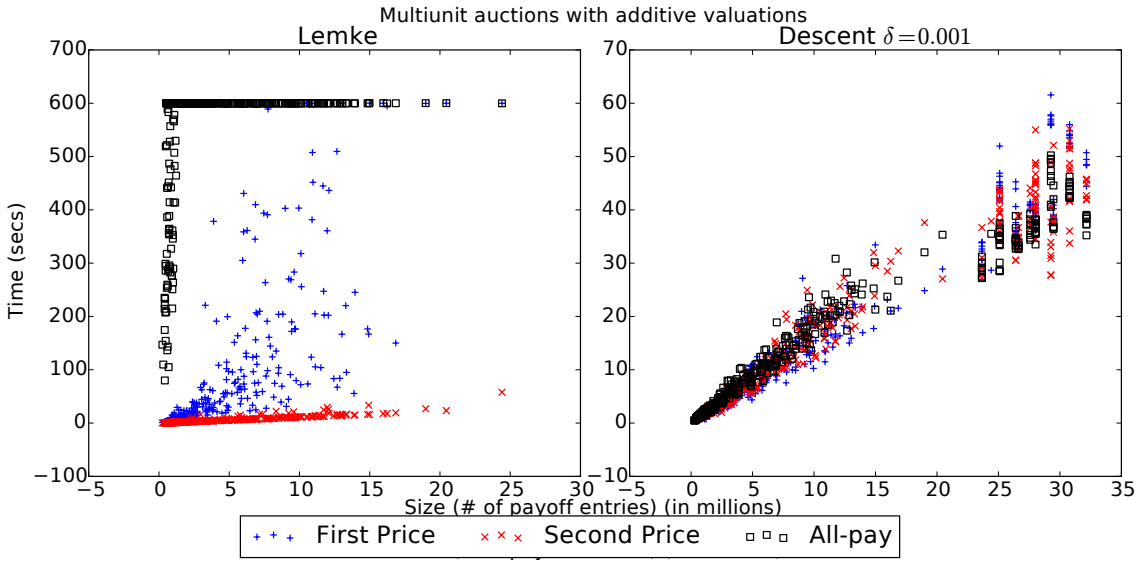


Figure 3.3: Plots showing the performance of our algorithms on multi-unit auctions with first price, second price, and all-pay payment methods. The chart of the left shows the performance of LEMKE's algorithm, where it can clearly be seen that the allocation rule impacts the performance of the algorithm. The chart on the right shows the performance of the DESCENT method with $\delta = 0.001$. The y-axis scales on the two charts are not equal: DESCENT is much faster than LEMKE.

auction can have a huge impact on the time that LEMKE takes to find an exact equilibrium. Table 3.3 shows the performance of the algorithm on otherwise identical auctions with different tie-breaking rules. It can be seen that resolving ties deterministically makes the game much easier to solve than resolving ties randomly.

Finally, we report on the difference between the implementation of DESCENT with line search and without. The results are shown in Table 3.4. It can be seen that, without line search, the more accurate DESCENT algorithm is often slower than LEMKE, and that using line search greatly speeds up the algorithm. This justifies the use of the line-search throughout the rest of our results. Interestingly, the line-search variant of the algorithm also finds more accurate approximate equilibria; it would be interesting to understand why this is the case.

3.9.1 Line Search

As noted in Section 3.9, the line search procedure for DESCENT greatly affects the running time of the running time of the algorithm. In this section, we give some results that describe this impact.

Recall that the line search procedure selects a number of equally spaced points for α

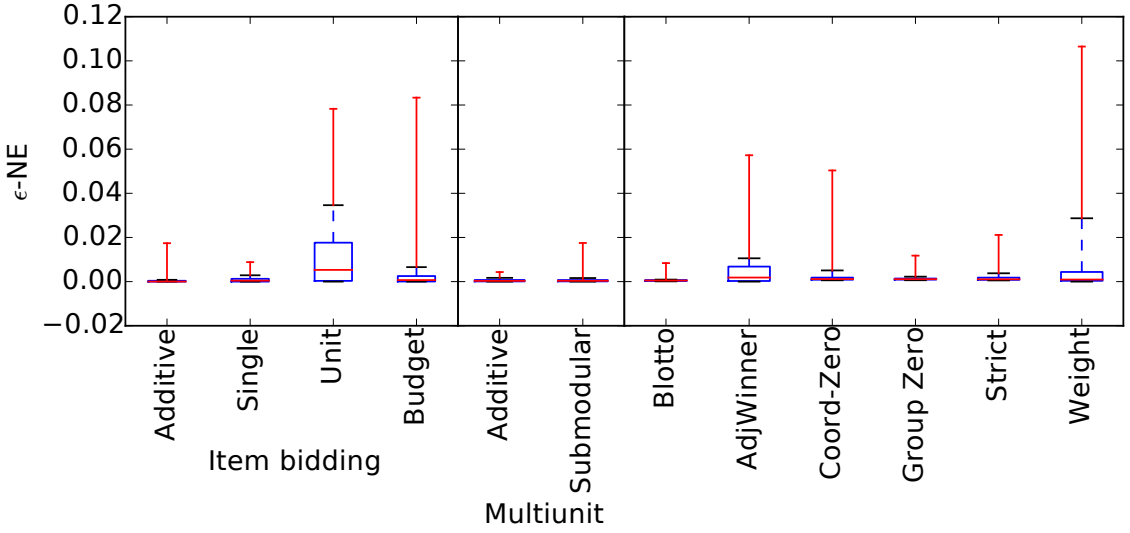


Figure 3.4: Box and whisker plots showing the quality of the approximations found by the DESCENT algorithm with $\delta = 0.001$. The results show that the algorithm almost always finds a high quality approximate equilibrium. It can be seen that on many classes of games, even the worst case approximation found by the algorithm is very good.

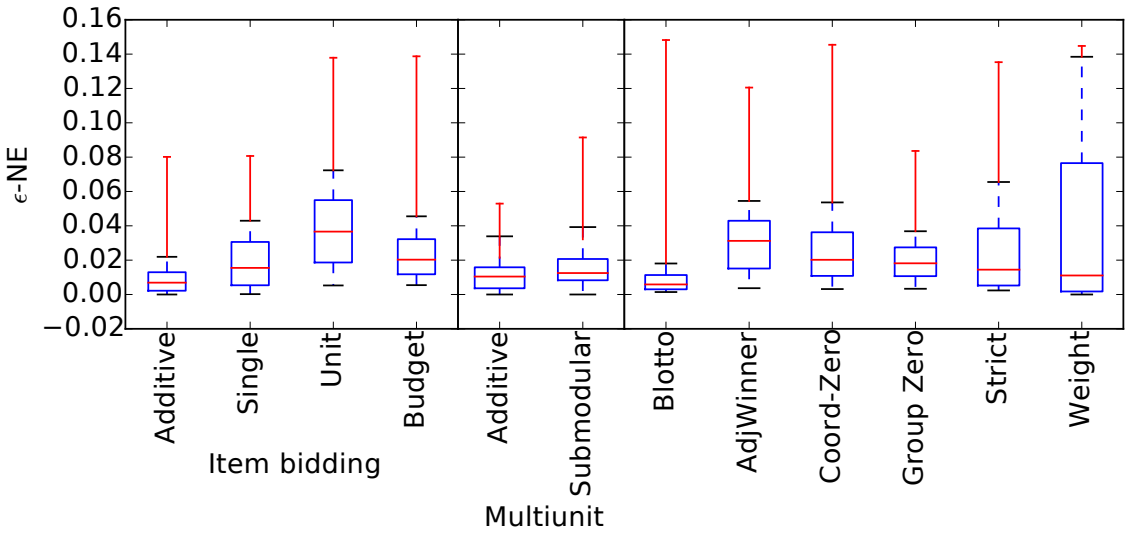


Figure 3.5: Box and whisker plots showing the quality of the approximations found by the DESCENT algorithm with $\delta = 0.1$. The results show that the algorithm almost always finds a high quality approximate equilibrium. It can be seen that on many classes of games, even the worst case approximation found by the algorithm is very good.

between 0 and 1, also including $\frac{\delta}{\delta+1}$, and the key decision is how many points to check. In order to study the effects that the number of points on line search has on the algorithm, we took the average over 100 random instances on Coordination/Zero-sum games on cycles and grids as well as Weighted Zero-sum games on Cycles.

Figure 3.6 shows the results. The red lines show how the overall running time of the algorithm varies depending on the number of points checked in each iteration, while the green lines show how the number of iterations change. As expected, the number of iterations tend to decrease as the number of points checked in each iteration increases. In other words, checking more points increases the amount of progress that each iteration makes. However, checking more points in each iteration also adds more computational cost, and eventually the overall running time begins to rise, which indicates that the cost of checking more points is too high relative to the extra progress that is made.

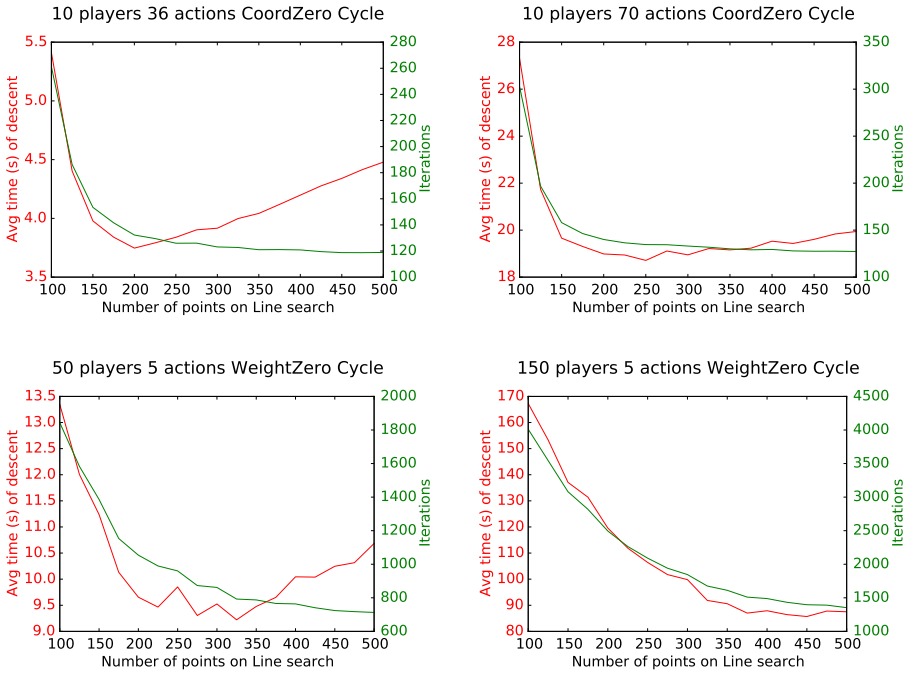


Figure 3.6: Plots showing the effects of the number of points on line search on the number of iterations and computation time of DESCENT

3.9.2 Game size: Players vs Actions

The size of a polymatrix game increases with more players and with more actions. In general, we found that the number of players had a much greater effect on the running

time of the algorithms than the number of actions per player. Table 3.6 illustrates this point.

