Contract Theory and Linear Programming: In Search of Strongly Polynomial-Time Algorithms

Aditya Prasad

Submitted to the University of Southern California in partial fulfillment

of graduation requirements for honors degree

Viterbi School of Engineering

Thomas Lord Department of Computer Science

Spring 2024

Research Advisor: Prof. Shaddin Dughmi

Signature: Shaddin Dughmi

Honors Advisor: Prof. Sandeep Gupta

Signature: Sandeep Gupta

Acknowledgements

First, I would like to thank Shaddin Dughmi for his invaluable guidance throughout my research career. His instructions and advice have inspired me to pursue research in TCS, and he has been an amazing mentor, collaborator, and role model for me.

My collaborators Neel Patel and Ramiro Deo-Campo Vuong have made my research experience both exciting and enjoyable. Their enthusiasm and dedication to research have motivated me during my undergraduate career. Without them, I do not think I would have gained nearly as much as I did from research.

I would like to thank my instructors for providing me with a solid foundation in computer science. In particular, Shaddin Dughmi has been an effortlessly charismatic lecturer and his problem sets, while challenging, have been immensely instructive. David Kempe has been an outstanding instructor and supervisor to me. Working under him, I have discovered a passion for teaching others, and he has provided me with life advice when I was in need. Vatsal Sharan has been an amazing teacher and has consistently motivated me to pursue my interests. Finally, I would like to thank Graciela Elia for her influence on me before USC. Without her, I would never have thought to study computer science and her constant enthusiasm and support of others have made her a role model to me.

I would like to thank the PhD students in Theoroom for their friendship and advice. I want to acknowledge Julian Asilis, Yusuf Kalayci, Neel Patel, Grayson York, and Miryam Huang. I would also like to thank Sid Devic for bringing me to his climbing sessions. He has provided me with great career advice, but more importantly, his friendship helped me feel at home in the lab.

Friends have played a large part in making my undergraduate experience invaluable. David Lee was my first friend at USC and he and Ramiro Deo-Campo Vuong have been consistent companions through problem sets, programming assignments, and parties. Over the past 2 years, Vibhav Laud and Eric Bui have provided me with a sense of home and dragged me out of Theoroom to (occasionally) experience college. Chris Pan and Tate Johnson have been a constant source of support over the past decade of my life.

Lastly, I would like to acknowledge those closest to me. I am grateful to Melody Gui for her constant support throughout the past few months — she has made stressful moments bearable and small milestones unforgettable. Finally, I thank my family for their love and unconditional support over the past 23 years, which have allowed me to mature as a researcher and become a better person.

Note about Overlapping Work in Contract Theory

The contract theory section of this work was done in conjunction with Ramiro Deo-Campo Vuong, Shaddin Dughmi, and Neel Patel. Due to the nature of theoretical computer science research, the contract theory work (both single and multi-agent) presented in this paper is difficult to attribute to any one person in the group and is a product of the efforts of the entire group. In this thesis, I will focus on covering the single agent problem, I will add additional sections with work that I have done individually that was not presented in the [10] paper.

Contents

1 Introduction

This work aims to provide an introduction to strongly polynomial-time algorithms, focusing mainly on the contract theory and linear programming settings. The first section of this thesis aims to define strongly polynomial-time algorithms and provide intuition for its definition. The next section focuses on the single agent contract theory setting, where we derive a strongly polynomial-time algorithm (in the number of sets in demand) for general reward functions. Then, we move to the linear programming (LP) setting, where we present two algorithms in hopes of showing that they solve LPs in strongly polynomial-time. However, we present a failure case for one of the algorithms and conjecture that the other is the same as the simplex method (and thus requires exponential time in the worst case). Finally, we return to the contract theory problem, but in the multi-agent setting, and show NP-hardness of solving the problem optimally and also for different classes of approximations (constant approximation and additive-FPTAS).

1.1 Universal Definitions

As this thesis deals primarily with the strongly polynomial-time algorithms and supermodularity, we present definitions of both terms in Section 2. We attempt to include helpful intuition for the reader through the use of examples. In the following sections, we will use our definitions of strongly polynomial-time and supermodularity to narrow our settings and motivate our results.

1.2 Single Agent Contract Theory

Emerging first in the context of microeconomics [28], contract theory forms a general model for the markets of services. It has received recent attention in CS-Economics field as it naturally combines insights in optimization theory to real-world applications such as crowdsourcing platforms [22], incentivizing quality healthcare [5], pricing of machine learning training services [21], and — as we'll see in our motivating example — project pricing in freelance markets.

1.2.1 A Motivating Example

Consider a setting with two actors, Paige and Alex. Paige (also known as the *principal*) is a small business owner and would like to build a popular and effective website for her business. For concreteness, we will say that Paige believes her site is popular if it receives 100 views on the first day it is published. Paige feels that a popular website is worth \$5000 to her. In our setting, we normalize Paige's reward and say that if she had a popular website, she would get reward 1.[1]

Unfortunately, Paige does not know how to build a website, so she needs to ask Alex (a.k.a., the *agent*), a freelance software engineer, to build it for her. Although Paige does not know how to build the website, she knows of all the n actions one can take to build a website. For example, she knows that when building a website, software engineers can choose to include a front page, a login system, a payment portal, and make sure the website is not buggy, among other things. Further, she knows the normalized cost (or effort) for each of these actions.

Paige hopes to use her knowledge of the actions to set a *contract* $t \in [0, 1]$ that incentivizes Alex to build her website. In our model, a contract specifies that if Alex builds Paige a successful website (according to her criteria), Alex will get a reward t. However, if Alex's work results in an unsuccessful website, Alex and Paige both get 0 reward.[2] Note that t is between 0 and 1 and can be thought of as the percent of the site's value that Paige is willing to give Alex should he build a popular site. For example, if Paige feels that a popular site is worth \$5000, setting $t = 0.5$ means that Paige would pay Alex $.5 * \$5000 = \$2500$ if he builds a popular site.

After Paige sets her contract t, Alex views it and decides to take some set of actions (say, he builds a front page and a payment portal and forgoes doing any other work), for which he pays the price himself,[3] and then sends Paige the completed website. Paige publishes the website and, based on whether 100 people view her site that day, chooses to pay Alex. Of course, whether 100 people

[1]A reward of 1 seems strange, but we think of 1 as a normalized reward or percentage. If Paige values the website at R dollars, we change Paige's unit of value to R and equivalently think of Paige valuing the website as $100\% * R = 1 * R$.

[2]Paige gets no reward because her website was not successful, and thus she cannot afford to pay Alex for his effort.

[3]In fact, we can assume that Paige doesn't even know what actions Alex takes.

view the website is a bit random (perhaps Paige is trending on Instagram, so a worse website might still get 100 views), so we say that there is a *reward* function f that takes the set of actions S that Alex takes as input and outputs the probability that the page is popular.

Naturally, we want this function to reflect that taking more effort cannot hurt, so we say that f must be monotone. That is, if Alex adds an action a to an existing set of actions S, he can do no worse than if he had just selected S, so taking an action can never hurt the probability that the site is popular. Further, to complement known results in the submodular regime, we assume f to be *supermodular*. Informally, supermodularity represents the notion of complementary dependencies. In our setting, for example, the value of a front page in the website is negligible if the website is extremely buggy and there is no payment portal, and vice versa. Thus, each action benefits disproportionately from picking the others as well, displaying the notion of increasing marginal returns.

To summarize:

1. Paige sets a contract $t \in [0, 1]$ for the completed task.
2. Alex views the contract, selects a set of actions $S \subseteq [n]$, and pays the cost of selecting those actions.
3. The task is successful with probability $f(S)$:
   (a) If the task is successful, Paige gets reward 1 and pays Alex t.
   (b) If the task is unsuccessful, Paige and Alex both get reward 0.

We take the role of Paige's advisor and seek to answer the following question: **Given our knowledge of the actions, how do we design a strongly polynomial-time algorithm to determine the contract t that Paige should set to optimize her expected utility?**

Classically, this model was studied when the agent (Alex) could only take one of the $[n] = \{1, 2, \ldots, n\}$ actions[4] [19, 23], meaning that Alex has n choices. In that setting, it was found that

[4]To keep in line with the literature, we enumerate the n actions and let $[n]$ denote the set of all actions. Then, $2^{[n]}$ is

the optimal contract could be found by solving n linear programs (one for each action) to determine the best contract to incentivize each action. Then, one could compare the utility of each of the contracts and pick the one that offers the principal the highest utility.

The combinatorial model differs from the classical one in that the agent (Alex) can take any subset of the $[n]$ actions. In this case, the classical solution technique no longer works — there are 2^n subsets of actions if the agent has n actions to choose from. Even writing down a linear program for each of the subsets would not be a polynomial time operation, so finding the cheapest price to incentivize each set is out of the question. Our techniques, then, must take into consideration more of the problem structure to obtain the optimal contract.

1.2.2 Overview of Results and Techniques

We will see in Section 3 that it becomes prudent to quantify the complexity of the problem based on the number of sets in demand. Informally, a subset of actions $S \subseteq [n]$ is in demand if there is some contract t that incentivizes a rational agent to select it over every other subset S. We observe that for a contract to be optimal, it must be the smallest (i.e., the cheapest) contract for some set of actions $S \subseteq [n]$ that is in demand. Our solution revolves around finding every set S_i in demand, finding the cheapest contract that incentivizes S_i (this contract is called the *break point* or *critical value* for the set S_i), and then putting that value into the set $\mathcal{D}_{f,c}$. Then, if we can find the contract $t \in \mathcal{D}_{f,c}$ that maximizes the principal's utility efficiently, we have found the optimal contract in polynomial time.

We leverage new problem structure to improve upon known weakly polynomial-time algorithms and present two algorithms that enumerate all the break points in strongly polynomial-time with respect to the number of sets in demand. Therefore, we have a strongly polynomial-time algorithm when the number of break points is polynomial.

Theorem 3.5 (Theorem 3.1 in [10]): If the size of the of the break point set $\mathcal{D}_{f,c}$ is polynomial in the number of actions, then there exists a strongly polynomial-time algorithm for computing an

the power set of all n actions (or the set of all subsets of the n actions).

optimal contract in the demand oracle model.

Our result strengthens prior known algorithms by Dütting et al. [14], in which they show a weakly polynomial-time algorithm (also in $|\mathcal{D}_{f,c}|$) that uses a binary search subprocess to enumerate all break points and obtain the optimal contract.

Our algorithms utilize more of the problem structure, focusing on *intersection contracts* — contracts where two sets of actions offer the same utility to the agent. Because the break point for every demanded set must be an intersection contract, our algorithms simply look at these points as potential candidates for break points, thereby forgoing the need for the binary search subprocess.