Games					LEMKE		DESCENT 0.1 LS		DESCENT 0.001 LS		
Players	Actions	# Payoff	LCP	p	Time	% T	Time	ϵ	Time	ϵ	% T
10	105	992250	1060	2	3.113	0.0	0.301	7.152e-04	12.586	8.502e-04	0.0
		992250	1060	3	1.933	0.0	0.262	5.853e-04	13.704	8.381e-04	0.0
		992250	1060	5	1.430	0.0	0.198	6.931e-04	15.256	7.425e-04	0.0
200	5	995000	1200	2	194.233	8.0	8.455	1.446e-02	86.116	9.603e-04	0.0
		995000	1200	3	410.118	38.0	6.551	1.909e-02	73.485	1.047e-03	0.0
		995000	1200	5	552.583	81.0	4.957	2.585e-02	64.676	1.130e-03	0.0

Table 3.6: Table showing Weighted Cooperation games on Complete graphs relatively close in size, comparing the case of a large number of players and few actions with the case of a few players with larger action sets. $\%T$ is the proportion of the timed out instances.

Chapter 4

Lower Bounds on Approximation Guarantees

4.1 Introduction

In the earlier chapters of this thesis, we have discussed the upper bounds and studied empirically the approximation guarantees which several approximation algorithms give us. We have yet, however, to discuss lower bounds. Unlike upper bounds, which give us a sense as to the limits of the algorithm, lower bounds help to paint a clearer picture by providing an actual instance where the performance of the algorithm in question is at its worst.

Starting with the simplest approximation algorithm, in constant-time a 1-NE can be achieved by selecting an arbitrary pure strategy profile. A simple game of matching pennies (as depicted in Figure 4.1) provides a lower bound for this basic method. Dedicating extra time by looking for the pure strategy profile with the best approximation does not affect this result. Regardless of the starting point, there is always going to be a player with regret of 1 whenever a pure strategy profile is played. This game also gives us a tight lower bound on the DMP algorithm, leading to a 0.5-NE for every possible starting point of the algorithm.

When Bosse, Byrka and Markakis [17] introduced two approximation algorithms, they also gave an in-depth analysis of their algorithms, along with a lower bound (shown in Figure 4.2) which the BBM2 algorithm yields a 0.36392-NE, showing the algorithm to be tight. Finally, for the KS algorithm, we are aware of two instances (shown in Figure 4.3), which were used to show that the KS algorithm is tight [53].

I \ II	*Head*	*Tail*
Head	1, 0	0, 1
Tail	0, 1	1, 0

Figure 4.1: Bimatrix game representation of the game of Matching Pennies

I \ II	1	2	3
1	0, 0	α, α	α, α
2	α, α	1, 0	$\frac{1}{2}$, 1
3	α, α	$\frac{1}{2}$, 1	1, 0

Figure 4.2: Tight lower bound for the BBM algorithm where $\alpha = 1/\sqrt{6}$

I \ II	*l*	*r*
T	1, $\frac{1}{3}$	$\frac{1}{3}$, 1
B	0, 0	0, 0

I \ II	ℓ	*r*
T	1, $\frac{1}{3}$	$\frac{1}{3}$, 1
M	$\frac{1}{3}$, 1	1, $\frac{1}{3}$
B	0, 0	0, 0

Figure 4.3: Tight lower bound for the KS algorithm

Prior to our empirical studies shown in Chapter 2 and Chapter 3, the worst known instance for the TS algorithm resulted in a 0.015-NE where $\delta = 0.001$. We were able to improve upon this with the analysis shown in Chapters 2 and 3, by finding instances which resulted in a 0.1544-NE for the TS algorithm, whereas for polymatrix games with more than 2 players the worst case instance gave a 0.1065-NE.

4.2 Contribution

The excellent performance of the TS [121] and DESCENT [43] algorithms, as observed in the empirical studies conducted in Chapters 2 and 3, prompt the question as to whether the upper bounds (of 0.3393 and 0.5 respectively) on the quality of approximation of these algorithms are actually tight. In this chapter, we cover the generation of almost tight instances for the TS and DESCENT algorithm.

Starting with the DESCENT algorithm, we were able to generate a class of graph transduction games, originally studied in [49] as a method of applying game theoretical methods to classification problems, for which the DESCENT algorithm was not able to easily achieve small approximation guarantees. Using this class of games, we were typically capable of generating instances where the computed approximation was 0.1214. In the worst case scenario, we generated games where DESCENT resulted in a 0.4996 and 0.4835 for δ having the values of 0.1 and 0.001 respectively.

For the TS algorithm, we were able to represent the problem as a search problem. By formulating the problem in this fashion, it allows us to make use of metaheuristic algorithms such as genetic algorithms, or optimization techniques which are used in machine learning for model selection and hyperparameter optimization.

By using these methods to search for worst-case examples, we were able to find a 5×5 bimatrix game in which the TS algorithm gives no better than a 0.3385-Nash equilibrium, which shows that the performance guarantee is essentially tight.

Finally, in order to test the limits of the approximation techniques, we ran the same procedure against the combination of our three best approximation algorithms (the TS algorithm, the algorithm of Bosse et al. [17], and the best approximation provided by a pure strategy pair.) We were able to find a game for which all of those three algorithms gave no better than a 0.3189-Nash equilibrium, which is relatively close to the theoretical upper bound of 0.3393, and perhaps indicates that new techniques will be needed to advance the theoretical state of the art.

4.3 Polymatrix Games: Graph Transduction Games

In this section, we provide a class of games generated from a real world application of game theory for which we are capable of generating an almost tight lower bound for the DESCENT algorithm which computes a $0.5 + \delta$-NE on polymatrix games. We first start by describing the construction of such games and the datasets which we used to generate our instances before finally showing the results generated with these games.

4.3.1 Graph Transduction Games

Graph transduction is a popular class of semi-supervised learning techniques which aims to estimate a classification function defined over a graph of labeled and unlabeled data points. In [49], the transduction problem is formulated as a polymatrix game, such that the pure Nash equilibria of the derived game corresponds to consistent labeling of the data in the original classification problem.

In their paper, Erdem and Pelillo [49] represent a classification problem as a game by first having each data point within the dataset be represented by a player in the corresponding game. Each player has its set of strategies defined as the set of possible classes in the problem. Although not used in the original paper due to the method of implementation, the strategy set for a player representing a labeled point can be reduced to a single element as the classification of this point has already been fixed.

In order to generate meaningful payoff matrices for the polymatrix game, the similarity matrix W of the dataset is first computed, such that the element W_{ij} represents how similar the i-th and j-th data points are. Based on the similarity matrix, the pair of players i, j are involved in a bimatrix game, with the payoff of the players being defined as $W_{ij} \cdot I$, where I is the $c \times c$ identity and c is the number of classes available. The similarity between two data points i and j is computed using the Gaussian kernel

$$W_{ij} = \exp\left(-\frac{dist(i,j)}{2\sigma^2}\right),$$

where $dist(i, j)$ is the distance between data points i and j, and σ is the width of the kernel being applied. Several measures of distance were used depending on the dataset, the Euclidean distance was used for Adult, USPS and MNIST [49, 127], Cosine distances for 20-News [49, 127] and the Value Difference Metric (VDM) [119] was used for the House-Votes-84 dataset [126]. In order to increase the accuracy of their method, they performed the following normalization to derive the final similarity matrix $\widehat{W} = D^{-1/2} W D^{-1/2}$

where D is the degree matrix of W, i.e. $D_{ii} = \sum_j W_{ij}$.

When evaluating their method, the value of the kernel width σ was chosen from the set $linspace(0.1r, r, 5) \cup linspace(r, 10, r, 5)$ with r being the average distance from each data point to its furthest neighbour and $linspace(a, b, n)$ represents the set of n linearly spaced numbers between and including a and b. We examine in a similar manner, for each subset of data, all 9 possible values of σ are used to create 9 varying representations of the specified data subset. Considering how the values of σ are not going to be the same across all instances, even those from the same dataset, we would use the σ to denote the actual value of the kernel width, while σ_i will be used to indicate the index of σ in the set of possible kernel widths as described above. For example σ_1 would refer to the first possible value of σ for a given instance, which corresponds to $0.1r$.

One key portion of their method [49] which we did not implement was the incorperation of nearest neighbours. The main reason for not going down this route is because it leads to really sparse games, which as shown in the Chapter 3 produces representations which are larger than they need to be.

4.3.2 Datasets

For this study, we make use of 5 different datasets which have been used in various studies related to classification algorithms. These datasets serve as the foundation for creating game classes where the DESCENT algorithm struggles to find a good approximation.

- **20-News**: This is the text classification data set which contains 3970 articles which were selected from the 20-newsgroup data set where the subjects partain to `autos`, `motorcycles`, `sport.baseball` and `sport.hockey`. We use the data as processed in [49] and [127], where each article is represented as a 8014-dimensional TF-IDF vector.

- **Adult**: This is a binary classification dataset, i.e. there are only two classes in this problem. The dataset is extracted from the US census bureau, with the task being to predict whether a person earns above $50,000 or not, based on several attributes such as age, gender, and so on. We make use of the same dataset seen in [102], where the dataset has been preprocessed in such a way that continuous attributes have been discretized into quintiles and the attributes transformed to yield a total of 123 binary attributes.

- **House-Votes-84**: This is another binary classification dataset, from the UCI data

archive [87]. The aim of the dataset is to classify congressmen as being either republican or democrat based on their votes on 16 key topics. For all congressmen, each attribute contains either a `y` representing a vote for, `n` representing a vote against, or a `?` representing an unknown disposition. As studied in [126], with this dataset, we made use of the value difference metric in order to compute the pairwise distance between data elements.

- **MNIST**: This is a dataset consisting of 70,000 hand-written digits of size 28×28 of digits 0 - 9. From this set of images, we made use of only 4 classes belonging to digits 1 - 4. These images are represented as a 784-dimensional vector, with the distance between any two images being the euclidean distance between their respective vector representations.

- **USPS**: This dataset contains images of hand-written digits of size 16×16 of digits 0 - 9. In a similar fashion to [49], we only make use of digits between 1 - 4, which gives us a total of 3874 images. As with the MNIST dataset, the distance between any two images in this dataset is determined by the euclidean distance of the vector representations of the image data.

4.3.3 Results

All computations were done using implementation of the DESCENT algorithm, as stated in Chapter 3, with values of δ being both 0.1 and 0.001. For each instance we ran, we started the DESCENT algorithm from a pure strategy profile where each players' pure strategy is chosen uniformly at random. Overall, we had a total of 2,000 classification instances of various sizes selected using the previously mentioned datasets (see Section 4.3.2). The size of the problem is defined by the number of labeled and unlabeled data points. In this study, we look at problems where the number of labeled and unlabeled points are the same per class. For example, a problem of size 3 on the MNIST dataset implies that for each instance there are 3 unlabeled and 3 labeled data points per class, which gives us a game with 24 players, 12 of which have 4 strategies, while the others have a strategy set of size 1. Using these instances of classification problems, we generated a total of 20,000 polymatrix games using the method described above.

Over all the games which were solved by both versions of the DESCENT algorithm, more than 90% of the instances had an approximation guarantee of at most 0.1-NE. As shown in Figure 4.4, however, a huge portion of these games which resulted in low ap-

proximation guarantees were from games other than $\sigma = 0.1r$. In general, as the value of σ increases, we tend to see a decrease in the values of ϵ found.

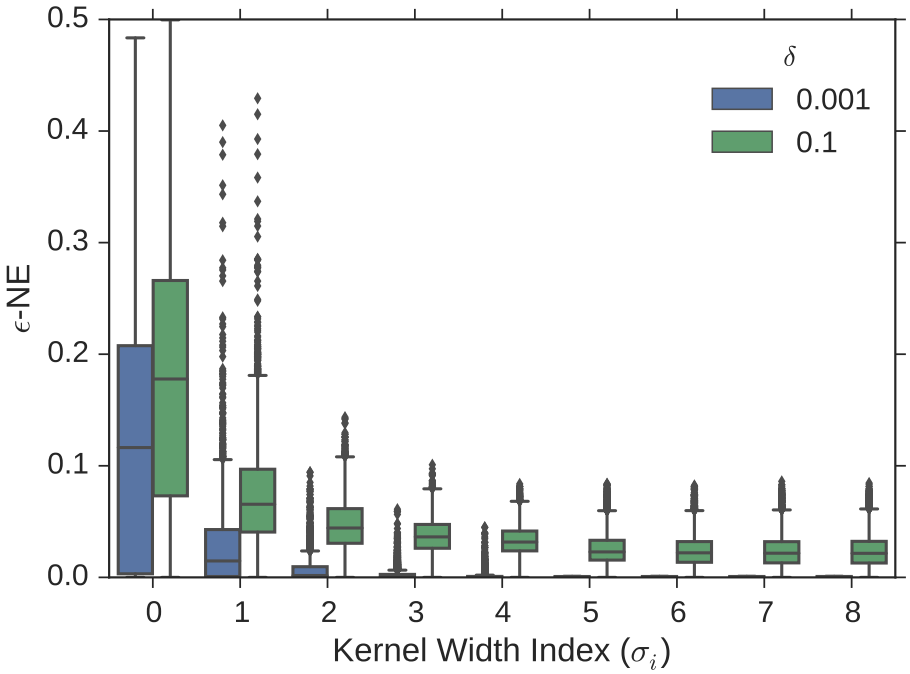
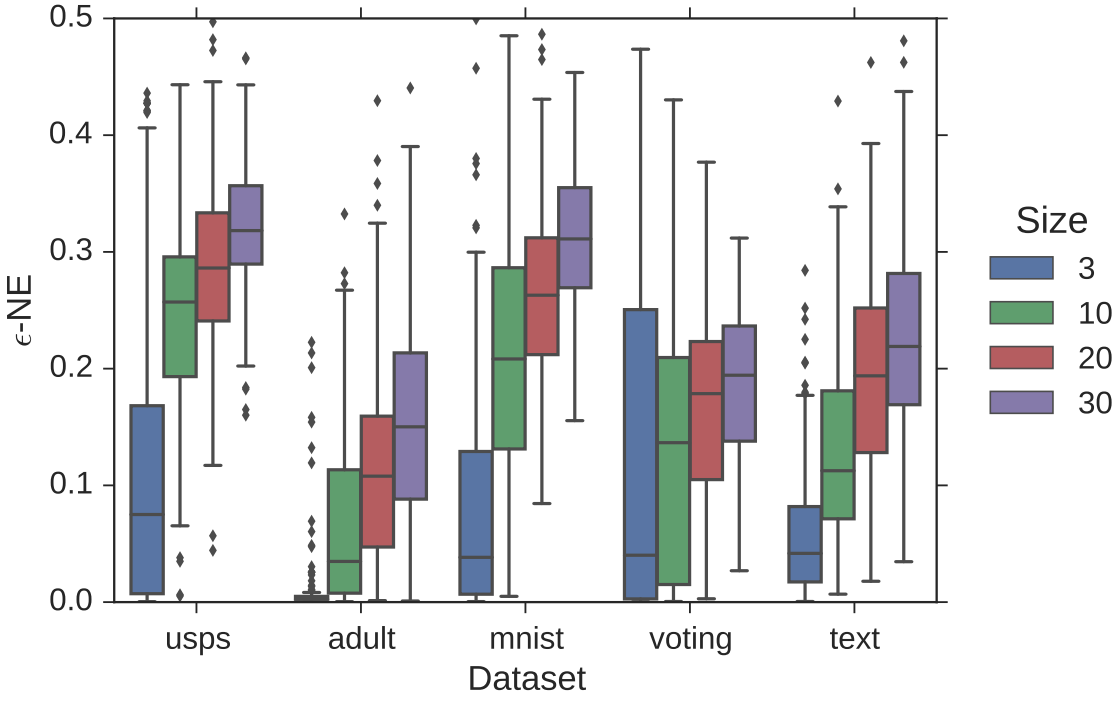


Figure 4.4: Overall performance of the DESCENT algorithm with δ of 0.1 and 0.001 across the different possible values for the kernel width σ

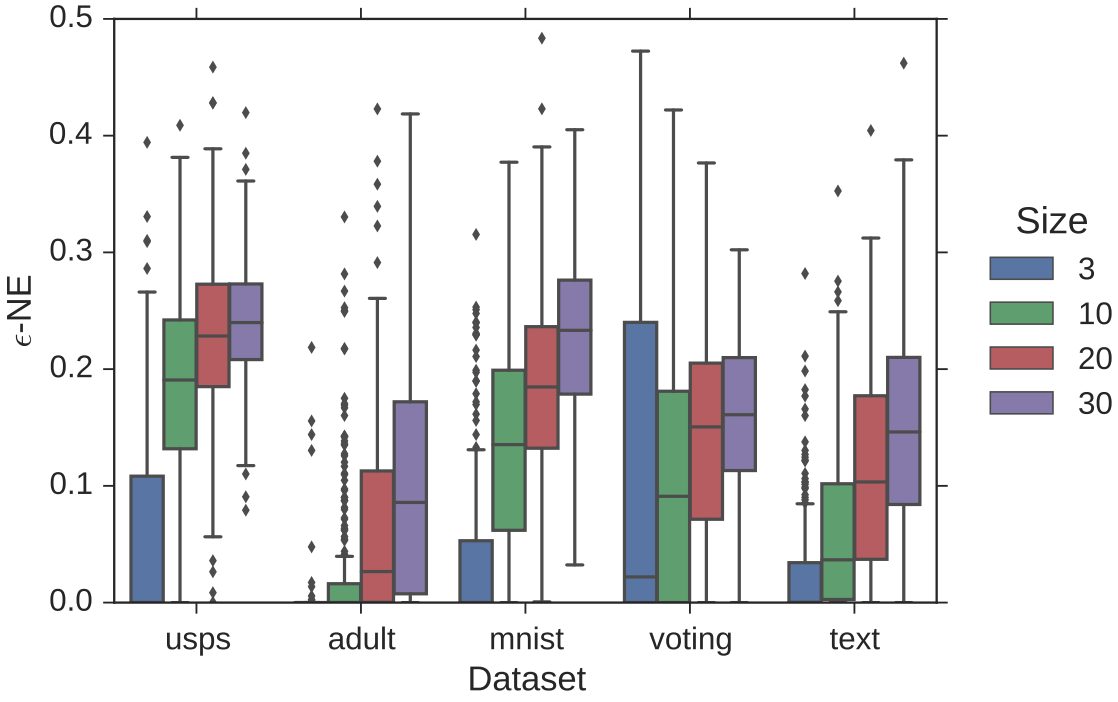
.

Limiting the choices to be the first possible value of the kernel width, i.e. $\sigma = 0.1r$, we are capable of generating games such that the approximation guarantee is considerably worse than any of the previously studied games.

From the results achieved from these classes of experiments, we were able to observe that in comparison to each other, the digit recognition datasets (both USPS and MNIST) despite the difference in sizes of the raw images, tended to provide not only the same ranges of results, but were also the datasets which provided the hardest games. A trend which can also be spotted from the results, is the relationship between the size and the ϵ-NE found. As the size of the classification instance increases, so does the potential ϵ-NE which was found by DESCENT.



(a) DESCENT algorithm with $\delta = 0.1$



(b) DESCENT algorithm with $\delta = 0.001$

Figure 4.5: Performance of the DESCENT algorithm across the different datasets with various sizes, where the kernel width parameter $\sigma = 0.1r$

4.4 Bimatrix Games: Heuristic Search Methods

In this section, we focus on generating lower bounds for the current state-of-the-art algorithm for computing approximate Nash equilibria in bimatrix games, which is guaranteed to find a 0.3393-NE. We present and show the results obtained by performing heuristic search using: a non-parameterized algorithm, Tree-structured Parzan Estimator which performs a form of Bayesian optimisation; and a parameterized algorithm, a Genetic Algorithm which is based on evolutionary mechanisms. Both these methods can be adapted to assist in the generation of lower bounds for future algorithms.

In general, when constructing lower bounds, the algorithm being studied is thoroughly investigated, from the initialization steps of the algorithm, looking at all possible edge cases that might happen along the way till the final output is generated. Algorithms such as TS and DESCENT, however, can be quite difficult to analyse due to their complex nature. In this section, we will show an overview of the method used in generating a near tight lower bound for the TS algorithm.

In most machine learning problems, the learning algorithm which is being used has to be trained on the data at hand and prior to training, has to be configured. Its configuration variables, otherwise known as *hyperparameters*, are key factors which can greatly affect the performance of the algorithm on the provided data. Depending on the problem set at hand, the algorithms and the complexity of the model being used, manual and brute-force search for the best choice of hyperparameters can be impossible to achieve.

In order to address this problem, several methods of searching for the best model and hyperparameters for a given problem have been introduced. In general, they involve the description of two main properties: the *search space* of the hyperparameters and the *loss function* which is used to attribute a choice of hyperparameters with a score, otherwise known as a *fitness value*.

In order to traverse the search space, we briefly look at two methods which are used in practice. The first being a *genetic algorithm*, which is a form of parameterized heuristic search, while the second method, follows a non-parameterized Bayesian optimization approach.

Search Space

In order to describe the search space for the game, we define a single variable to correspond to each entry in the payoff matrices of the game. In other words, searching for an $n \times n$ bimatrix game would be represented by a vector $2n^2$ variables. Each variable is

associated to a probability distribution over a range of values, representing the possible values the variable can have. In situations with no prior knowledge of games for which the algorithms performance is close to its upper bound, the variables can be initialised so as to be uniformly distributed over a range of values.

Loss Function

The loss function, is the objective function which the search method is trying to minimize. As mentioned earlier, the loss function is applied to each choice of parameters, or in our case, each game which is represented by the parameters. Seeing as we are looking for a lower bound, the loss function is described below as a function of the approximation produced by the algorithm

$$f^{\texttt{ALG}}(G) = 1 - \epsilon^{\texttt{ALG}}(G),$$

where G is the game being evaluated for algorithm ALG, and $\epsilon^{\texttt{ALG}}(G)$ is the value of ϵ in the approximate or well-supported Nash equilibrium found by ALG on game G. The above loss function does not place any restriction (besides the value of ϵ) on the games being evaluated. This can lead to games which, although had a high value of ϵ for the algorithm being investigated, could have pure Nash equilibria which could be easily found. This problem can be resolved by modifying the above cost function to take into consideration the best approximation which can be found by searching for the best pure strategy profile

$$f^{\texttt{ALG}}(G) = 1 - \min(\epsilon^{\texttt{ALG}}(G), \epsilon^{\texttt{PURE}}(G)),$$

where $\epsilon^{\texttt{PURE}}(G)$ is the value of the best approximation found by playing pure strategies. Also, another preprocessing step which can be carried out is the check of strictly dominated strategies. Given that the search is for a game G of a fixed size, rather than eliminating dominated strategies to form a smaller game, the game can be completely eliminated by setting the cost to 1.

$$l^{\texttt{ALG}}(G) = \begin{cases} 1 & \exists \text{ a strategy in } G \text{ that is strictly dominated} \\ f^{\texttt{ALG}}(G) & \text{otherwise} \end{cases}.$$

The loss functions can be further extended to find hard games across a given set of algorithms A

$$l^A(G) = \begin{cases} 1 & \exists \text{ a strategy in } G \text{ that is strictly dominated} \\ \min_{\text{ALG} \in A}(f^{\text{ALG}}(G)) & \text{otherwise} \end{cases}.$$

4.4.1 Tree-structured Parzan Estimator

One method which is used in practice to optimize machine learning hyperparameters is the Tree-structured Parzen Estimator (TPE) [14]. This algorithm is a form of sequential model-based optimization (SMBO) algorithm which works by sequentially creating models to estimate the performance of hyperparameters in the search space based on historical results [12]. Using the models which it generates, the algorithm then suggests new hyperparameters to be evaluated which according to its model, gives the maximum expected improvement of the loss function.