Next, we show that when the *reward function* is supermodular, the number of sets in demand (and thus the size of the break point set $\mathcal{D}_{f,c}$) is linear in the number of actions.

Lemma 3.6 (Lemma 3.1 in [10]): When the reward function f is supermodular, there can only be at most $n + 1$ sets in demand ($|\mathcal{D}_{f,c}| \leq n + 1$).

Then, since supermodular functions admit a strongly polynomial time demand oracle [24], we can use our general algorithms to obtain strongly polynomial-time algorithms when f is supermodular.

1.2.3 Related Work

The first few works on algorithmic contract theory considered a model in which the agent could take at most one of the n actions [3, 17]. As mentioned earlier, these works constructed a linear program for each of the n actions to calculate the optimal contract. Later, Dütting et al. [13] showed that *linear contracts* — contracts in which the principal offers the agent some percent of her reward — are optimal or approximately optimal in a max-min sense. Linear contracts are generally favored as they are easier for each party to understand and are also easier to compute.

Dütting et al. [12, 14] work in the same model as the one we consider in this thesis (in which an agent selects a subset of the $[n]$ actions). In [14], they focus on setting in which the reward function is *submodular* (i.e., diminishing marginal returns), in which they show hardness for generalized submodular rewards and give a polynomial-time algorithm for gross substitutes rewards. In contrast, Dütting et al. and Deo-Campo Vuong et al. [10, 12] consider settings with supermodular rewards, and give a polynomial-time algorithm for this setting.

1.3 Linear Programming

Linear programming (LP) is a classic optimization problem — with the task being to optimize a linear objective subject to satisfying some set of linear constraints. Though linear programs had been studied as far back as the 1700s, they came under heavy research attention in World War 2 when it was discovered that LPs could be used to formulate critical military objectives. Since then, linear programs have been used extensively — from theoretical computer science research to business applications when determining the optimal allocations of resources to run a company [20].

1.3.1 Motivating Example

Consider taking the role of a car factory manager choosing to manufacture two types of car models: Car A and Car B. Car A takes 2 tons of steel and 8 tons of rubber to manufacture and sells for \$1250. Car B takes 6 tons of steel and 2 tons of rubber to manufacture and sells for \$1000. If we have 30 tons of steel and 32 tons of rubber, how many of Car A and Car B should we make to maximize our revenue?

At first glance, the problem seems quite complicated: Car A takes less steel than Car B to create but uses more rubber. Further, Car A sells for more than Car B, but we might run out of rubber by prioritizing Car A. How should we determine the trade off between making units of Car A vs making Car B?

To break this down, assume that we have already made a units of Car A and b units of Car B. Then, by looking at the selling prices for both cars, we know that we make $R = 1250a + 1000b$ dollars in revenue. Thus, our objective is to $\max R = \max(1250a + 1000b)$.

At this point, we haven't yet considered our resource constraints. If we wanted to maximize R at this point, we should just set a and b to ∞ (meaning we manufacture an infinite amount of both our Car types A and B) and obtain infinite revenue. Of course, this is not possible due to our resource constraints, so we now aim to take them into account in our optimization problem.

First, the steel constraint: a units of car type A will require $2a$ tons of steel, and b units of car type B will require $6b$ tons of steel. In sum, we cannot exceed 30 tons of steel. In other words, we know that $2a + 6b \leq 30$. Using similar reasoning, we can obtain a corresponding constraint for rubber: $8a + 2b \leq 32$. Finally, we observe that we can never make a negative amount of cars of either type: in other words, $a \geq 0$ and $b \geq 0$.

Combining our objective and constraints, we get the following optimization program:

$$\begin{aligned} \text{Maximize: } & 1250a + 1000b \\ \text{Subject to: } & 2a + 6b \leq 30 \\ & 8a + 2b \leq 32 \\ & a, b \geq 0 \end{aligned}$$

With the corresponding geometric interpretation, where the optimal value is the furthest point in the direction of the objective vector (the orange vertex in the top right vertex of our polytope).

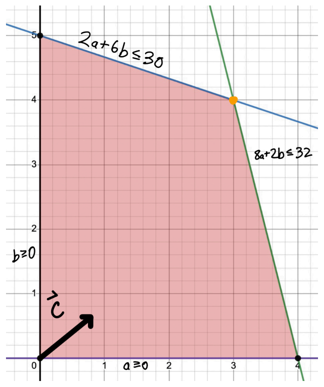

Figure 1: The geometric interpretation of our example LP. Each side of our polytope is associated with one of our constraints, the direction of objective vector is drawn as $\vec{c} = [1250, 1000]$, and the optimal vertex is pictured in orange.

We can observe that the objective and each of the constraints in this program is linear in a and b, so we are dealing with a linear program! Using standard LP solvers, we can find the that optimal revenue obtained is \$7750 when manufacturing 3 of Car A and 4 of Car B.

Through this example, we can observe the versatility of the linear programming framework: we can easily modify the optimization problem to accomodate different prices for the cars (change the coefficents of a and b), add another type of car (add a new variable for the new car in each expression), or even add a new material constraint. At its core, the fundamental program will still be a linear program. Sometimes, even when an optimization problem is not linear, its objective and constraints are close enough that relaxing the equations slightly allows us to find near-optimal solutions quickly. Due to the versatility of linear programming, algorithms that solve LPs see frequent use for real-world problems.

The catch: all known linear program solving algorithms are either weakly polynomial (intuitively, they discretize continuous space at some level), or they have exponential time worst case time complexity. In this thesis, we aim to find algorithms that attempt to solve general linear programs in strongly polynomial time.

1.3.2 Overview of Results

In an attempt to create a strongly polynomial-time algorithm for general linear programs, we will prove a few facts that we will attempt to use in our algorithms. First, we notice that the general optimization problem is as difficult as finding a constraint that is tight at optimality.

Lemma 4.2 (Projection Lemma): Suppose a face or constraint f is tight at optimality. Then, the point maximizing the objective c in f is also the point that maximizes the objective $\textit{proj}_{\mathbf{f}}(\mathbf{c})$ in f.

We will attempt to use this observation to build a strongly polynomial-time algorithm for linear programming, but after showing the algorithm and stepping through its geometric interpretation, we will show a failure case.

Next, we will change our lens to use sets of constraints to define natural upper bounds for the optimization problem.

Observation 4.4 (Upper Bound on Optimal Solutions): The intersection of n constraints $A_1, \dots, A_n$ such that the objective vector c is a conic combination of normal vectors of $A_1, \dots, A_n$ is an upper bound to the optimal solution in the LP.

Again, we will attempt to use this observation to design another strongly polynomial-time algorithm for LPs, but it will prove difficult to show any concrete guarantees. We will end the section with a brief discussion of duality and with our understanding of the dual program, we will make a conjecture on the second algorithm we present:

Conjecture 4.5 (Simplex over the Dual): Algorithm 7 is equivalent to running the simplex algorithm over the dual polytope.

Because the worst-case time complexity of the simplex method is exponential, if Conjecture 4.5 is true, the second algorithm also does not run in strongly polynomial-time.

1.3.3 Related Work

Linear programming has been a mainstay in optimization theory. One of the first polynomial time algorithms for LPs was developed by Karmarkar in [25], which is an interior-point method that iteratively finds better feasible points for the LP until it is close enough to round to the nearest vertex. Interior-point methods quickly grew to incorporate both the primal and dual linear programs: Kojima et al. [27] show that by generating primal-dual feasible pairs, one can iteratively decrease the duality gap between the primal and dual solutions to 0 efficiently.

Separately, Dantzig developed one of the first simplex algorithms [8], which essentially aim to iterate over vertices of the feasible polytope (according to a specific pivot rule) in order to find the one that maximizes the objective function. The worst-case runtime of the simplex algorithm was unknown until Klee and Minty showed an adversarial polytope for which the simplex algorithm was shown to take exponential time [26]. Regardless, the simplex method seemed to be efficient in practice, and Spielman and Teng showed through the use of smoothed analysis that the simplex method *usually* takes polynomial time [29].

The previous papers presented algorithms that run in weakly polynomial-time and thus depend on the precision of the underlying computer. Adler et al. [1] present a strongly polynomial-time algorithm for LPs when the constraint matrix is a *Leontief* matrix. Finally, Tardos [30] presents another general strongly polynomial-time algorithm to determine the optimal vertex when the constraint matrix A is *combinatorial*. Note that a *combinatorial* matrix is one in which the size of each entry of A is bounded polynomially in the dimension of the problem. To our knowledge, no generalized strongly polynomial-time algorithm for linear programs exist and it remains a widely-known open problem to this day.

1.4 Multi-Agent Contract Theory

In Section 5, we return to contract theory, but in the multi-agent setting. Now, there is one principal and n agents each with *exactly* one action. In this model, agents simply choose between taking a costly action or inaction. Although we can consider a model in which the agents choose between sets of actions, we will see that this version of the problem is already NP-Hard, thereby implying that the more general model is also NP-Hard.

1.4.1 Motivating Example

Suppose Paige has built her website and it is a huge success. Now she wants to build a physical store for her business, and she knows that she needs to incentivize some subset of the $[n]$ people to build her store. For example, perhaps she is choosing whether to hire an architect to design blueprints for the store, a set of construction workers, an interior designer, etc. Each agent i again has a cost of action c_i, which for our purposes will be the same across all agents (so $c = c_i, \forall i \in [n]$).[5] In order to incentivize the agents to act, Paige designs an n-dimensional contract $t \in [0,1]^n$ where t_i (the ith

[5]Of course, we can extend our model to incorporate different costs for each agent, but the uniform cost model is already NP-Hard.

component of t) corresponds to the contract that she offers the ith agent.

After Paige sets her contract, the agents each get to view it and determine whether they want to take action (like in the single-agent case, this setting is *hidden action*, meaning we assume that Paige does not know who acts). If an agent chooses to act given his contract, he will pay c. After every agent chooses between action and inaction, the set of active agents S is determined, and the store is built with probability $f(S)$. Again, our reward function $f : 2^{[n]} \to [0, 1]$ is assumed to be monotone and supermodular to reflect the natural complementary dependencies latent in this problem. For example, the value of incentivizing an architect is disproportionately higher when we also incentivize a set of construction workers to build the blueprints, and vice versa.

If the store is built successfully, Paige again receives reward 1, and pays each agent the agreed upon t_i, regardless of whether or not they acted. If the store is not built successfully, Paige and all of the agents receive reward 0.