Assuming the search space to be completely unknown, each payoff in the matrix is assummed to be drawn from a uniform distribution. The range in which it is distributed is not of much importance, as the values of the chosen payoff values would get normalized before being solved. For the purposes of our implementation, each payoff value was chosen from the range $[0, 1]$.

In order to perform the search using this method, we make use of the *hyperopt* [13,14] library, which is available within Python. One of the main advantages of using this method, is the fact that it is an automated search method which has little to no parameters to be passed to it. The only setup neccessary for this method is the description of the search space and an implementation of the loss function.

4.4.2 Genetic Algorithm

In a similar fashion to the TPE algorithm, a genetic algorithm is a structured form of random search. Genetic algorithms are a subgroup of evolutionary algorithms that stem from the concept of evolution in biological sciences. In these algorithms, the term *individual* is used to represent a single choice of parameters, whereas *population* refers to a set of individuals.

The genetic algorithm which we employed, as shown in Algorithm 4.1, is a $(\mu + \lambda)$ genetic algorithm from within the *deap* Python library [58]. In this algorithm, the next population is formed of μ individuals from the current population and λ individuals created from the μ selected individuals.

The population is initialised to be a selection of random individuals selected from the search space. After the initialization, the genetic algorithm iterates / runs for a specified number of generations. During each generation t, the fitness (which in our setting is also the loss function) of all individuals in the population P_t is evaluated. A subset of size μ is then selected from the population based on the fitness of the individuals and a new population of size $\mu + \lambda$ is generated using the selected individuals by creating λ new individuals from the selected set using either crossover (combination of two or more individuals), or mutation (randomly changing portions of an individual).

Algorithm 4.1: A $(\mu+\lambda)$ genetic algorithm, where μ is the number of individuals to be selected for the next generation and λ is the number of new individuals to create every generation

```
setup the initial population P_0
evaluate all individuals in P_0
t = 0
while TRUE do
    Let O be the set of offsprings
    foreach i ∈ [λ] do
        O_i is created with probability p_c using crossover otherwise use mutation of
        random individuals from P_t
    evaluate all individuals in O
    P_{t+1} = select μ individuals from P_t ∪ O
    t = t + 1
```

Representation & Population

One of the basic building blocks of a genetic algorithm is the individual. Traditionally, an individual in a genetic algorithm is usually represented by a bitstring. This form of representation gives the algorithm quite a lot of power as it is capable of performing its operations on a bitwise level. If we were to represent a 5×5 bimatrix game using 32-bit integers, it would be equivalent to having an individual of length 1600.

Having large individuals eventually becomes a problem when choosing the population size, as it has been argued that having a *small* population size could lead the algorithm towards early convergence at poor solutions [82,90,100], while a population size that is too *large* would use much more computational time than neccessary to find a solution [82,90]. Despite this, studies have been done with regards to making use of populations of small sizes, where techniques such as varying the population size [82] or reinitialization of the population [50, 62] have been used as a counter-measure against early convergence.

For the purpose of our study, we represent each individual as a vector of integers. Doing so allows us to keep a reasonable size in terms of the individual, without losing a lot of the capabilities of representing as a bitstring.

Selection

Upon evaluation of the individuals in the population, the genetic algorithm needs to select the individuals which it is going to use to create the population for the next generation. From the wide variety of selection methods, we decided to make use of the *tournament* selection method. This involves selecting k individuals by performing k tournaments of size s. Each tournament consists of s random individuals from the population, with the winner of the tournament being the individual with the highest fitness score. The winners of the tournaments define the set of selected individuals to be used for reproduction.

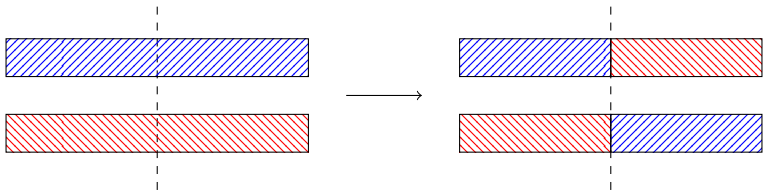
Crossover

Crossover is the process of generating new individuals (children) from parent individuals. Although there are several forms of crossover, the three prominent ones are (see Figure 4.6 for a visual representation of these methods):

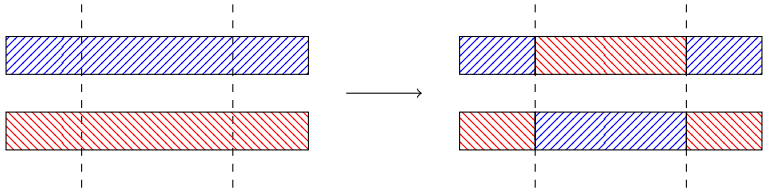
- *Single-point* crossover, where a single point in the two individuals is chosen at random and all data past the crossover point is swapped between the two individuals.
- *Two-point* crossover, where two points in the two individuals are chosen at random and all data within the chosen points is swapped.
- *Uniform* crossover, where each corresponding data point in the two individuals get swapped independently at random with a predefined probability.

Mutation

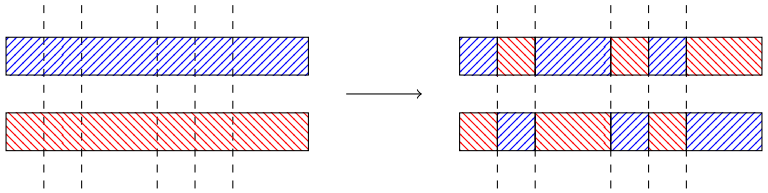
Unlike crossover which requires two parent individuals in order to produce a new individual, mutation only requires one. This is a method which is used to maintain some diversity within the individuals in the population. Although there are various forms of mutation, the one which best suits our requirements is a uniform mutation. In this form of mutation, each parameter within an individual selected for mutation is modified independently at random with a predefined probability. The new values of the parameters to be modified



(a) Example of a one point crossover



(b) Example of a two point crossover



(c) Example of a uniform crossover

Figure 4.6: Examples of different crossover methods

are then chosen uniformly at random from the set of possible values it is allowed to have as dictated by the search space.

4.4.3 Results

Before presenting the instances found that are capable of providing a good enough lower bound, we shall briefly investigate the performance of the two methods for performing the search which we mentioned in Sections 4.4.1 and 4.4.2.

In order to track the progress of a single run of the genetic algorithm, we computed four statistics in order to describe the population at any given generation. The first two, the maximum and average fitness values in the population, help to portray how close to the expected outcome the algorithm is. The last two being the minimum fitness value and standard deviation of the fitness values of the population convey the diversity of the population.

Each plot shown in Figure 4.7 shows for each generation, the average value of the four stated statistics over a total of 25 separate runs of the genetic algorithm.

With regards to the genetic algorithm, despite there being several variables with which we could dive deeply into and fine tune, doing so is beyond the scope of this study. For

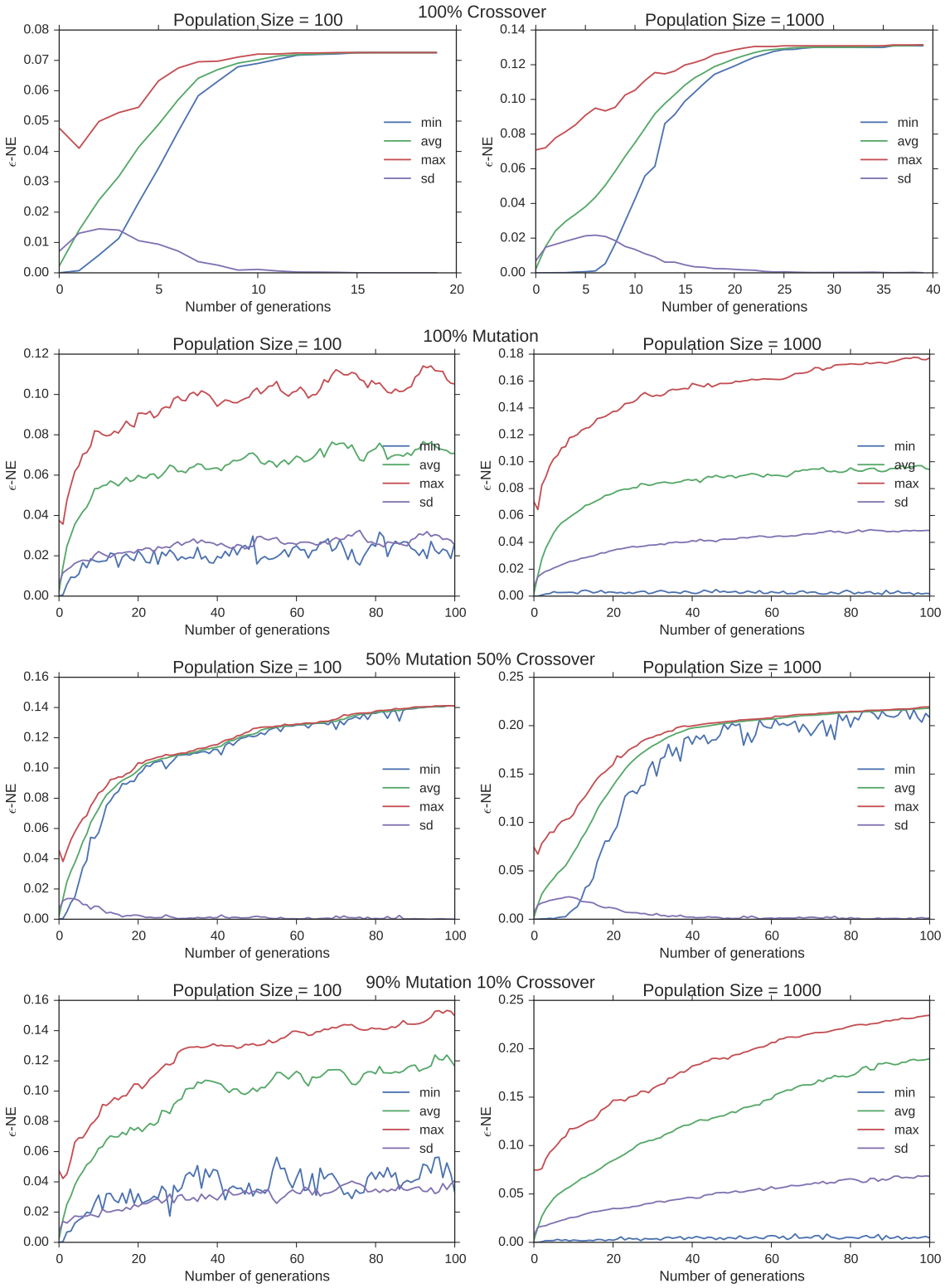


Figure 4.7: Average performance of genetic algorithms over 100 generations for population size of 100 and 1000 (left and right columns respectively) on 25 independent runs of the algorithm

the most part, we picked our parameters based on values which have been used in prior research work. We will briefly investigate two key parameters which influence the performance.

The first of which is the population size. In coherence to results from previous research on the population size, we find that it is harder to maintain diversity on a small population in comparison to a larger one. As a result, smaller populations tend to converge quicker to poor solutions.

The second parameter, the ratio between mutation and crossover probabilities, also has a direct impact in maintaining the diversity of a population over time. As shown in Figure 4.7, when we have 100% crossover, the algorithm quickly converges to a game in approximately 30 generations. On the contrary, having 100% mutation, gives a lot of diversity with the population especially in the setting with a larger population.

Having a mix between mutation and crossover, gives us some of the capabilities of both. The algorithm is able to converge towards specific solutions as a result of the crossover while being able to maintain a diverse population via mutation. The amount of diversity which one needs to achieve can be reached by tweaking the proportion.

To produce the lower bounds, we kept various instances of the genetic algorithm running, each starting from a different randomly chosen initial popution, with the following parameters.

- *Individual*: A vector of 50 integers within the range $[0, 10^5]$
- *Population Size*: 1000
- *Selection Method*: Tournament selection with tournaments of size 10
- *Crossover*: Uniform Crossover
- *Mutation*: Each element of an individual selected for mutation is set to a new value x independently with a probability 0.1, otherwise retains its previous value, where $x \in [0, 10^5]$.

For the TPE algorithm, considering that each iteration of this algorithm is an evaluation of a single game, unlike the genetic algorithm which processes several games within a single iteration, the handling of the data is slightly different. For each run of the algorithm, at any point in time, we take the running maximum of the ϵ found (i.e. the maximum from the start to that point in time) as a measure of the progress made by the algorithm. Using this value, the plot shown in Figure 4.8 shows over 25 different runs of this algorithm for

a total of 10,000 games. For each number of game evaluations, it shows the best, worst, and average progress which is being made by the different instances being run.

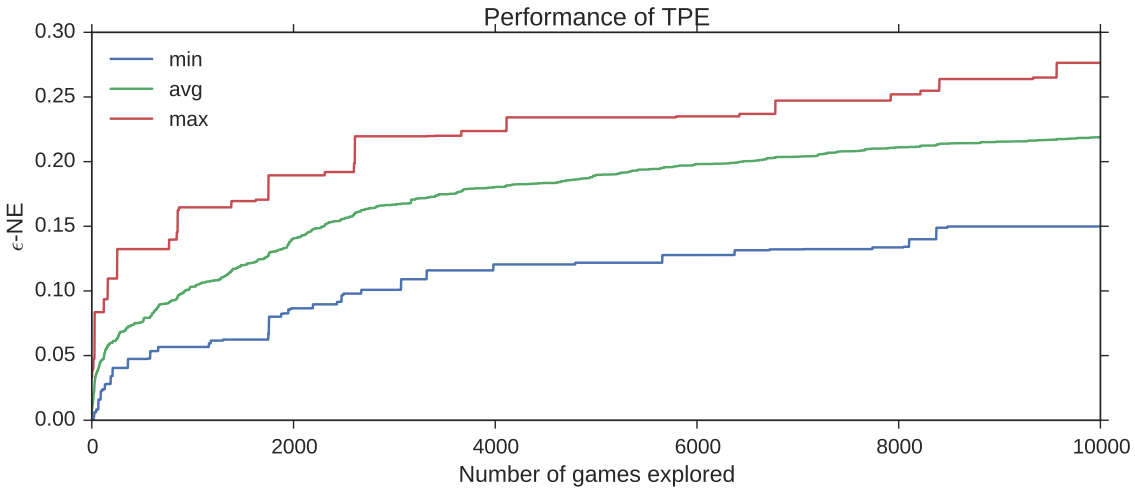


Figure 4.8: Performance of the TPE algorithm over 25 independent runs

4.4.4 A 0.3385-NE Lower Bound for TS001

Consider the game shown in Figure 4.9, in order to verify that this game does indeed provide a lower bound on the TS algorithm with a $\delta = 0.001$, we only need to investigate the final steps of the algorithm. Given a mixed strategy (x, y),

$$x = \begin{bmatrix} 0.0 & 0.0 & 0.176 & 0.823 & 0.001 \end{bmatrix}^T, \quad y = \begin{bmatrix} 0.000292 & 0.0 & 0.916 & 0.084 & 0.0 \end{bmatrix}^T,$$

making use of the normalized version of the game (Figure 4.10), we can verify that this mixed strategy yields a 0.338533-NE for both players. Plugging this information into the linear program to find the direction of the steepest descent produces the point (x', y')

$$x' = \begin{bmatrix} 0.0 & 0.046 & 0.130 & 0.824 & 0.0 \end{bmatrix}^T, \quad y' = \begin{bmatrix} 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \end{bmatrix}^T.$$

Computing the value of the gradient $Df(x, y, x', y')$ gives us a value of -0.000987 which is greater than -0.001, hence is a 0.001 stationary point. Continuing with the rest of the algorithm leads us to find the mixed strategy $(\widetilde{x}, \widetilde{y})$

$$\widetilde{x} = \begin{bmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}^T, \quad \widetilde{y} = \begin{bmatrix} 0.251 & 0.513 & 0.217 & 0.020 & 0.0 \end{bmatrix}^T,$$

resulting in a 0.338514-NE.

$$A = \begin{bmatrix} 56887 & 96209 & 18255 & 39791 & 34582 \\ 85196 & 65524 & 99782 & 97070 & 96765 \\ 72792 & 22633 & 59840 & 99422 & 20117 \\ 32963 & 18315 & 72881 & 85009 & 88426 \\ 26072 & 57677 & 19770 & 67548 & 43422 \end{bmatrix}$$

$$B = \begin{bmatrix} 97209 & 43186 & 16342 & 16048 & 39059 \\ 60251 & 99413 & 3320 & 4596 & 28038 \\ 39941 & 78448 & 52995 & 10725 & 76914 \\ 91891 & 83644 & 53468 & 15871 & 16222 \\ 83130 & 99012 & 95399 & 95610 & 44184 \end{bmatrix}$$

Figure 4.9: A bimatrix game for which TS001 finds a 0.3385-NE.

4.4.5 A 0.3189-NE Lower bound for PURE, BBM2 and TS001

Consider the game shown in Figure 4.11, normalization of both payoff matrices yields the equivalent game shown in Figure 4.12.

PURE

Performing a search for the best pure strategy, we end up with the row and column players playing their first and second pure strategies respectively, which corresponds to the first row and second column of the bimatrix game. In this outcome, the row player gains a payoff of approximately 0.6283, while the column player gains a payoff of 1.0. The column player is currently playing its best response and does not have any regret. On the other hand, the row player's best response is to switch to the second row, where a payoff of 0.9523 is possible. This gives the row player a regret of 0.324.

BBM

For both the BBM1 and BBM2 algorithms, the result is exactly the same, as there is no difference between the steps the different algorithms go through for this game instance. The algorithms initially solve the zero-sum game $(A-B, B-A)$, this results in the mixed

$$A = \begin{bmatrix} 0.473855 & 0.956174 & 0.000000 & 0.264158 & 0.200265 \\ 0.821090 & 0.579796 & 1.000000 & 0.966735 & 0.962994 \\ 0.668944 & 0.053700 & 0.510076 & 0.995584 & 0.022839 \\ 0.180406 & 0.000736 & 0.670036 & 0.818796 & 0.860709 \\ 0.095882 & 0.483545 & 0.018583 & 0.604622 & 0.308695 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.977064 & 0.414869 & 0.135515 & 0.132455 & 0.371921 \\ 0.592457 & 1.000000 & 0.000000 & 0.013279 & 0.257230 \\ 0.381100 & 0.781826 & 0.516947 & 0.077061 & 0.765862 \\ 0.921722 & 0.835899 & 0.521869 & 0.130613 & 0.134266 \\ 0.830550 & 0.995827 & 0.958228 & 0.960424 & 0.425255 \end{bmatrix}$$

Figure 4.10: A normalized version of the game shown in Figure 4.9

$$A = \begin{bmatrix} 60599 & 68002 & 94170 & 91622 & 92353 \\ 25814 & 90811 & 37672 & 74870 & 45424 \\ 45917 & 33775 & 23776 & 45343 & 71345 \\ 25339 & 61618 & 37098 & 24770 & 24292 \\ 39656 & 24016 & 63463 & 63396 & 75634 \end{bmatrix}$$

$$B = \begin{bmatrix} 59772 & 98580 & 12901 & 34113 & 34916 \\ 98451 & 59776 & 20776 & 13893 & 27315 \\ 17890 & 81316 & 36363 & 20947 & 36334 \\ 53317 & 45329 & 14652 & 42939 & 17098 \\ 16562 & 67590 & 62024 & 16783 & 13882 \end{bmatrix}$$

Figure 4.11: A bimatrix game for which TS001 finds a 0.3189-NE, PURE finds a 0.324-NE, and BBM2 finds a 0.321-NE.

strategies (x, y)

$$x = \begin{bmatrix} 0.0 & 0.0 & 0.400 & 0.600 & 0.0 \end{bmatrix}^T, \quad y = \begin{bmatrix} 0.53606 & 0.46933 & 0.0 & 0.0 & 0.0 \end{bmatrix}^T.$$

The expected payoff gotten with the mixed strategies (x, y) are $x^T Ay = 0.25263$ for the row player and $x^T By = 0.42069$ for the column player. Based on this information, we can easily work out that the regret for the column player is 0.1258, while the row player has a regret of 0.32265, giving a 0.32265-NE. Due to the approximation guarantee being less than $\frac{1}{3}$, the algorithms terminate and report the current solution.

$$A = \begin{bmatrix} 0.523099 & 0.628264 & 1.000000 & 0.963804 & 0.974188 \\ 0.028951 & 0.952283 & 0.197403 & 0.725829 & 0.307526 \\ 0.314530 & 0.142043 & 0.000000 & 0.306376 & 0.675754 \\ 0.022204 & 0.537574 & 0.189249 & 0.014121 & 0.007330 \\ 0.225587 & 0.003409 & 0.563784 & 0.562832 & 0.736682 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.547054 & 1.000000 & 0.000000 & 0.247575 & 0.256947 \\ 0.998494 & 0.547100 & 0.091913 & 0.011578 & 0.168233 \\ 0.058229 & 0.798504 & 0.273836 & 0.093909 & 0.273498 \\ 0.471714 & 0.378482 & 0.020437 & 0.350588 & 0.048985 \\ 0.042729 & 0.638301 & 0.573338 & 0.045309 & 0.011450 \end{bmatrix}$$

Figure 4.12: A normalized version of the game shown in Figure 4.11

TS001

Out of the three algorithms, the TS001 algorithm provides us with the lowest approximation of 0.3189. Given the mixed strategy,

$$x = \begin{bmatrix} 0.247 & 0.0 & 0.0 & 0.0 & 0.753 \end{bmatrix}^T, \quad y = \begin{bmatrix} 0.0 & 0.0 & 0.936 & 0.0 & 0.064 \end{bmatrix}^T,$$

and the direction of the steepest descent

$$x' = \begin{bmatrix} 0.272 & 0.0 & 0.0 & 0.0 & 0.728 \end{bmatrix}^T, \quad y' = \begin{bmatrix} 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \end{bmatrix}^T,$$

following the definition of a stationary point, these vectors correspond to a 0.001-stationary point. Taking into consideration the set of extra points which the TS algorithm finds, the mixed strategy profile which minimizes the maximum regret of both players is (x, y), resulting in a 0.3189-Nash equilibrium.

Chapter 5

Conclusion

In this thesis, we studied algorithms for computing approximate equilibria in several classes of games from an empirical perspective.

Starting off with Chapter 2, where we studied the performance of several algorithms for computing both exact and approximate Nash equilibria in bimatrix games. We found that the existing library of games provided by GAMUT is biased towards games that always have pure Nash equilibria which were fairly easy for the algorithms being studied to find, so we introduced several classes of games where this is not the case. Having done so, we found that the exact algorithms LH and SE are quite limited in their ability to solve large games, particularly on the games that we introduced. In contrast to this, we have seen that approximation methods can tackle much larger instances, and that they provide approximate equilibria that are good enough to be practically useful.