To summarize:

1. Paige sets a contract $t \in [0, 1]^n$ for the completed task.
2. Each of the $[n]$ agents views the contract and decides whether or not to act. If they choose to act, they pay cost c.
3. The set of active agents is denoted S and the task is completed with probability $f(S)$:
   (a) If the task is successful, Paige gets reward 1 and pays agent i the agreed upon value t_i, regardless of whether or not they acted.
   (b) If the task is unsuccessful, Paige and all of the agents get reward 0.

1.4.2 Overview of Results Discussed

The multi-agent hidden-action setting is known to be significantly more challenging than the single-agent setting [7, 15]. In fact, Dütting et al. in [15] showed that the problem was NP-Hard even when the reward function is *additive*. We contribute to the literature of the multi-agent setting by showing that the supermodular reward setting is NP-Hard and admits no constant approximation nor additive-FPTAS, assuming $P \neq NP$.

Theorem 5.2 (Theorem 4.1 in [10]): The supermodular multi-agent contract problem admits no polynomial time constant multiplicative approximation algorithm nor an additive-FPTAS unless $P = NP$. The hardness holds even for uniform costs and graph supermodular rewards, denoted as U-GSC.

We will focus on the setting in which the n agents can be represented as nodes in a graph $G = (V, E)$, and the reward of a set of nodes S is proportional to the number of edges in the induced subgraph of S. In this thesis, we will attempt to provide an intuitive understanding of Theorem 5.2 by showing that an optimal contract in this setting must identify and incentivize a *dense subgraph* in G, but we refer the reader to [10] for the formal proof.

It is worth noting that Theorem 5.2 does not rule out the possibility of an additive-PTAS, and indeed we find an additive-PTAS for the multi-agent setting with uniform cost and graph supermodular rewards. However, since this result falls a bit out of scope of the thesis, we refer the reader to [10] for the complete treatment of the algorithm.

1.4.3 Related Work

There are two general models of the multi-agent contract problem. In the model studied by Castiglioni et al. [7], agents complete individual tasks, and each outcome is observed by the principal. This model effectively removes externalities between agents, and they show that in in this setting an optimal contract can be exactly computed when the reward function is supermodular and approximated when the reward function is submodular.

The second model is most related to this paper: n rational agents individually decide on exerting effort toward a task, and the principal observes the task's outcome. Babaioff et al. [3] and Emek and Feldman [16] focus on the specific setting in which each agent has an individual outcome, and a boolean function maps the individual outcomes to an overall result of the task. Babaioff et al. give an algorithm to compute the optimal contract for AND networks, and Emek and Feldman show that computing an optimal contract for OR networks is NP-Hard. Dütting et al. in [15] generalize the previous setting and provide a constant factor approximation for XOS reward functions.

Graph Density Our problem setting quickly reduces to a graph density problem, contributing to the rich literature in graph density and normalized Densest k Subgraph. Braverman et al. [6] show that, assuming ETH, N-DkS requires quasi-polynomial time to approximate additively, and Arora et all [2] derive an additive-PTAS when k is linear in the size of the graph by estimating the coefficients to a polynomial program via sampling. Through not discussed in this thesis, the additive-PTAS we derive for the multi-agent setting rely fundamentally on *sampling* randomly and obliviously from the entire graph, drawing on prior work by Barman [4] and Daskalakis et al. [9].

2 Universal Definitions

2.1 Strongly Polynomial

When we encode a problem in a computer, we must encode problem parameters to specify the instance of the problem. When we encode some rational number a as a parameter for a problem, we say that the *length* of a is the number of digits necessary to encode a in its binary representation (in this setup, irrational numbers are said to have infinite length and cannot be represented exactly). The length of a problem instance, denoted either as k or L, is simply the sum of all of the lengths of the problem parameters.

Classically, a polynomial-time algorithm is an algorithm whose number of elementary computations depends polynomially on the size of the input. In other words, the number of elementary computations is allowed to depend both on the *number* of parameters and the *length* of each individual parameter. Elementary operations are assumed to take time $O(1)$, and are limited to addition, subtraction, multiplication, division, and comparison between numbers whose lengths are a polynomial in k. Since the length k is part of the the size of the input, the number of elementary operations of polynomial time algorithms are allowed to depend on k. Algorithms that depend on k are known as *weakly polynomial-time algorithms*.

A *strongly polynomial-time algorithm* is one in which the number of elementary operations is a polynomial function in the size of the input, but it is independent of k. That is, the algorithm is polynomial in the *number* of paramenters, but it is independent of the length of each individual parameter. Next, we provide an example problem and solution to illustrate weakly polynomial-time algorithms.

2.1.1 Example and Stepping Through the Algorithm

Suppose that a rational number $x \in [0, 1]$ is picked at random and then written down into a computer that stores rational numbers with k bits. We are tasked with finding the value of x, given access only to an oracle $\mathrm{Greater} : [0,1] \to \{\text{True}, \text{False}\}$ that takes as input a guess $g \in [0,1]$ and outputs True if $g > x$ and False of $g \leq x$. This problem is solved with binary search (let $\mathbb{Q}$ denote the set of all rational numbers):

Algorithm 1 BINARYSEARCH(α_L, α_R)

```
Require: Underlying computer requires at most $k$ bits to store any number.
  if $\alpha_R - \alpha_L \leq \frac{1}{2^{k+1}}$ then
      Return the unique rational number in $\mathbb{Q} \cap [\alpha_L, \alpha_R)$
  else
      if $\mathrm{Greater}(\frac{\alpha_L+\alpha_R}{2})$ then
          Return BINARYSEARCH$(\alpha_L, \frac{\alpha_L+\alpha_R}{2})$
      else
          Return BINARYSEARCH$(\frac{\alpha_L+\alpha_R}{2}, \alpha_R)$
      end if
  end if
```

Then, we simply call BINARYSEARCH with arguments 0 and 1.

Note that the first if statement is required for BINARYSEARCH to terminate. Without it, we would never be able to narrow down the set of rational numbers in a closed interval to a single number.[6] The runtime of this algorithm is $O(k)$ (because $\mathrm{Greater}$ is an elementary operation), since each iteration of binary search tells us the next digit in the binary representation of x, so it will terminate after exactly $k+2$ calls.

For intuition, let us work through an example setting of this problem. Suppose that our computer uses 3 bits to store numbers between 0 and 1, and the underlying value of the random number x is $.25_{10}$. Its binary representation on our computer would be 0.010_2.

In the first iteration of BINARYSEARCH (with end points 0 and 1), $\mathrm{Greater}(0.5)$ will return True, so we will recurse to call BINARYSEARCH(0, 0.5). Essentially we discover that the first digit in x's binary representation is 0. Thus, our temporary knowledge of the value we are searching for is $x' = 0.0_2$.

In the next iteration of BINARYSEARCH (with end points 0 and $.5$), $\mathrm{Greater}(0.25)$ will return False,

[6]The rational numbers are *dense* in the real numbers, meaning that there will be an infinite number of rational numbers in any interval $[a, b]$ with $a \neq b$.

so we will recurse to call BINARYSEARCH(0.25, 0.5). We learn that the second digit in x's binary representation is 1, so now our working value is $x' = 0.01_2 = 0.25_{10}$.

Although our working value of x' is exactly equal to x, we must continue since we know that our computer can represent two numbers with the same 2 first digits (these two numbers being $0.010_2 = 0.25_{10}$ and $0.011_2 = 0.375_{10}$). In the final iteration, $\text{Greater}(0.375)$ will return True, so we will recurse to call BINARYSEARCH(0.25, 0.375). Now, we have learned that our working value $x' = 0.010_2 = 0.25_{10}$. Since we know that our computer must represent floating point numbers with 3 bits, we know that our working value of x' is exactly equal to x.

In the following recursive call, $\text{Greater}(0.3125)$ will return True and then we call BINARY-SEARCH(0.25, 0.3125). Since the difference between the two bounds is now $\leq \frac{1}{2^{k+1}} = \frac{1}{16}$, we will terminate and return $x = 0.25$.

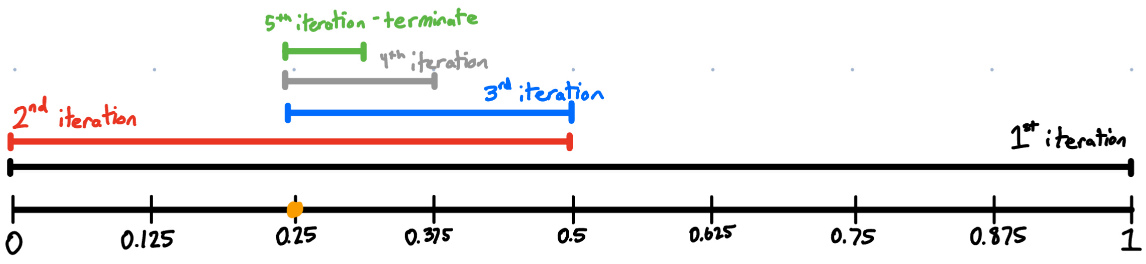

Figure 2: A visual representation of BINARYSEARCH as it finds the target value $x = 0.25$ (pictured in orange). Different colors indicate the search intervals on different iterations of the algorithm. BINARYSEARCH terminates on the 5th iteration (pictured in green) because the possible interval is smaller than the precision of our 3-bit computer. In other words, the green interval is smaller than the range between 2 ticks on the number line, so we can round to the only possible value still within our search interval: $x = 0.25$.

2.1.2 Discussion of Strong vs Weak Polynomial Time

Notice that the runtime of our algorithm depends on k, the number of digits it takes to encode x in binary, which is a property of the precision of the computer on which the program is running. Even if x was the same value, if we were running on a computer that represented numbers using 32 bits, we would not be able to terminate until after 34 iterations of our algorithm.

Weakly polynomial-time algorithms often rely on discretizing continuous space – at the end of our previous example, there are an infinite number of rational numbers in between 0.25 and 0.3125. It was only our realization that only one of those rational numbers could be exactly represented in k bits that allowed us to stop and recognize that $x = 0.25$.

Strongly polynomial-time algorithms must forgo the (often easier) notion of discretizing continuous space to remove their dependence on k. In the process, these algorithms often end up using more problem structure, and thus they are generally favored in TCS research.

2.2 Supermodularity

Supermodular functions, often studied in the context of economics or game theory, are set functions that satisfy the following.[7]

Definition 2.1. *A set function $f : 2^{[n]} \to \mathbb{R}$ defined over the set of $[n]$ objects is supermodular if and only if:*

$$f(B \cup \{i\}) - f(B) \geq f(A \cup \{i\}) - f(A)$$

for $A \subseteq B \subseteq [n]$ and $i \notin B$. Equivalently, f is supermodular if and only if:

$$f(A \cap B) + f(A \cup B) \geq f(A) + f(B) \tag{1}$$

for all $A, B \subseteq [n]$.