In Chapter 3, we extended the work performed on bimatrix games and showed results of a study on polymatrix and two-player Bayesian games. We started by tackling one of the issues of the GAMUT library, a concise representation for polymatrix and two player Bayesian games. Based on this representation, we implemented several classes of games which we used to evaluate the Lemke's algorithm (for computing exact equilibria), and the DESCENT algorithm (for computing approximate equilibria). Both algorithms produce good results for most of the test instances, even though many were drawn from theoretically hard classes. In all of our experiments DESCENT produced ϵ-NE far better than the (best-known) 0.5 theoretical worst-case guarantee, even in cases where it timed-out before reaching a stationary point. In general, across all the games which we studied, we found the DESCENT algorithm performed worse on cycles when compared to different graph structures on the same game class.

In Chapter 4, due to the performance of both the TS and DESCENT algorithms, we

turned our attention towards the search for lower bounds. We were capable of generating a near-tight lower bound for the DESCENT algorithm which computes a 0.5-NE on polymatrix games. This lower bound was constructed from a real world application of game theory known as graph transduction games, which formulates the problem of data classification as a game. Using these games, we were able to generate lower bounds of 0.4996-NE and 0.4835-NE for the DESCENT algorithm with δ of 0.1 and 0.001 respectively. For the bimatrix games, we explored the use of two heuristic search methods in order to find lower bounds for the TS algorithm. These methods proved to be robust, yielding a game where the algorithm finds a 0.3385-NE when run with a δ of 0.001.

5.1 Future Directions

Although we do provide a library of games for both bimatrix and polymatrix games, by no means does it cover the plethora of interesting classes of games in the literature. It is also likely for there to be other game classes which could be hard instances for the algorithms which we studied. In its current state, the library of polymatrix game generators (introduced in Section 3.6) does not make full use of the range of game generators provided by bimatrix game generators (introduced in Section 2.6.1). Improvements to the GAMUT library should also be made in order to allow for the creation of bimatrix games while maintaining its compact representation.

For several game classes, a PTAS [30, 35, 37] or a QPTAS [11, 88] is known to exist. These have been studied extensively in the theoretical body of work and it would be beneficial to have an equally extensive study on the practical performance of these algorithms. Comparisons could be made between these algorithms and the TS and DESCENT algorithms which are the best performing algorithms for a constant value of ϵ. This comparison can be carried out in two ways. The first is by using the upper bounds of TS and DESCENT as the value of ϵ passed to the approximation schemes. The second method involves using the value of ϵ which TS and DESCENT compute as the input for the approximation schemes.

In our investigation of combinatorial auctions with item-bidding in Chapter 3, we assumed a discretized bid space with a discretization of 1. Further investigation into the relationship between the discretization factor and the quality of approximation which gets computed in relation to the original game. In other words, how much trade-off is there between the accuracy of solution and the gain in performance due to a reduction in the game size.

Although the lower bounds shown in Chapter 4 only hold for instances of the TS and DESCENT algorithms that are run with a $\delta > 0.001$, the methods presented provide a starting point to look at the key points which are holding back these algorithms. Although only the instances for which these algorithms performed the worst was presented, there were several instances along the way which could lend some insight into forming a class of games for which these algorithms struggle on.

Bibliography

[1] T. Adamo and A. Matros. A Blotto game with incomplete information. *Economics Letters*, 105(1):100 – 102, 2009.

[2] A. Ahmadi, M. Hajiaghayi, B. Lucier, H. Mahini, and S. Seddighin. From duels to battlefields: Computing equilibria of Blotto and other games. In *Proc. of AAAI*, pages 376–382, 2016.

[3] Y. Anbalagan, S. Norin, R. Savani, and A. Vetta. Polylogarithmic supports are required for approximate well-supported Nash equilibria below 2/3. In *Proc. of WINE*, pages 15–23, 2013.

[4] K. R. Apt, M. Rahn, G. Schäfer, and S. Simon. Coordination games on graphs. In *Proc. of WINE*, pages 441–446. 2014.

[5] A. Arad and A. Rubinstein. Multi-dimensional reasoning in games: Framework, equilibrium and applications. *Working Paper*, 2017.

[6] C. Audet, S. Belhaiza, and P. Hansen. Enumeration of all the extreme equilibria in game theory: Bimatrix and polymatrix games. *Journal of Optimization Theory and Applications*, 129(3):349–372, 2006.

[7] D. Avis, G. D. Rosenberg, R. Savani, and B. von Stengel. Enumeration of Nash equilibria for two-player games. *Economic Theory*, 42(1):9–37, 2010.

[8] M. Babaioff, B. Lucier, N. Nisan, and R. P. Leme. On the efficiency of the walrasian mechanism. In *Proc. of EC*, pages 783–800, 2014.

[9] Y. Babichenko, C. Papadimitriou, and A. Rubinstein. Can almost everybody be almost happy? In *Proc. of ITCS*, pages 1–9, 2016.

[10] S. Barman. Approximating Nash equilibria and dense bipartite subgraphs via an approximate version of caratheodory's theorem. In *Proc. of STOC*, pages 361–369, 2015.

[11] S. Barman, K. Ligett, and G. Piliouras. Approximating Nash equilibria in tree polymatrix games. In *Proc. of SAGT*, pages 285–296, 2015.

[12] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Proc. of NIPS*, pages 2546–2554, 2011.

[13] J. S. Bergstra, D. Yamins, and D. Cox. Making a science of model search: Hyper-parameter optimization in hundreds of dimensions for vision architectures. In *Proc. of ICML*, pages 115–123, 2013.

[14] J. S. Bergstra, D. Yamins, and D. D. Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proc. of Python in Science Conference*, pages 13–20, 2013.

[15] N. A. R. Bhat and K. Leyton-Brown. Computing Nash equilibria of action-graph games. In *Proc. of UAI*, pages 35–42, 2004.

[16] K. Bhawalkar and T. Roughgarden. Welfare guarantees for combinatorial auctions with item bidding. In *Proc. of SODA*, pages 700–709, 2011.

[17] H. Bosse, J. Byrka, and E. Markakis. New algorithms for approximate Nash equilibria in bimatrix games. *Theoretical Computer Science*, 411(1):164–173, 2010.

[18] S. J. Brams. Fair division*. In *Computational Complexity*, pages 1073–1080. Springer, 2012.

[19] F. Brandt, F. A. Fischer, and M. Holzer. Symmetries and the complexity of pure nash equilibrium. *Journal of Computer and System Sciences*, 75(3):163–177, 2009.

[20] Y. Cai and C. Daskalakis. On minmax theorems for multiplayer games. In *Proc. of SODA*, pages 217–234, 2011.

[21] Y. Cai and C. H. Papadimitriou. Simultaneous bayesian auctions and computational complexity. In *Proc. of EC*, pages 895–910, 2014.

[22] X. Chen, X. Deng, and S.-H. Teng. Settling the complexity of computing two-player Nash equilibria. *Journal of the ACM*, 56(3):14:1–14:57, 2009.

[23] X. Chen, D. Paparas, and M. Yannakakis. The complexity of non-monotone markets. In *Proc. of STOC*, pages 181–190, 2013.

[24] G. Christodoulou, A. Kovács, and M. Schapira. Bayesian combinatorial auctions. *Journal of the ACM*, 63(2):11:1–11:19, 2016.

[25] E. H. Clarke. Multipart pricing of public goods. *Public choice*, 11(1):17–33, 1971.

[26] B. Codenotti, S. D. Rossi, and M. Pagan. An experimental analysis of Lemke-Howson algorithm. *CoRR*, abs/0811.3247, 2008.

[27] V. Conitzer and T. Sandholm. New complexity results about nash equilibria. *Games and Economic Behavior*, 63(2):621–641, 2008.

[28] A. Czumaj, A. Deligkas, M. Fasoulakis, J. Fearnley, M. Jurdziński, and R. Savani. Distributed methods for computing approximate equilibria. In *Proc. of WINE*, pages 15–28, 2016.

[29] A. Czumaj, M. Fasoulakis, and M. Jurdziński. Zero-sum game techniques for approximate Nash equilibria. In *Proc. of AAMAS*, pages 1514–1516, 2017.

[30] C. Daskalakis. An efficient PTAS for two-strategy anonymous games. In *Proc. of WINE*, pages 186–197, 2008.

[31] C. Daskalakis. On the complexity of approximating a nash equilibrium. *ACM Trans. Algorithms*, 9(3):23:1–23:35, 2013.

[32] C. Daskalakis, P. W. Goldberg, and C. H. Papadimitriou. The complexity of computing a Nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009.

[33] C. Daskalakis, A. Mehta, and C. H. Papadimitriou. Progress in approximate Nash equilibria. In *Proc. of EC*, pages 355–358, 2007.

[34] C. Daskalakis, A. Mehta, and C. H. Papadimitriou. A note on approximate Nash equilibria. *Theoretical Computer Science*, 410(17):1581–1588, 2009.

[35] C. Daskalakis and C. H. Papadimitriou. Computing equilibria in anonymous games. In *Proc. of FOCS*, pages 83–93, 2007.

[36] C. Daskalakis and C. H. Papadimitriou. On oblivious PTAS's for Nash equilibrium. In *Proc. of STOC*, pages 75–84, 2009.

[37] C. Daskalakis and C. H. Papadimitriou. On oblivious ptas's for nash equilibrium. In *Proc. of STOC*, pages 75–84, 2009.

[38] C. Daskalakis and C. H. Papadimitriou. Continuous local search. In *Proc. of SODA*, pages 790–804, 2011.

[39] C. Daskalakis and C. H. Papadimitriou. Approximate Nash equilibria in anonymous games. *Economic Theory*, 156:207–245, 2015.

[40] C. Daskalakis, G. Schoenebeck, G. Valiant, and P. Valiant. On the complexity of Nash equilibria of action-graph games. In *Proc. of SODA*, pages 710–719, 2009.

[41] C. Daskalakis and V. Syrgkanis. Learning in auctions: Regret is hard, envy is easy. In *Proc. of FOCS*, pages 219–228, 2016.

[42] A. Deligkas, J. Fearnley, T. P. Igwe, and R. Savani. An empirical study on computing equilibria in polymatrix games. In *Proc. of AAMAS*, pages 186–195, 2016.

[43] A. Deligkas, J. Fearnley, R. Savani, and P. G. Spirakis. Computing approximate Nash equilibria in polymatrix games. *Algorithmica*, 77:487–514, 2017.

[44] S. Dobzinski, H. Fu, and R. D. Kleinberg. On the complexity of computing an equilibrium in combinatorial auctions. In *Proc. of SODA*, pages 110–122, 2015.

[45] S. N. Durlauf and L. E. Blume. Game theory and biology. In *Game Theory*, pages 119–126. Springer, 2010.

[46] P. Dütting, F. A. Fischer, and D. C. Parkes. Truthful outcomes from non-truthful position auctions. In *Proc. of EC*, page 813, 2016.

[47] P. Dütting, M. Henzinger, and M. Starnberger. Valuation compressions in vcg-based combinatorial auctions. In *Proc. of WINE*, pages 146–159, 2013.

[48] P. Dütting and T. Kesselheim. Best-response dynamics in combinatorial auctions with item bidding. In *Proc. of SODA*, pages 521–533, 2017.

[49] A. Erdem and M. Pelillo. Graph transduction as a noncooperative game. *Neural Computation*, 24(3):700–723, 2012.

[50] L. J. Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging. In *Proc. of FOGA*, volume 1, pages 265–283, 2014.