That is, an element's *marginal contribution* in the value of a roughly "larger" (formally, a superset) set of items is more than its marginal contribution to the value of "smaller" sets of items. Intuitively a function f is considered *supermodular* if it satisfies the notion of *increasing marginal returns*.

For an instructive example, let f denote the number of possible edges in a graph $G = (V, E)$ with

[7]The proof that the following definitions of supermodularity are equivalent is well-known and left as an exercise for the reader. If necessary, it is also given in [18].

$|V| = n$ nodes. Then, a bit of math (handshaking lemma) shows us that:

$$f(V) = \sum_{i \in V} \frac{\deg i}{2} = \sum_{i \in V} \frac{n-1}{2} = \frac{n(n-1)}{2}$$

Note that f is supermodular because if we add a node u to a graph of n nodes, we add another n potential edges, so u's marginal contribution to graphs with more vertices is larger than its contribution in graphs with fewer vertices.

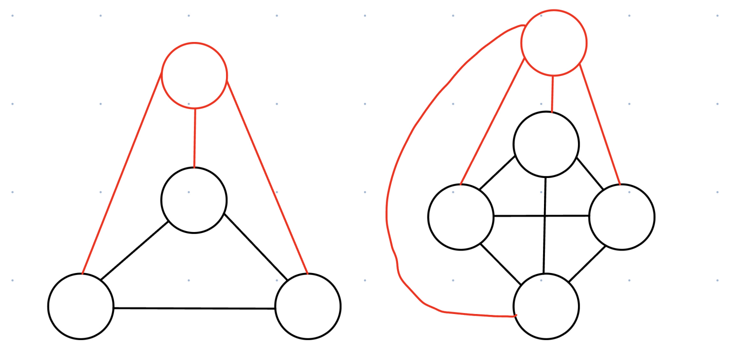

Figure 3: A visual representation of our example supermodular function f. Consider adding the node in red to the graph on the left versus the graph on the right. The marginal utility of the node in red is the number of edges it adds to each graph (i.e. the number of red edges in each graph). The marginal utility of the node in red is larger for supersets of nodes, so f is supermodular.

Supermodular functions are often used to model the notions of complementary actions. For example, when building a house, building a kitchen offers much more utility when the house has a foundation, so the utility of building both is greater than the sum of each individually. Similarly, economies of scale exhibit supermodularity, since each additional unit offers more utility than the last.

For our purposes, we simply need to know that supermodular functions can be maximized in strongly polynomial-time [24], which we will use to develop an efficient demand oracle (defined in Section 3.1) that we use as a subprocess in our strongly polynomial-time algorithm.

3 Single Agent Combinatorial Contracts

3.1 Problem Setting

In the single agent combinatorial contracts setting, there is a ground set of costly actions $[n] = \{1, \dots, n\}$. We denote the cost of action i with c_i and let the cost of some subset of actions $S \subseteq [n]$ be $c(S) = \sum_{i \in S} c_i$.

The principal's utility when the contract is set to t and the agent chooses to take actions $S \subseteq [n]$ is:

$$\mu_p(S, t) = f(S) - t \cdot f(S) = (1 - t) f(S). \tag{2}$$

The agent's utility when the contract is set to t and they perform some subset of action actions $S \subseteq [n]$ is:

$$\mu_a(S, t) = t \cdot f(S) - c(S), \tag{3}$$

When a contract t is set, a rational agent's best response is to select the set of actions $S \subseteq [n]$ that maximizes $\mu_a(S, t)$. We assume that when two sets S, S' maximize the agents utility and offer the agent the same utility at contract t ($\mu_a(S, t) = \mu_a(S', t)$), then the agent is inclined to tie-break in favor of the principal. Therefore, they will select the set with higher f value.

Combining the above definitions, we can write the task of finding the optimal contract for the principal as the following optimization problem (we are finding t):

$$\begin{aligned} \max_{t \geq 0, S \subseteq [n]} \quad & \mu_p(S, t) \\ \text{s.t.} \quad & \mu_a(S, t) \geq \mu_a(S', t), \qquad \forall S' \subseteq [n] \end{aligned}$$

We assume access to a *value oracle*, which is denoted as f, which simply takes as input some set of actions S and outputs $f(S)$ (for our purposes, the value oracle is the same as the reward function).

We also assume access to a *demand oracle*, denoted Φ, which is commonly used in algorithmic game theory. Formally, the *demand oracle* takes as input a vector of prices $p \in \mathbb{R}^n$ and outputs a set $S \subseteq [n]$ that maximizes $f(S) - \sum_{i \in S} p_i$. Again, we assume that if two sets have the same maximal value, the demand oracle outputs the set with higher f.

Then, it can be shown that the agent's problem of maximizing his utility can be solved by calling the demand oracle Φ with prices $p_i = c_i/t$. To see this, suppose S maximizes the agent's utility at contract t. That is:

$$\begin{aligned} f(S) * t - \sum_{i \in S} c_i &\geq f(S') * t - \sum_{i \in S'} c_i, \quad \forall S' \subseteq [n] \\ \implies f(S) - \sum_{i \in S} \frac{c_i}{t} &\geq f(S') - \sum_{i \in S'} \frac{c_i}{t}, \quad \forall S' \subseteq [n] \end{aligned} \tag{4}$$

Where the set satisfying 4 can be found by calling the demand oracle with prices $p_i = c_i/t$. For this reason, we will slightly abuse notation and say that the demand oracle Φ takes as input some $t \in [0, 1]$ and outputs the set S that the agent will take when the contract is set to t.

Although the demand oracle is not efficiently computable in general, it can be computed in strongly polynomial-time when when f is supermodular [24]. We will use this to derive a strongly polynomial-time algorithm to compute the optimal contract for the principal.

Finally, we say S is *in demand* at contract t if $S = \Phi(t)$ (with tie-breaking). More generally, we say that S is *in demand* if there exists a contract $t \in [0, 1]$ such that $S = \Phi(t)$.

3.2 Problem Structure

If we fix the agent to take a set of actions $S \subseteq [n]$, the agent's utility is $\mu_a(S, t) = f(S)t - c(S)$. Observe that the agent's utility for fixed set of actions S is linear with respect to t (as $f(S)$ and $c(S)$ are taken as constant). The general agent utility curve $\mu_a(t) = \max S \subseteq [n] f(S)t - c(S)$ is then a maximization over a set of linear functions, which is well known to be a convex, nondecreasing, piece-wise linear function. Further, each piece of the agent's utility curve (also called the demand curve) corresponds to a unique subset of actions $S \subseteq [n]$ that is in demand, and we let $\mathcal{D}_{f,c} = \{S \subseteq [n] : \exists t \in [0,1] \text{ s.t. } S = \Phi(t)\}$ denote the set that contains all sets in demand.

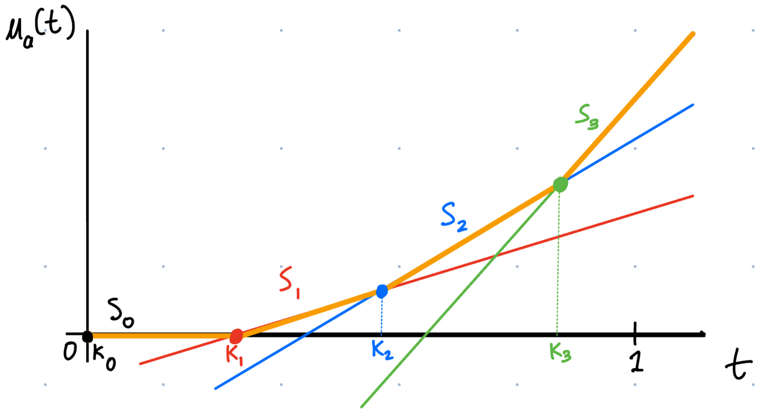

Figure 4: An example agent demand curve is pictured in orange. In the figure above, sets $S_0, S_1, S_2,$ and S_3 are in demand and the linear utility curve for each set is represented with corresponding colors. The break point for each set $\kappa_1, \kappa_2, \kappa_3, \kappa_4$ is given (also color-coded).

If we let $T_S \subseteq [0,1]$ denote the set of contracts for which S is in demand, we can observe that if the principal is constrained to incentivize S, she must pick $\kappa_S = \min_{t \in T_S} t$. This corresponds to a *break point* of the agent's demand curve, and we call κ_S the break point of set S. Dütting et al. [13, 14] show that the optimal contract is always a break point of the agent's demand curve, so it suffices to check each break point and select the break point that maximizes the principal's utility.

We define an important notion of an intersection contract *IC*(L, R) between two sets in demand $L, R \subseteq [n]$. We let the intersection contract denote the contract at which two sets of actions provide the agent the same utility (*IC*(L, R) is precisely the x-value of the point of intersection of the two lines associated with L and R). Note, that when we call the demand oracle with value *IC*(L, R), the demand oracle will tie-break in favor of the one with higher f value (or slope), and thus it will always return R.

Formally:

$$IC(L, R) = \frac{c(L) - c(R)}{f(L) - f(R)}$$

We note that every break point in the agent's demand curve is an intersection contract, but not every intersection contract is a break point. For ease of notation, we define an ordering over the sets in demand.

Definition 3.1 (Ordering on Sets in Demand)**.** *We define a strict total ordering $\prec$ on sets in demand as follows: Set L precedes set R, denoted as $L \prec R$, means that $f(L) < f(R)$ or, equivalently, $\kappa_L < \kappa_R$.*

Note that this definition lends itself to the observation that as we increase the contract t, the reward for the set in demand at t monotonically increases.

Observation 3.2. *If L and R are both in demand, and L is in demand at contracts smaller than R, then $L \prec R$ (or $f(L) < f(R)$).*

We call two sets L, R that are in demand *adjacent* if there does not exist another set in demand S such that $f(L) < f(S) < f(R)$ (or equivalently $L \prec S \prec R$). Further, the intersection contract of two non-adjacent sets in demand yields a contract at which a new set is in demand.

Observation 3.3. *Let $L, R \in \mathcal{D}_{f,c}$ with $L \prec R$. The break point of R, denoted as κ_R, is the intersection contract of L and R if and only if L and R are adjacent.*

Observation 3.4. *Let L and R be two non-adjacent sets in demand such that $L \prec R$. Then, the set in demand S at contract $t = IC(L, R)$ satisfies $L \prec S \prec R$.*

3.3 Algorithms

The general theory for algorithms for the single agent combinatorial contracts problem is to first find the break point of every set in demand. Then, simply iterating over each break point to find the one maximizing the principal's utility yields us the optimal contract. The runtime of these algorithms is considered acceptable if it is bounded polynomially in the number of sets in demand $|\mathcal{D}_{f,c}|$.