[51] K. Etessami and M. Yannakakis. On the complexity of Nash equilibria and other fixed points. *SIAM J. Comput.*, 39(6):2531–2597, 2010.

[52] A. Fabrikant, C. H. Papadimitriou, and K. Talwar. The complexity of pure nash equilibria. In *Proc. of STOC*, pages 604–612, 2004.

[53] J. Fearnley, P. W. Goldberg, R. Savani, and T. B. Sørensen. Approximate well-supported Nash equilibria below two-thirds. *Algorithmica*, 76(2):297–319, 2016.

[54] J. Fearnley, T. P. Igwe, and R. Savani. An empirical study of finding approximate equilibria in bimatrix games. In *Proc. of SEA*, pages 339–351, 2015.

[55] U. Feige and I. Talgam-Cohen. A direct reduction from k-player to 2-player approximate Nash equilibrium. In *Proc. of SAGT*, pages 138–149, 2010.

[56] M. Feldman, H. Fu, N. Gravin, and B. Lucier. Simultaneous auctions are (almost) efficient. In *Proc. of STOC*, pages 201–210, 2013.

[57] M. Feldman, L. Lewin-Eytan, and J. Naor. Hedonic clustering games. *ACM Transactions on Parallel Computing*, 2(1):4:1–4:48, 2015.

[58] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné. DEAP: Evolutionary algorithms made easy. *Machine Learning Research*, 13:2171–2175, 2012.

[59] D. Fotakis, S. C. Kontogiannis, E. Koutsoupias, M. Mavronicolas, and P. G. Spirakis. The structure and complexity of nash equilibria for a selfish routing game. *Theoretical Computer Science*, 410(36):3305–3326, 2009.

[60] V. Fragnelli and S. Moretti. A game theoretical approach to the classification problem in gene expression data analysis. *Computers & Mathematics with Applications*, 55(5):950–959, 2008.

[61] N. Gatti, G. Patrini, M. Rocco, and T. Sandholm. Combining local search techniques and path following for bimatrix games. In *Proc. of UAI*, pages 286–295, 2012.

[62] D. Goldberg. Sizing populations for serial and parallel genetic algorithms. In *Proc. of ICGA*, pages 70–79, 1989.

[63] L. A. Goldberg, P. W. Goldberg, P. Krysta, and C. Ventre. Ranking games that have competitiveness-based strategies. *Theoretical Computer Science*, 476:24–37, 2013.

[64] P. W. Goldberg, C. H. Papadimitriou, and R. Savani. The complexity of the homotopy method, equilibrium selection, and Lemke-Howson solutions. *Transactions of Economics and Computation*, 1(2):9:1–9:25, 2013.

[65] G. Gottlob, G. Greco, and F. Scarcello. Pure nash equilibria: Hard and easy games. *Journal of Artificial Intelligence Research*, 24:357–406, 2005.

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

[67] S. Govindan and R. Wilson. Computing Nash equilibria by iterated polymatrix approximation. *Journal of Economic Dynamics and Control*, 28(7):1229–1241, Apr. 2004.

[68] S. Govindan and R. Wilson. A decomposition algorithm for n-player games. *Economic Theory*, 42(1):97–117, 2010.

[69] T. Groves. Incentives in teams. *Econometrica*, pages 617–631, 1973.

[70] A. Hassidim, H. Kaplan, Y. Mansour, and N. Nisan. Non-price equilibria in markets of discrete goods. In *Proc. of EC*, pages 295–296, 2011.

[71] P. J. Herings and A. van den Elzen. Computation of the Nash equilibrium selected by the tracing procedure in n-person games. *Games and Economic Behavior*, 38(1):89–117, 2002.

[72] J. T. Howson, Jr and R. W. Rosenthal. Bayesian equilibria of finite two-person games with incomplete information. *Management Science*, 21(3):pp. 313–315, 1974.

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

[74] R. Jin, X. He, and H. Dai. Collaborative IDS configuration: A two-layer game-theoretical approach. In *Proc. of GLOBECOM*, pages 1–7, 2016.

[75] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. How easy is local search? *Journal Computer and System Sciences*, 37(1):79–100, 1988.

[76] R. Kannan and T. Theobald. Games of fixed rank: A hierarchy of bimatrix games. In *Proc. of SODA*, pages 1124–1132, 2007.

[77] M. King, J. Atkins, and M. Schwarz. Internet advertising and the generalized second-price auction: Selling billions of dollars worth of keywords. *The American economic review*, 97(1):242–259, 2007.

[78] S. C. Kontogiannis, P. N. Panagopoulou, and P. G. Spirakis. Polynomial algorithms for approximating Nash equilibria of bimatrix games. *Theoretical Computer Science*, 410(17):1599–1606, 2009.

[79] S. C. Kontogiannis and P. G. Spirakis. Exploiting concavity in bimatrix games: New polynomially tractable subclasses. In *Proc. of APPROX*, pages 312–325. Springer, 2010.

[80] S. C. Kontogiannis and P. G. Spirakis. Well supported approximate equilibria in bimatrix games. *Algorithmica*, 57(4):653–667, 2010.

[81] S. C. Kontogiannis and P. G. Spirakis. Approximability of symmetric bimatrix games and related experiments. In *Proc. of SEA*, pages 1–20, 2011.

[82] V. K. Koumousis and C. P. Katsaras. A saw-tooth genetic algorithm combining the effects of variable population size and reinitialization to enhance performance. *Transactions on Evolutionary Computation*, 10(1):19–28, Feb 2006.

[83] D. Kovenock and B. Roberson. A Blotto game with multi-dimensional incomplete information. *Economics Letters*, 113(3):273 – 275, 2011.

[84] V. Krishna. *Auction theory*. Academic press, 2009.

[85] C. E. Lemke. Bimatrix equilibrium points and mathematical programming. *Management Science*, 11(7):681–689, 1965.

[86] C. E. Lemke and J. T. Howson, Jr. Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, 12(2):413–423, 1964.

[87] M. Lichman. UCI machine learning repository, 2013.

[88] R. J. Lipton, E. Markakis, and A. Mehta. Playing large games using simple strategies. In *Proc. of EC*, pages 36–41, 2003.

[89] J. A. List. Friend or foe? a natural experiment of the prisoner's dilemma. *The Review of Economics and Statistics*, 88(3):463–471, 2006.

[90] F. G. Lobo and D. E. Goldberg. The parameter-less genetic algorithm in practice. *Information Sciences*, 167(14):217 – 232, 2004.

[91] R. D. McKelvey, A. M. McLennan, and T. L. Turocy. Gambit: Software tools for game theory, version 16.0.0. `http://www.gambit-project.org`, 2014.

[92] D. A. Miller and S. W. Zucker. Copositive-plus Lemke algorithm solves polymatrix games. *Operations Research Letters*, 10(5):285–290, 1991.

[93] D. Monderer and L. S. Shapley. Potential games. *Games and economic behavior*, 14(1):124–143, 1996.

[94] M. Montero, A. Possajennikov, M. Sefton, and T. L. Turocy. Majoritarian blotto contests with asymmetric battlefields: an experiment on apex games. *Economic Theory*, 61(1):55–89, 2016.

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

[96] N. Nisan and A. Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35(1-2):166–196, 2001.

[97] N. Nisan and A. Ronen. Computationally feasible vcg mechanisms. *Journal of Artificial Intelligence Research*, 29:19–47, 2007.

[98] E. Nudelman, J. Wortman, Y. Shoham, and K. Leyton-Brown. Run the GAMUT: A comprehensive approach to evaluating game-theoretic algorithms. In *Proc. of AAMAS*, pages 880–887, 2004.

[99] C. H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences*, 48(3):498–532, 1994.

[100] M. Pelikan, D. E. Goldberg, and E. Cantu-Paz. Bayesian optimization algorithm, population sizing, and time to convergence. In *Proc. of GECCO*, pages 275–282, 2000.

[101] J. Pita, M. Jain, J. Marecki, F. Ordóñez, C. Portway, M. Tambe, C. Western, P. Paruchuri, and S. Kraus. Deployed ARMOR protection: the application of a game theoretic model for security at the los angeles international airport. In *Proc. of AAMAS*, pages 125–132, 2008.

[102] J. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning*. MIT Press, January 1998.

[103] G. Polevoy, S. Trajanovski, and M. de Weerdt. Nash equilibria in shared effort games. In *Proc. of AAMAS*, pages 861–868, 2014.

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

[105] J. Rosenmüller. On a generalization of the Lemke-Howson algorithm to noncooperative n-person games. *SIAM Journal on Applied Mathematics*, 21(1):73–79, 1971.

[106] R. W. Rosenthal. A class of games possessing pure-strategy nash equilibria. *International Journal of Game Theory*, 2(1):65–67, 1973.

[107] T. Roughgarden. The price of anarchy in games of incomplete information. *Transactions of Economics and Computation*, 3(1):6:1–6:20, 2015.

[108] T. Roughgarden, V. Syrgkanis, and É. Tardos. The price of anarchy in auctions. *J. Artif. Intell. Res.*, 59:59–101, 2017.

[109] G. Rowe, D. U. K. D. of Mathematics, and C. Science;. Game theory in biology. *Physical Theory in Biology: Foundations and Explorations*, 4:443, 1997.

[110] A. Rubinstein. *Game theory in economics*. Edward Elgar Publishing, 1990.

[111] A. Rubinstein. Inapproximability of Nash equilibrium. In *Proc. of STOC*, pages 409–418, 2015.

[112] A. Rubinstein. Settling the complexity of computing approximate two-player Nash equilibria. *SIGecom Exchanges*, 15(2):45–49, 2017.

[113] C. T. Ryan, A. X. Jiang, and K. Leyton-Brown. Computing pure strategy nash equilibria in compact symmetric games. In *Proc. of EC*, pages 63–72, 2010.

[114] T. Sandholm, A. Gilpin, and V. Conitzer. Mixed-integer programming methods for finding Nash equilibria. In *Proc. of AAAI*, pages 495–501, 2005.

[115] R. Savani. *Finding Nash equilibria of bimatrix games*. PhD thesis, London School of Economics, 2006.

[116] R. Savani and B. von Stengel. Hard-to-solve bimatrix games. *Econometrica*, 74(2):397–429, 2006.

[117] R. Savani and B. von Stengel. Unit vector games. *Economic Theory*, 12:7–27, 2016.

[118] Y. Shoham and K. Leyton-Brown. Protocols for multiagent resource allocation: Auctions. In *Multiagent Systems*, pages 315–366. Cambridge University Press, 2008.

[119] C. Stanfill and D. L. Waltz. Toward memory-based reasoning. *Communications of the ACM*, 29(12):1213–1228, 1986.

[120] R. Tripodi and M. Pelillo. Document clustering games. In *Proc. of ICPRAM*, pages 109–118, 2016.

[121] H. Tsaknakis and P. G. Spirakis. An optimization approach for approximate Nash equilibria. *Internet Mathematics*, 5(4):365–382, 2008.

[122] H. Tsaknakis, P. G. Spirakis, and D. Kanoulas. Performance evaluation of a descent algorithm for bi-matrix games. In *Proc. of WINE*, pages 222–230, 2008.

[123] G. van der Laan, A. J. J. Talman, and L. van der Heyden. Simplicial variable dimension algorithms for solving the nonlinear complementarity problem on a product of unit simplices using a general labelling. *Mathematics of Operations Research*, 12(3):377–397, 1987.

[124] H. R. Varian. Position auctions. *international Journal of industrial Organization*, 25(6):1163–1178, 2007.

[125] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of finance*, 16(1):8–37, 1961.

[126] D. R. Wilson and T. R. Martinez. Improved heterogeneous distance functions. *Artificial Intelligence Research*, 6:1–34, 1997.