The main result of this section is the following:

Theorem 3.5. *If the size of the break point set $|\mathcal{D}_{f,c}|$ is polynomial in the number of actions, then there exists a strongly polynomial-time algorithm for computing an optimal contract when given access to an efficient demand oracle.*

3.3.1 A Known Weakly Polynomial-Time Algorithm

We will briefly cover a known method to solve the problem. This algorithm was presented in [14]. The purpose of Algorithm 2 is to find the next break point after some given break point α (initially 0). To find the next break point, we simply call SEARCH(α, 1). Note that $\mathbb{Q}$ is simply the set of rational numbers.

Algorithm 2 SEARCH(α_L, α_R) from Dütting et al. [14]

```
Require: Poly-time demand oracle $\Phi$ that tiebreaks in favor of higher success probability $f$
Require: Underlying computer requires at most $k$ bits to store any number.
  if $\alpha_R - \alpha_L \leq \frac{1}{2^{2k}}$ then
      Return the unique element of $\mathbb{Q} \cap (\alpha_L, \alpha_R)$
  else
      $S_{left} = \Phi(\alpha_L)$, $S_{middle} = \Phi(\frac{\alpha_L+\alpha_R}{2})$
      if $f(S_{middle}) > f(S_{left})$ then
          Return SEARCH$(\alpha_L, \frac{\alpha_L+\alpha_R}{2})$
      else
          Return SEARCH$(\frac{\alpha_L+\alpha_R}{2}, \alpha_R)$
      end if
  end if
```

Algorithm 2 makes use of Observation 3.2 in order to find the next critical value. Because we can find the reward of the set in demand at the current break point ($f(S_{left})$), we can use this as our query to see if we should look at the right half of the interval or the left half of the interval. If the set in demand at the middle of the interval S_{middle} is the same as S_{left}, then we know that we should look on the right half of our search space for the next break point after α. Otherwise, we know to look on the left half of our search space for the next break point after α.

Observe that the Algorithm 2 is very similar to Algorithm 1, presented in Section 2 as an example of a weakly polynomial time algorithm.[8] Indeed, SEARCH is a weakly polynomial-time algorithm

[8]In fact, it is exactly BINARYSEARCH, but with a different oracle.

for much the same reasons as BINARYSEARCH is — there is always an infinite amount of numbers between α_L and α_R, so the only way reason for us to stop is if the search interval is too precise to be represented by our computer. Thus, the runtime of this algorithm is dependent on the number of bits k (it runs for $2k+1$ iterations), and thus the algorithm runs in weakly polynomial-time.

3.3.2 An Initial Strongly Poly-time Algorithm for Instances with Poly-Size Critical Sets

Now, we show how to use *intersection contracts* to remove the dependence on k.

Find Break Points. We define a Newton's method-like algorithm to find the break point of $S \subseteq [n]$.

Algorithm 3 FINDBREAKPOINT(S)

```
Require: Poly-time demand oracle $\Phi$ that tiebreaks in favor of higher success probability $f$
Require: S in demand at some $0 \le t \le 1$
  $S' \leftarrow \emptyset$
  while $f(S') < f(S)$ do
      $t' \leftarrow IC(S', S)$
      $S' \leftarrow \Phi(t')$
  end while
  return $t'$ if $S = S'$, else return $NULL$
```

Proof of Termination. It suffices to show that for all iterations, f strictly increases. Consider S' at some non-terminating instance of our algorithm. Let $t_{S'}$ be an arbitrary contract such that S' is in demand (that is, $\Phi(t_{S'}) = S'$). Observe that $t_{S'} < t'$ because for every contract t_u larger than t', S offers more utility than S'. That is, $\mu_a(S, t_u) \geq \mu_a(S', t_u) \implies S' \neq \Phi(t_u)$.

Thus, t' is a (possibly slack) upper bound on the values $t_{S'}$ for which S' is in demand. Then, by Observation 3.2, $f(\Phi(t_{S'})) \leq f(\Phi(t'))$. Since no two sets in the agent's demand curve can share values of f, $f(\Phi(t_{S'})) < f(\Phi(t'))$. S' changes each iteration and since $f(S'_i) < f(S'_{i+1})$ (Observation 3.4), we know that S' never repeats sets. Our algorithm terminates since there are a finite number of sets in demand.

Proof of Correctness. Let S_{-1} be S' prior to the last iteration of the algorithm. Then, t' at the output of the algorithm is $t' = \boldsymbol{IC}(S_{-1}, S) = \frac{c(S)-c(S_{-1})}{f(S)-f(S_{-1})}$. First, we show that for any $t_l = t' - \epsilon$ for $\epsilon > 0$, $S \notin \mathcal{D}(t_l)$.

$$\mu_a(S_{-1}, t_l) > \mu_a(S, t_l)$$
$$f(S_{-1})t_l - c(S_{-1}) > f(S)t_l - c(S)$$
$$f(S_{-1})(t' - \epsilon) - c(S_{-1}) > f(S)(t' - \epsilon) - c(S)$$
$$f(S_{-1})t' - c(S_{-1}) - f(S_{-1})\epsilon > f(S)t' - c(S) - f(S)\epsilon \tag{5}$$
$$-f(S_{-1})\epsilon > -f(S)\epsilon \tag{6}$$
$$f(S_{-1}) < f(S) \tag{7}$$

Line 6 holds since, for the last iteration of the algorithm, $f(S_{-1})t' - c(S_{-1}) = f(S)t' - c(S)$ and we can cancel them in line 5. Line 7 is true because f strictly increases every iteration. Thus, S is not in demand at any contract smaller than t.

At termination, $\Phi(t') = S' = S$, so S is in demand at t' at the termination of the algorithm. Thus, t' is the smallest value that incentivizes set S, so $t' = \kappa_S$ is the break point of S.

Runtime. In each iteration, the demand oracle is called once to determine the set in demand at the new value of t'. In the worst case, the number of iterations is the number of sets in demand before S (the number of sets demanded at $t < t'$, where t' is its value at the output of the algorithm). Formally, the runtime of our algorithm is $O(|\mathcal{D}_{f,c}| * \text{Runtime of demand oracle})$.

Maximizing Principal Utility. Now that we have a polynomial time subprocess for finding the critical value given a set S in demand, we can use it in an algorithm to find the maximum principal utility. Let $\epsilon > 0$ be such that $t - \epsilon$ is less than a bit away from t.[9]

This algorithm finds the critical values for all the sets in demand by the agent and returns the one maximizing principal utility. Its runtime is $O(|\mathcal{D}_{f,c}| * \text{Runtime of } \textsc{FindBreakPoint}(\text{S})) = O(|\mathcal{D}_{f,c}|^2 * \text{Runtime of demand oracle})$.

[9]This is to ensure that $\Phi(t) \neq \Phi(t - \epsilon)$, but we don't skip any sets in demand.

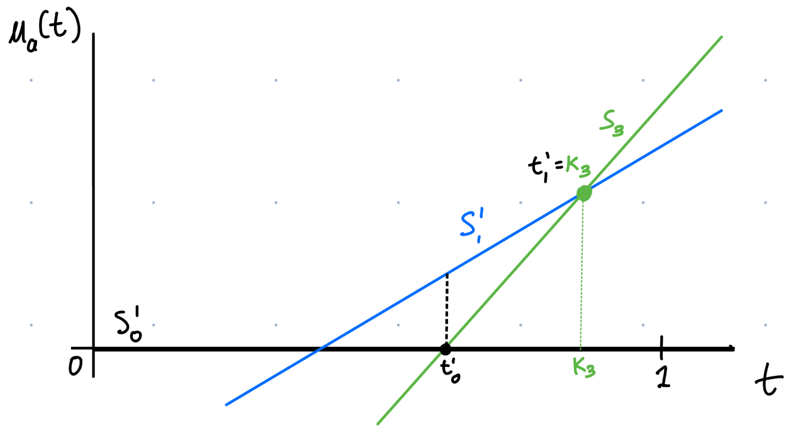

Figure 5: A visual representation of FINDBREAKPOINT as it finds break point κ_3 (pictured in green). Different colors indicate the S' and t' values on different iterations of the algorithm. FINDBREAKPOINT terminates on the 2nd iteration (pictured in green), because κ_3 is found. Observe that $f(S')$ increases every iteration, and not every set in demand must be found before the critical value is found.

Algorithm 4 SEARCHOPTIMAL

```
Require: Polytime demand oracle $\Phi$ that tiebreaks in favor of higher success probability $f$
  $O \leftarrow \emptyset$
  $t \leftarrow 1$
  $S \leftarrow \Phi(t)$
  while $t > 0$ do
      $t \leftarrow$FindBreakPoint$(S)$
      $O \leftarrow O \cup (S, t)$
      $t \leftarrow t - \epsilon$
      $S \leftarrow \Phi(t)$
  end while
  return $(S, t) \in O$ maximizing $\mu_p(S, t)$
```

3.3.3 An Optimized Strongly Polynomial Time Algorithm [10]

Finally, we present an optimized version of Algorithm 3 that was published in [10, 11]:

Algorithm 5 utilizes the same basic observations as Algorithm 3, but in a more streamlined manner. Specifically, it removes the need for Algorithm 4 by using a divide and conquer approach to find all of the sets in demand. If we know the sets in demand for two "boundary" sets L and R (initially, the sets in demand at $t = 0$ and $t = 1$), then we can find their intersection contract using *IC*(L, R). Then, there are two cases, either L and R are adjacent and κ_R is the intersection contract, or

[10]The discussion of this algorithm is unique to this work, but the algorithm presented in this section is joint work with Ramiro Deo-Campo Vuong, Shaddin Dughmi, and Neel Patel.

Algorithm 5 BREAKPOINT(L, R) from [10]

```
Result: A set of break points.
Input: $L, R \subseteq [n]$ in demand with $f(L) < f(R)$.
Require: Access to an agent oracle $\Phi$ (derived from a demand oracle), value oracle $f$, and cost function $c$.
  $t_S \leftarrow \textit{\textbf{IC}}(L, R)$
  $S \leftarrow \Phi(t_S)$
  if $f(S) = f(R)$ then
      return $\{t_S\}$
  else
      return BREAKPOINT$(L, S) \cup$ BREAKPOINT$(S, R)$
  end if
```

a new demanded set is found upon which our algorithm recurses. Using this observation, we can cut the runtime of finding the optimal contract from $O(|\mathcal{D}_{f,c}|^2 *$ Runtime of demand oracle$)$ to $O(|\mathcal{D}_{f,c}| *$ Runtime of demand oracle$)$. We refer the reader to [10] for a detailed proof of the algorithm.

3.4 Results for Supermodular f[11]

Now that we have an algorithm that obtains the optimal contract in strongly polynomial-time in the number of sets in demand $|\mathcal{D}_{f,c}|$, we show that there can there can only be at most $n + 1$ sets in demand when f is supermodular. Intuitively, when it becomes profitable to include some item i to a set S in demand, it only becomes disproportionately more profitable to include i in supersets of S. Thus, all sets in demand for larger contracts should also include item i, meaning sets in demand for larger t should be supersets of sets in demand for smaller t. We include the proof from our work in Deo-Campo Vuong et al. [10].

Lemma 3.6. *When f is supermodular, there can be at most $n + 1$ sets in demand ($|\mathcal{D}_{f,c}| \leq n + 1$).*

Proof (from [10]). Let S_1 and S_2 be sets in demand for contracts t_1 and t_2, respectively. Given that $t_1 + k = t_2$ for some $k \in \mathbb{R}_{>0}$, it must be true that $S_1 \subseteq S_2$. For contradiction, assume that this is not the case and consider the utility the set $S_1 \cup S_2$ at contract t_2 provides for the agent.

[11]Though our discussion of results is unique to this work, the results presented in this section are joint work with Ramiro Deo-Campo Vuong, Shaddin Dughmi, and Neel Patel.

$\mu_a(S,t) = f(S)t - c(S)$ is the agent utility function.

$$\begin{aligned}
\mu_a(S_1 \cup S_2, t_2) &= f(S_1 \cup S_2)t_2 - c(S_1 \cup S_2) \\
&\geq [f(S_1) + f(S_2) - f(S_1 \cap S_2)]t_2 - [c(S_1) + c(S_2) - c(S_1 \cap S_2)] \\
&= \mu_a(S_2, t_2) + \mu_a(S_1, t_2) - \mu_a(S_1 \cap S_2, t_2) \\
&= \mu_a(S_2, t_2) + [\mu_a(S_1, t_1) - \mu_a(S_1 \cap S_2, t_1)] + [f(S_1)k - f(S_1 \cap S_2)k] \\
&\geq \mu_a(S_2, t_2)
\end{aligned}$$

By supermodularity of f, lines 1 and 2 hold. Invoke the fact that S_1 is in demand and supermodularity to go from lines 4 to 5. In the event that the utility between S_1 and $S_1 \cup S_2$ is equivalent and $S_1 \cup S_2 \neq S_2$, then $S_1 \cup S_2$ will be in demand because it provides a larger success probability (by monotonicity). This results in a contradiction. S_2 cannot be in demand at contract $t_2 \geq t_1$ unless it is a super set of the S_1.

As the contract t increases, each new demand set contains at least one more item along with all items previously in demand, so there are at most $n+1$ different sets in demand.

Combining Lemma 3.6, Theorem 3.5, and the fact that a demand oracle for supermodular functions can be constructed in strongly polynomial-time [24], we obtain that the optimal contract for the principal can be computed in strongly polynomial-time when the reward function f is supermodular.

4 Linear Programming

4.1 Setting and Problem Statement

The classical definition of linear programming is defined with objects $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $x \in \mathbb{R}^n$, and $c \in \mathbb{R}^n$ and solves the program:

$$\begin{aligned} \text{Maximize: } & c \cdot x \\ \text{Subject to: } & Ax \preceq b \end{aligned}$$

Written out more explicitly, when given $x, c, A_1, \ldots, A_m \in \mathbb{R}^n$ and $b_1, \ldots, b_m \in \mathbb{R}$, solve:

$$\begin{aligned} \text{Maximize: } & c \cdot x \\ \text{Subject to: } & A_1 \cdot x \leq b_1 \\ & A_2 \cdot x \leq b_2 \\ & \vdots \\ & A_m \cdot x \leq b_m \end{aligned}$$

In our setting, we will only focus on solving linear programs with only *effective* or *non-redundant constraints*.

Definition 4.1. *We call a constraint A of the form $a \cdot x \leq b$ effective or non-redundant if and only if $\exists x \in \mathcal{P}$ such that $a \cdot x = b$ and there exists an $n-1$-dimensional sphere S centered on x with radius $\epsilon > 0$ such that every point in $p \in S$ satisfies $a \cdot p = b$.*

Informally, *effective* constraints define $n-1$ dimensional faces of the polytope $\mathcal{P}$, so we are only focused on finding solutions for LPs where every constraint defines a face of the polytope $\mathcal{P}$.

4.2 Objective

Our goal in this work is to find an algorithm to find the optimal point $x^* \in \mathcal{P}$ in strongly polynomial-time in n, the number of dimensions, and m, the number of constraints. Thus, we would like to remove any dependence on the number of bits it takes to store an integer L. Known methods that run in polynomial time (ellipsoid method, interior point method, etc.) currently depend on the bit complexity of an integer L to determine when to stop. Methods that do not require this dependence (such as the simplex method) run in exponential time in the worst case (via variants of the Klee-Minty cube originally presented in [26]), but have been shown to run in polynomial time in most cases using smoothed analysis [29].

4.3 Building Intuition

Throughout this work, we adopt the physical interpretation of linear programming. In this interpretation, c defines the direction of gravity and A defines a feasible polytope $\mathcal{P}$ (in this thesis, we will use A, $\mathcal{P}$, and $A_1, \ldots, A_m$ interchangeably when it makes sense to do so). Then, consider letting go of a ball anywhere inside the polytope. The position of the ball when it is at rest is precisely the optimal solution to the program.

4.4 A First Algorithm: Naively Tightening the "Tightest" Constraint

4.4.1 Motivation and Algorithm

Our first algorithm attempts to conjecture that the constraint that is most normal to the objective vector c (our analogue for the direction of gravity) is always tight at optimality. We make the observation that when a constraint is known to be tight at optimality, it suffices to maximize the projection of the objective onto the tight constraint, while restricting ourselves to solutions that set that constraint tight.

Lemma 4.2 (Projection lemma)**.** *Suppose face f is tight at optimality. Then, the point maximizing* *the objective c in f is also the point that maximizes the objective $proj_{\mathbf{f}}(\mathbf{c})$ in f.*

Translate f to contain the origin. Any point p on f is orthogonal to $b_{\perp}$. Suppose c is not spanned by f. Then, c can be written as $c = \delta proj_{\mathbf{f}}(\mathbf{c}) + \lambda f_{\perp}$ for $\delta, \lambda \geq 0$. Consider an arbitrary x on f:

$$\begin{aligned} c \cdot x &= (\delta\, proj_{\mathbf{f}}(\mathbf{c}) + \lambda f_{\perp}) \cdot x \\ &= \delta(proj_{\mathbf{f}}(\mathbf{c}) \cdot x) + \lambda(f_{\perp} \cdot x) \\ &= \delta\, proj_{\mathbf{f}}(\mathbf{c}) \cdot x \end{aligned}$$

Therefore, solving $\arg\max_{x \in \mathcal{P}} c \cdot x = \arg\max_{x \in \mathcal{P} \cup f} c \cdot x = \arg\max_{x \in \mathcal{P} \cup f} proj_{\mathbf{f}}(\mathbf{c}) \cdot x$ since $\delta > 0$.

Thus, once a face f is known to be tight, we can restrict our search space to the solutions tight at f. Then, the Projection Lemma (4.2) states that the component of c that is orthogonal to f does not contribute to the objective value of these solutions, so we can modify the objective vector c to $proj_{\mathbf{f}}(\mathbf{c})$.

We now formalize our conjecture that the tightest effective constraint must be tight at optimality.

Conjecture 4.3 (Tightest effective constraint)**.** *Let $A^* = \arg\max_{A_i \in \{A_1, \ldots, A_m\}} A_i \cdot c$ be the tightest constraint with respect to objective vector c (assume all constraint vectors are normalized). Then, there must be an optimal solution that sets A^* tight.*

Lemma 4.2 and Conjecture 4.3 seem to lend themselves to a simple algorithm, which seems to work for all 2-dimensional linear programs.

At each of its n iterations, Algorithm 6 checks each of the m constraints to find the one that is tightest with the current version of c. Then, it adds the tightest constraint to the set T and projects c onto the tightest constraint (this is an elementary operation). After the n constraints are tight, we solve a system of equations to find the point that sets all the constraints in T tight at the same time. Therefore, the runtime of Algorithm 6 is bounded by finding the n tightest constraints iteratively, and is thus $O(mn)$.

Algorithm 6 TIGHTENCONSTRAINTS

```
Result: A vertex solution to the given LP.
Input: Objective vector $c \in \mathbb{R}^n$ and constraint matrix $A \in \mathbb{R}^{m \times n}$.
Require: Every constraint must be effective (i.e. define a face of $\mathcal{P}$).
  $T \leftarrow \emptyset$
  while $|T| < n$ do
      Let $A_i$ be $\arg\max_{A_i \in \{A_1,\ldots,A_m\}} A_i \cdot c$
      $T \leftarrow T \cup A_i$
      $c = \mathit{proj}_{\mathbf{f}}(\mathbf{c})$
  end while
  Return the unique $x$ that sets all the constraints in $T$ tight.
```

4.4.2 Failure case

Although it can be shown that TIGHTENCONSTRAINTS returns the optimal vertex in 2 dimensions, Algorithm 6 fails in general because Conjecture 4.3 is not true for dimensions larger than 2.

Consider the following 3 dimensional instance:

$$\text{Maximize:} \quad -z$$

$$\text{Subject to: } x - 2z \leq 3 \tag{8}$$

$$5y - 2z \leq -8 \tag{9}$$

$$x - 4y - 2z \leq -8 \tag{10}$$

$$-x \leq 50 \tag{11}$$

$$z \leq 8 \tag{12}$$

A bit of math shows us that the "tightest" constraint, corresponding to the vector whose normalized dot product with c is largest, is Constraint 8. In setting Constraint 8 tight, our conjecture and our algorithm fails, as the optimal solution x^* sets tight Constraints 9, 10, and 11 and obtains $x^* = (-50, -22/3, -43/3)$.

4.5 Algorithm 2: A Constraint-Based Approach

4.5.1 Motivation and Algorithm

Our second algorithm focuses on *conic combinations* of constraints. Formally, a conic combination of some set of vectors $v_1, \ldots, v_n$ is $v = \omega_1 v_1 + \cdots + \omega_n v_n$, where $\omega_i \geq 0$ for every i. In this case, we also say that v is a conic combination of $v_1, \ldots, v_n$ and that $v \in \text{Cone}(\{v_1, \ldots, v_n\})$.