[127] D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf. Learning with local and global consistency. In *Proc. of NIPS*, volume 16, pages 321–328, 2003.

[128] Q. Zhu, W. Saad, Z. Han, H. V. Poor, and T. Basar. Eavesdropping and jamming in next-generation wireless networks: A game-theoretic approach. In *Proc. of MILCOM*, pages 119–124, 2011.

Appendix A

Bimatrix Game Library

Bimatrix games is library of games useful for testing game theoretic algorithms. The library was built as an extension to the currently existing GAMUT library of games due to its lack of games which were hard to solve for approximate algorithms. With that in mind, we built this library as a test suite for approximation algorithms. There are currently two projects hosted:

Algorithms `http://bimatrix-games.github.io/eps-nash-algos`: This contains implementations of varios algorithms for the approximation of Nash equilibrium. The algorithms implemented consist of DMP, BBM1, BBM2, TS and KS.

Game Library `http://bimatrix-games.github.io/bimatrix-generators`: This contains the implementaions of several classes of games which were introduced to study their properties with regards to approximation.

A.1 File Format

Both the game generators and the algorithm implementations make use of the Gambit file format which

A.2 Algorithms

Global parameters

- `-i filename`: Specify the game which is to be solved

- `-r seed`: Specify a random seed to randomly select starting point of certain algorithms

Pure

```
./pure/pure -i game.nfg
```

DMP

```
./dmp/dmp -i game.nfg -[k:]
```

- `-k k`: This argument is used to set the first pure strategy which is used as the starting point of the DMP algorithm

BBM

```
./bbm/bbm -i game.nfg -[t]
```

- `-t`: By default, the implementation runs the BBM1 algorithm. However, running it with this argument causes it to run the BBM2 algorithm.

TS

```
./pure/pure -i game.nfg -[d:]
```

- `-d delta`: This causes the implementation to run the TS algorithm with the specified value of δ. If this parameter is not passed in, it defaults to the value of 0.1.

KS

```
./pure/pure -i game.nfg
```

KS+

```
./pure/pure -i game.nfg
```

A.3 Game Library

Global parameters

- `-f filename`: Output the generated instance into the specified file using the gambit game format.
- `-g game`: Specifies which game to generate
- `-r seed` (*Optional*): Specify a random seed for the generator to use

Colonel Blotto Games

`./blotto.py -[n:T:r:f:c:]`

- `-n hills`: Indicates the number of hills being fought for
- `-T troops`: Indicates the number of troops for each player
- `-c p`: The covariance value of the multivariate distribution used to generate the players valuation of the hills

Ranking Games

`./gen -g Ranking -[s:]`

- `-s m`: Generates a ranking game where both players have m effort levels

SGC Games

`./gen -g SGC -[s:]`

- `-s m`: Generates an instance of the SGC game where both players have a support size of m in the Nash equilibrium.

Tournament Games

`./gen -g Tournament -[s:k:]`

- `-s n`: Sets the number of nodes in the tournament graph to n
- `-k k`: Sets the size of the subset of nodes to k

Unit Vector Games

`./gen -g Unit -[s:k:]`

- `-s m`: Generates a Unit vector game of size $m \times m$
- `-s m`: If the argument is 0, then generates a Unit vector game which avoids having a pure Nash equilibrium, otherwise, generates a random unit vector game.

Appendix B

Polymatrix Game Generators Library

B.1 File Format

The closest generator for polymatrix games with the exception of this library is the GAMUT library, which has a generator for random polymatrix games. However, due to there being a lack of a standard file format for polymatrix games, the game gets converted to an n-player normal form game before being output. In this section I present the file format which we used for both the generators and algorithm implementations.

Consider the polymatrix game represented in figure B.1, this is represented as a matrix of the form shown in figure B.2. As can be seen from the representation, a significant portion of the matrix is covered by entries of 0 due to there not being an edge between two nodes.

Each file representation of a polymatrix game is split into 2 main sections separated by whitespaces. Although not currently implemented, in the future, we hope to include comments (which would be lines starting with a # symbol) into the file format to aid with readability.

1. The number of players, n.
2. An adjacency matrix representing the graph G.
3. For all pairs of players (i, j) in lexicographic order
   (a) The number of strategies for both players, $|s_i||s_j|$
   (b) The payoff matrices for both players $A_{ij}A_{ji}$

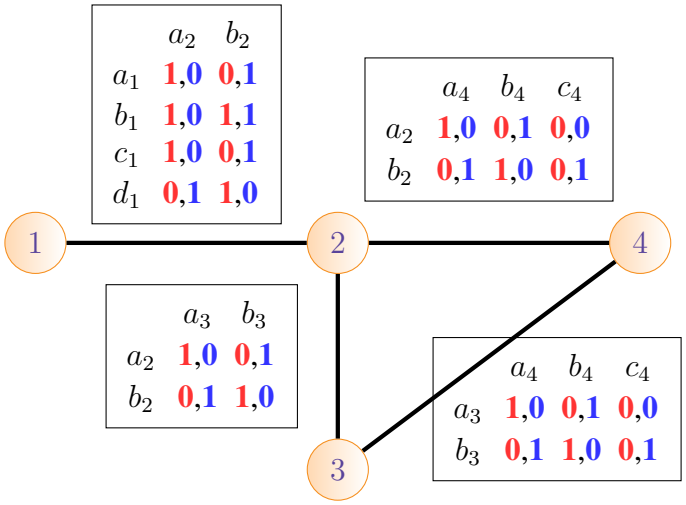


Figure B.1: An example of a 4 player polymatrix game

	a_1	b_1	c_1	d_1	a_2	b_2	a_3	b_3	a_4	b_4	c_4
a_1	0	0	0	0	1	0	0	0	0	0	0
b_1	0	0	0	0	1	1	0	0	0	0	0
c_1	0	0	0	0	1	0	0	0	0	0	0
d_1	0	0	0	0	0	1	0	0	0	0	0
a_2	0	0	0	1	0	0	1	0	1	0	0
b_2	1	1	1	0	0	0	0	1	0	1	0
a_3	0	0	0	0	0	1	0	0	1	0	0
b_3	0	0	0	0	1	0	0	0	0	1	0
a_4	0	0	0	0	0	1	0	0	0	0	0
b_4	0	0	0	0	1	0	0	0	0	0	0
c_4	0	0	0	0	0	1	0	0	0	0	0

Figure B.2: Representation of the polymatrix game in figure B.1

B.2 Game Generators

Two-player Bayesian games

- **Item Bidding Auctions**: These are combinatorial auctions with item bidding. The type of a bidder gives the valuation function for the different subsets of the available items.

- **Multi-unit Auctions**: This is a special case of combinatorial auctions where the items being sold are identical. The type of a player gives the valuation function representing the player's marginal value for recieving a copy of an item.

- **Blotto Games**: The players in this game have a given number of soldiers that are to

```
4

0 1 0 0
1 0 1 1
0 1 0 1
0 1 1 0

4 2
1 0
1 1
1 0
1 1
0 0 0 1
1 1 1 0

2 2
1 0
0 1
0 1
1 0

2 3
1 0 0
0 1 0
0 1
1 0
0 1
```

Figure B.3: File representation of the polymatrix game in figure B.1

be assigned to n hills. Each player has a value for each of the different hills, which is received if the player has assigned more soldiers to the hill than the other player. The utility for each player is then the sum of the valuation gotten on all hills.

- **Adjusted Winner**: In these games, the players wish to split a set of divisible items. The players have preferences for the items expressed by numerical values where the sum of the values over all items sums up to 1. In a similar fashion to Blotto games, the players have to divide a number of points across the available set of items.

Multiplayer Polymatrix games

- **Coordination/Zero-sum Games**: In this category of games, each edge of the underlying graph is either a coordination game (A, A) or a zero-sum game (A, -A).
- **Groupwise Zero-sum Games**: The players are partitioned into groups so that the edges going across groups are zero-sum while those within the same group are coordination games.
- **Strictly Competitive Games**: A two-player bimatrix game is strictly competitive if for every pair of mixed strategy profiles s and s', we have that: if the payoff of one player is better in s than s', then the payoff of the other player is worse in s than in s'.
- **Weighted Cooperation Games**: Each player chooses a colour for a set of available colours (which might not be the same set for each player). The payoff of a player is the number of neighbours who choose the same colour.

B.3 Parameters

Global Parameters

- `-a actions`: Indicates the number of actions available to each player
- `-f filename`: Output the generated instace into the specified file
- `-g game`: Specifies which game to generate
- `-r seed`: Specifies a random seed for the generator

Item Bidding Auctions

- `-A auction type`: Specifies the auction rule, 0 - First price, 1 - Second price, and 2 - All-pay
- `-g Itembidding`: Indicates that the itembidding generators should be used
- `-S`: Indicates that the valuation of player 1 and 2 are the same
- `-t types`: Number of types per player

- `-T tie`: Indicates the tie breaking rule. 0 - First player, 1 - Second player, and 2 - Uniformly at random

- `-v val`: A string representing the valuation of all items for all player types, first set of t valuations are for the first player and the second set of t valuations are for the second player. For each type, the valuation function is the first value, followed by the valuation per item. The following examples show how different valuation functions can be created for three items:

  - Unit demand '0': '0 2 3 4' is a unit demand valuation where the player values the first item at 2, the second at 3 and the third at 4.
  - Single minded '1': '1 1 0 1 3' is a single minded valuation where the player has a value of 3 for getting both the first and third items.
  - Additive '2': '2 2 3 4' is a an additive valuation function where the player values the first item at 2, the second at 3 and the third at 4.
  - Budget Additive '3': '3 2 3 4 5' is a budget additive valuation function where the player values the first item at 2, the second at 3, the third at 4 and has a budget of 5.

Multi-unit Auctions

- `-A auction type`: Specifies the auction rule, 0 - Discriminatory price, 1 - Uniform price, and 2 - All-pay

- `-g Multiunit`: Indicates that the multi-unit generators should be used

- `-t types`: Number of types per player

- `-v val`: A string representing the valuation of all items for all player types. Each value represents the marginal valuation for receiving the next copy of the item. Example:

  - Additive valuation: '5 5 5 5' is an additive valuation function for four items where the value of a single copy is 5.
  - Subadditive valuation: '10 7 7 3' is a sub-additive valuation function for four items.

Graph structure parameters

- `-G graphname`: Specifies the graph structure to be used for the game
- `-m m`: Parameter used to determine the size of the graph
- `-n n`: Parameter used to determine the size of the graph

Examples:
For Complete graphs, m determines the number of nodes on the graph

```
$ ./pm-gen ... -G Complete -m 10 -n 1
```

For Cycle graphs, m determines the number of nodes on the graph

```
$ ./pm-gen ... -G Cycle -m 10 -n 1
```

For Grid graphs, m and n represent the number of rows and columns respectively.

```
$ ./pm-gen ... -G Grid -m 5 -n 2
```

For Tree graphs, m and n represent the branching and depth of the tree respectively

```
$ ./pm-gen ... -G Tree -m 5 -n 3
```

Coordination/Zero-sum Games

```
$ ./pm-gen -g CoordZero [graph options] -p
```

- `-p p`: A value within range [0, 1] representing the proportion of games that are zerosum

Group Zerosum Games

```
$ ./pm-gen -g GroupZero [graph options] -p
```

- `-p p`: Indicates the number of groups

Strict Competition Games

```
$ ./pm-gen -g StrictComp [graph options]
```

Weighted Cooperation Games

```
$ ./pm-gen -g WeightCoop [graph options] -p
```

- `-p p`: A multiplier which indicates the total number of available strategies