In our setting, we consider the vectors normal to the faces of our polytope $\mathcal{P}$. We say that $v \in \text{Cone}(A_1, \ldots, A_k)$ if v is a conic combination of the vectors perpendicular to those faces. Formally, $v \in \text{Cone}(A_1, \ldots, A_k)$ implies that there exists some non-negative weights $\omega_1, \ldots \omega_k \geq 0$ such that $v = \sum_{i=1}^{k} \omega_i A_i$.

Consider the faces $f_1, \ldots, f_n$ that are tight at the optimal vertex x^*. It is well known that the objective vector $c \in \text{Cone}(f_1, \ldots, f_n)$. We make a further observation that if we consider some set of n effective constraints $f_1, \ldots, f_n$ such that c is a conic combination of those constraints, then their intersection point x' is an upper bound on the value of the optimal solution x^*.

Observation 4.4 (Upper Bound on Optimal Solutions)**.** *Consider some set of n effective constraints $A_1, \ldots, A_n$ such that $c \in \text{Cone}(\{A_1, \ldots, A_n\})$, then their intersection point x' is an upper bound on the value of the optimal solution x^*.*

Proof. $c \in \text{Cone}(\{A_1, \ldots, A_n\})$, so $c = A_1\omega_1 + ... + A_n\omega_n$, where $\omega_i \geq 0$. Notice that the $\vec{\omega}$ is a feasible solution to the dual linear program:

$$\text{Minimize: } b \cdot y$$

$$\text{Subject to: } A^{\intercal} y \succeq c$$

Using this observation and weak duality, we can obtain that the intersection point $x' = A^{-1}b$ is an upper bound for the optimal solution x^* of the primal LP.

Observation 4.4 lends itself to an intuitive algorithm. Say we are given a set of n constraints T that contains c in its conic combination and thus defines an upper bound to the optimal solution x^*. We aim to iteratively swap out constraints in T while keeping $c \in \text{Cone}(T)$, in the process lowering our upper bound on the primal solution until we obtain x^* (by strong duality). The algorithm is laid out formally in Algorithm 7.

Algorithm 7 CONICFOCUSING

```
Result: A vertex solution to the given LP.
Input: Objective vector $c \in \mathbb{R}^n$ and constraint matrix $A \in \mathbb{R}^{m \times n}$.
Require: An initial set of $n$ constraints $T_0$ such that $c \in \text{Cone}(T_0)$.
  Let $x_0$ solve $T_0 x = b_{T_0}$, where $b_{T_0}$ is the vector of $b$ values corresponding to the constraints in $T_0$
  while $x_t$ is infeasible do
      Select a constraint $A_i$ such that $A_i^\intercal x_t > b_i$
      Find constraint $A_j$ such that $c \in \text{Cone}(T_t \setminus \{A_j\} \cup \{A_i\})$
      $T_{t+1} \leftarrow T_t \setminus \{A_j\} \cup \{A_i\}$
      Let $x_{t+1}$ solve $T_{t+1} x = b_{T_{t+1}}$
  end while
  Return the unique $x$ that sets all the constraints in the final iteration of $T$ tight.
```

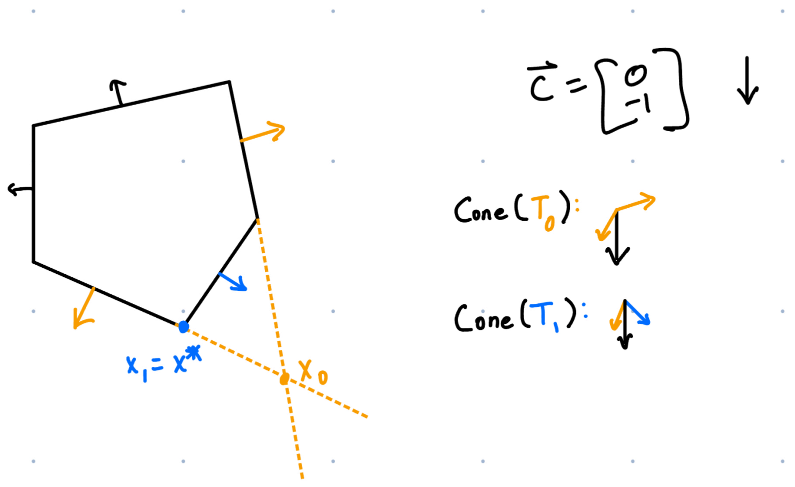

Figure 6: An example run through of CONICFOCUSING when given the orange vectors as the original cone. The objective vector (pictured in black, pointing down) is a conic combination of the two vectors in T_0, but x_0 is not feasible. We swap the top right orange vector with the vector for a violated constraint (pictured in blue) so that c is still in $\text{Cone}(T_1)$. The resulting point x_1 is the optimal vertex of our polytope.

4.5.2 Simplex Over the Dual LP

We have had trouble analyzing Algorithm 7 to determine termination, and we suspect that in many cases the algorithm may not terminate. However, because every single x_i is an upper bound for x^*

by the same reasoning as the proof of Observation 4.4, we conjecture that Algorithm 7 is simply running the simplex algorithm over the dual polytope.

Conjecture 4.5. *Algorithm 7 is equivalent to running the simplex method over the the dual polytope.*

A direct result of Conjecture 4.5 is that Algorithm 7 does not run in polynomial time. Indeed, the simplex algorithm is known to have a worst-case exponential time complexity [26].

5 Multi Agent Combinatorial Contracts

5.1 Setting

In this setting, there is a fixed set of agents $[n] = \{1, \ldots, n\}$, a reward function f, and a contract $t \in [0,1]^n$. After a contract is set, some set of agents $S \subseteq [n]$ chooses to take action. Consider the expected utility for each agent i:

$$\mu_a^i(S,t) = \begin{cases} f(S) \cdot t_i - c_i & \text{if } i \in S \\ f(S) \cdot t_i & \text{otherwise} \end{cases} \tag{13}$$

With this contract and set of active agents, the principal's expected utility is:

$$\mu_p(S,t) = \left(1 - \sum_{i \in [n]} t_i\right) \cdot f(S) \tag{14}$$

When the agents seek to maximize their utilities individually, it can be shown that they play a ordinal potential game [3, 10, 15]. It is known that in this setting, in order to incentivize some set of agents $S \subseteq [n]$ to act, the principal needs to set the contract:

$$\begin{aligned} t_i &= \frac{c_i}{f(i \mid S)} && \text{for all } i \in S \\ t_i &= 0 && \text{for all } i \notin S \end{aligned}$$

Where $f(i|S)$ denotes the marginal value of i to the set S. Note that if the marginal of some $i \in S$ is 0, then $t_i = \frac{c_i}{f(i|S)}$ is ∞, meaning that the principal cannot incentivize agent i to act.

Under this payment scheme, the principal's objective is to maximize the following set function:

$$g(S) = \left(1 - \sum_{i \in S} \frac{c_i}{f(i \mid S)}\right) f(S). \tag{15}$$

Graph Theory Notation We will soon restrict our setting to graphs, so we will define some useful graph theory notation for this thesis. First, we note undirected graphs as $G = (V, E)$ where V is the set of nodes and E is the set of edges. The degree of a node $u \in V$ to some subset of nodes $S \subseteq V$ is denoted $\deg_S(u)$ and is precisely the number of neighbors u has in S.

In the following section, we focus on the specific case when $f = \frac{|E(S)|}{E_{max}}$ is the graph supermodular function and cost of every node is a uniform constant $c \in [0, 1)$. This problem setting is called the *Uniform Cost Graph Supermodular Contracts Problem* (or U-GSC). Then, we note that the marginal value of i to a set of nodes S is simply the normalized degree of i to the nodes in set S. That is, $f(i|S) = \frac{\deg_S i}{E_{max}}$ when f is the graph supermodular function.

Then, the principal's utility to incentivize a set of nodes S is exactly (substituting into Equation 15):

$$\begin{aligned} g(S) &= \left(1 - \sum_{i \in S} \frac{c_i}{f(i \mid S)}\right) f(S) \\ &= \left(1 - \sum_{i \in S} \frac{c * E_{max}}{\deg_S(i)}\right) \left(\frac{|E(S)|}{E_{max}}\right) \end{aligned}$$

For ease of notation, we will combine the $c * E_{max}$ term and obtain a new $c \in [0, \frac{1}{E_{max}})$, so our final utility function for incentivizing set S is:

$$g(S) = \left(1 - \sum_{i \in S} \frac{c}{\deg_S(i)}\right) \left(\frac{|E(S)|}{E_{max}}\right) \tag{16}$$

5.2 Characteristics of the Optimal Set S^*

In this section, we seek to illustrate the connections between U-GSC and graph density. We will start with the utility function in the uniform cost graph supermodular setting:

$$g(S) = \left(1 - \sum_{i \in S} \frac{c}{\deg_S(i)}\right) \left(\frac{|E(S)|}{E_{max}}\right)$$

We split the utility function into two terms: $L(S) = \left(1 - \sum_{i \in S} \frac{c}{\deg_S(i)}\right)$ (also known as the degree term) and $R(S) = \left(\frac{|E(S)|}{E_{max}}\right)$ (also known as the edge density term). Then, clearly $g(S) = L(S)R(S)$.

We will focus on each term individually in order to obtain characteristics of the optimal set of nodes S^*. Observe that in order to maximize g, we would ideally like to maximize both L, the degree term, and R, the edge density term.

Consider $R(S) = \frac{|E(S)|}{E_{max}}$. Clearly, in order to maximize R, we must choose a set S with many induced edges. In order to maximize $L(S) = \left(1 - \sum_{i \in S} \frac{c}{\deg_S(i)}\right)$, we first observe that, by convexity, it is better for the degrees of every node in the induced subgraph to be equal. Further, we want the degree of every node to be large in the induced subgraph, so that each individual term in the summation $\sum_{i \in S} \frac{c}{\deg_S(i)}$ is not too large and thus $1 - \sum_{i \in S} \frac{c}{\deg_S(i)}$ is not small.

Thus, from analyzing each term individually, we have obtained 3 important characteristics of S^*:

1. S^* must contain many edges to maximize $R(S^*)$.
2. The degree of every node in S^* must be large in order to maximize $L(S^*)$.
3. The degree of every node in S^* must be equal in order to maximize $L(S^*)$.

Given the following characteristics, it seems that S^* would ideally be a *clique*, or complete graph. Recall that a clique is a subgraph of k nodes in which every pair of nodes is connected by an edge. Cliques maximize edge density and the degrees of nodes while guaranteeing that each of the nodes has equal degree. Thus, we see that our objective essentially is looking for cliques, which motivates the reduction in our next section.

5.3 Overview of Proof of NP-Hardness

In the previous section, we saw that our objective function is maximized when we select subsets of nodes $S \subseteq V$ that resemble cliques. Thus, we have our motivation to define a reduction from k-clique to U-GSC.

Definition 5.1. *(k-Clique Decision) k-Clique Decision is given as input a graph $G = (V, E)$ and a positive integer k. The task is to determine if G contains a subgraph of size k in which there is an edge between every pair of nodes.*

Using the above definition of k-Clique, we can outline the reduction from k-clique to U-GSC. Note that because k-clique is NP-Hard, this reduction shows that U-GSC is also NP-Hard.

Theorem 5.2. *There is a reduction from k-clique to U-GSC. This reduction shows that U-GSCis NP-Hard, and rules out the possibility of a constant approximation and an additive-FPTAS.*

Consider an arbitrary instance of k-clique over some graph $G = (V, E)$. We want to use a black-box solver for U-GSC to solve our version of k-clique. First, observe that if we set $c = 1$ for each node in the graph, then even k-cliques obtain negative utility.

To see this, assume K is a k-clique (for any arbitrary k). Then every one of the k nodes in K has degree $k - 1$. Consider the the value of $L(K)$ when $c = 1$:

$$L(K) = \left(1 - \sum_{i \in K} \frac{1}{\deg_K(i)}\right) = \left(1 - k\left(\frac{1}{k-1}\right)\right) = -\frac{1}{k-1} < 0$$

Since $L(K)$ is negative when $c = 1$ and K is a clique, so is its utility $g(K) = L(K)R(K) < 0$ (because $R(K)$ is always bounded between 0 and 1). Informally, since even the optimal non-empty set of nodes (i.e. a *clique*) obtains negative utility, we know that when $c = 1$, every set achieves negative utility.

The basic idea of our reduction is to set c sufficiently close to 1 such that *only* a k-clique achieves positive utility. Indeed, we find that by setting $c = \frac{1}{E_{max}}\left(\frac{k-2}{k-1}\right)$, only a k-clique achieves positive utility.[12]

Consider $g(K)$ when K is a k-clique and c is set as defined above (note that $E_{max} = O(n^2)$:

[12]For brevity, we omit the proofs showing that any set that is not a k-clique achieves negative utility. The reader is directed to [10] for the full proof.

$$\begin{aligned}
g(K) &= \left(1 - \sum_{i\in K} \frac{c}{\deg_K(i)}\right)\left(\frac{|E(K)|}{E_{max}}\right)\\
&= \left(1 - \left(\frac{k-2}{k-1}\right)\left(\sum_{i\in K}\frac{1}{k-1}\right)\right)\left(\frac{k(k-1)}{2E_{max}}\right)\\
&= \left(1 - \left(\frac{k-2}{k-1}\right)\left(\frac{k}{k-1}\right)\right)\left(\frac{k(k-1)}{2E_{max}}\right)\\
&= \frac{k}{2E_{max}(k-1)}\\
&= O\left(\frac{1}{n^2}\right)
\end{aligned}$$

Now, we know that the utility of a k-clique is $g(K) = O\left(\frac{1}{n^2}\right) > 0$. So, if we used our black-box solver for U-GSC on G with $c = \frac{1}{E_{max}}\left(\frac{k-2}{k-1}\right)$ and our solver returned a positive value on the graph G, G must contain a k-clique. If our black-box solver returned 0 (corresponding to the utility of the empty set), then we know that G must not contain a k-clique. Thus, solving U-GSC optimally is at least as hard as k-clique decision and U-GSC is NP-Hard.

Similarly, if we had a black-box solver that returned a constant approximation to U-GSC (that is, it returns a solution with utility at least $\frac{1}{p} * g(S^*)$, for some $p > 1$), we could again run our black-box approximation for U-GSC on G with the same cost. Since the only sets of nodes that attain positive value are k-cliques, we know that if our approximation outputs a positive value, G must contain a k-clique. Otherwise, our approximation returns 0 and G does not contain a k-clique. Thus, obtaining a constant approximation to U-GSC is at least as hard as k-clique decision and is thus NP-Hard.

Finally, suppose that we had a black-box additive-FPTAS to U-GSC. In this case, our black box solver finds solutions that are within $O\left(\frac{1}{n^3}\right)$ of the optimal value in polynomial time. Since $g(K) = O\left(\frac{1}{n^2}\right)$, if the graph G contains a k-clique, our black-box must output some positive value (since $O\left(\frac{1}{n^2}\right) - O\left(\frac{1}{n^3}\right) > 0$ for $n >> 0$). So, if our black-box additive-FPTAS returns a positive value, we know that G contains a k-clique. Otherwise our black-box returns 0 and G does not contain a k-clique. Thus, obtaining an additive-FPTAS to U-GSC is at least as hard as k-clique decision and is thus NP-Hard.

References

[1] I. Adler and S. Cosares. "A Strongly Polynomial Algorithm for a Special Class of Linear Programs". In: *Oper. Res.* 39.6 (Dec. 1991), pp. 955–960. ISSN: 0030-364X. DOI: 10.1287/opre.39.6.955. URL: https://doi.org/10.1287/opre.39.6.955.

[2] Sanjeev Arora, David Karger, and Marek Karpinski. "Polynomial time approximation schemes for dense instances of NP-hard problems". In: *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. 1995, pp. 284–293.

[3] Moshe Babaioff, Michal Feldman, and Noam Nisan. "Combinatorial agency". In: *Proceedings of the 7th ACM Conference on Electronic Commerce*. 2006, pp. 18–28.

[4] Siddharth Barman. "Approximating Nash Equilibria and Dense Subgraphs via an Approximate Version of Carathéodory's Theorem". In: *SIAM Journal on Computing* 47.3 (2018), pp. 960–981. DOI: 10.1137/15M1050574. eprint: https://doi.org/10.1137/15M1050574. URL: https://doi.org/10.1137/15M1050574.

[5] Hamsa Bastani et al. "Analysis of medicare pay-for-performance contracts". In: *Available at SSRN 2839143* (2016).

[6] Mark Braverman et al. "ETH hardness for densest-k-subgraph with perfect completeness". In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2017, pp. 1326–1341.

[7] Matteo Castiglioni, Alberto Marchesi, and Nicola Gatti. *Multi-Agent Contract Design: How to Commission Multiple Agents with Individual Outcome*. 2023. arXiv: 2301.13654 [cs.GT].

[8] George B Dantzig. "Origins of the simplex method". In: *A history of scientific computing*. 1990, pp. 141–151.

[9] Constantinos Daskalakis and Christos H Papadimitriou. "On oblivious PTAS's for Nash equilibrium". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 2009, pp. 75–84.

[10] Ramiro Deo-Campo Vuong et al. “On supermodular contracts and dense subgraphs”. In: *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2024, pp. 109–132.

[11] Paul Dutting, Michal Feldman, and Yoav Gal Tzur. “Combinatorial contracts beyond gross substitutes”. In: *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2024, pp. 92–108.

[12] Paul Dütting, Michal Feldman, and Yoav Gal Tzur. *Combinatorial Contracts Beyond Gross Substitutes*. 2023. arXiv: 2309.10766 [cs.GT].

[13] Paul Dütting, Tim Roughgarden, and Inbal Talgam-Cohen. “Simple versus optimal contracts”. In: *Proceedings of the 2019 ACM Conference on Economics and Computation*. 2019, pp. 369–387.

[14] Paul Dütting et al. “Combinatorial contracts”. In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2022, pp. 815–826.

[15] Paul Dütting et al. “Multi-Agent Contracts”. In: *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*. STOC 2023. Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 1311–1324. ISBN: 9781450399135. DOI: 10.1145/3564246.3585193. URL: https://doi.org/10.1145/3564246.3585193.

[16] Yuval Emek and Michal Feldman. “Computing optimal contracts in combinatorial agencies”. In: *Theoretical Computer Science* 452 (2012), pp. 56–74.

[17] Michal Feldman et al. “Hidden-action in network routing”. In: *IEEE Journal on selected areas in Communications* 25.6 (2007), pp. 1161–1172.

[18] Evangelia Gergatsouli and Xiating Ouyang. *Lecture notes in Submodular Functions and their Applications*. Dec. 2019.

[19] Sanford J Grossman and Oliver D Hart. “An analysis of the principal-agent problem”. In: *Foundations of Insurance Economics: Readings in Economics and Finance*. Springer, 1992, pp. 302–340.

[20] Abdulrahman H. Mustafa, Mohamad Sayegh, and Saim Rasheed. “Application of Linear Programming for Optimal Investments in Software Company”. In: *Open Journal of Applied Sciences* 11 (Jan. 2021), pp. 1092–1101. DOI: 10.4236/ojapps.2021.1110081.

[21] Minbiao Han et al. *A Data-Centric Online Market for Machine Learning: From Discovery to Pricing*. 2023. arXiv: 2310.17843 [cs.LG].

[22] Chien-Ju Ho, Aleksandrs Slivkins, and Jennifer Wortman Vaughan. “Adaptive contract design for crowdsourcing markets: Bandit algorithms for repeated principal-agent problems”. In: *Proceedings of the fifteenth ACM conference on Economics and computation*. 2014, pp. 359–376.

[23] Bengt Holmström. “Moral hazard and observability”. In: *The Bell journal of economics* (1979), pp. 74–91.

[24] Satoru Iwata, Lisa Fleischer, and Satoru Fujishige. “A Combinatorial Strongly Polynomial Algorithm for Minimizing Submodular Functions”. In: *J. ACM* 48.4 (July 2001), pp. 761–777. ISSN: 0004-5411. DOI: 10.1145/502090.502096. URL: https://doi.org/10.1145/502090.502096.

[25] Narendra Karmarkar. “A new polynomial-time algorithm for linear programming”. In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. 1984, pp. 302–311.

[26] Victor Klee and George J Minty. “How good is the simplex algorithm”. In: *Inequalities* 3.3 (1972), pp. 159–175.

[27] Masakazu Kojima, Shinji Mizuno, and Akiko Yoshise. *A primal-dual interior point algorithm for linear programming*. Springer, 1989.

[28] Royal Swedish Academy of Sciences. “Scientific background on the 2016 Nobel prize in economic sciences”. In: *Royal Swedish Academy of Sciences* (2016).

[29] Daniel A Spielman and Shang-Hua Teng. “Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time”. In: *Journal of the ACM (JACM)* 51.3 (2004), pp. 385–463.

[30] Éva Tardos. “A Strongly Polynomial Algorithm to Solve Combinatorial Linear Programs”. In: *Operations Research* 34.2 (1986), pp. 250–256. ISSN: 0030364X, 15265463. URL: `http://www.jstor.org/stable/170819` (visited on 04/24/2024).