Electronic Colloquium on Computational Complexity, Report No. 67 (2012)

On the Complexity of Trial and Error

Xiaohui Bei* Ning Chen† Shengyu Zhang‡

Great scientific discoveries have been made by men seeking to verify quite erroneous theories about the nature of things.

– Aldous Huxley

Abstract

Motivated by certain applications from physics, biochemistry, economics, and computer science in which the objects under investigation are unknown or not directly accessible because of various limitations, we propose a trial-and-error model to examine search problems with *unknown* inputs. Given a search problem with a hidden input, we are asked to find a valid solution. The way to find such a solution is to propose candidate solutions, i.e., *trials*, and, using observed violations, i.e., *errors*, to prepare future proposals. In accordance with our motivating applications, we consider the fairly broad class of constraint satisfaction problems, and assume that errors are signaled by a verification oracle in the format of the index of a violated constraint (with the exact content of the constraint still hidden). The objective is to design time- and/or trial-efficient algorithms that will find a valid solution or alternatively, to show that the problem is intrinsically hard.

Our discoveries can be summarized as follows. On one hand, despite the seemingly very little information provided by the verification oracle, efficient algorithms do exist for a number of important problems. For the Nash, Core, Stable Matching, and SAT problems, the unknown-input versions are as hard as the corresponding known-input versions, up to a factor of polynomial. We further conduct a closer study of the latter two problems and give almost tight bounds on their trial complexities. The techniques employed to prove these results vary considerably, including, e.g., order theory and the ellipsoid method with a strong separation oracle.

On the other hand, there are problems whose complexities are substantially increased in the unknown-input model. For Graph Isomorphism and Group Isomorphism, in particular, although there are trial-efficient algorithms, no time-efficient algorithms exist (unless **PH** collapses and $\mathbf{P} = \mathbf{NP}$, respectively). These results also imply lower bounds on the tradeoff between time and trial complexities. The proofs use quite nonstandard reductions, in which an efficient simulator is carefully designed to simulate a desirable but computationally unaffordable oracle.

Our model investigates the value of information, and our results demonstrate that the lack of input information can introduce various levels of extra difficulty. The model accommodates a wide range of combinatorial and algebraic structures, and exhibits intimate connections with (and we hope can also serve as a useful supplement to) certain existing learning and complexity theories.

*Tsinghua University, China. Email: bxh08@mails.tsinghua.edu.cn.

†Nanyang Technological University, Singapore. Email: ningc@ntu.edu.sg.

‡The Chinese University of Hong Kong, Hong Kong. Email: syzhang@cse.cuhk.edu.hk.

ISSN 1433-8092

1 Introduction

In a broad sense, computer science studies computation and other information processing tasks. Theoretical computer science, in particular, focuses on understanding the ultimate power and limits of computation in various models. A central question in theoretical computer science is to find the minimum cost of an algorithm for computing a function f on inputs x. Algorithm design and computational complexity analysis assume that the input is given explicitly. However, in many scenarios, we actually *lack input information*.

- In a normal-form game, it is usually assumed that the payoff function of every player is given explicitly (or can be computed easily in a certain way). However, in many circumstances, players do not necessarily know their payoffs or possible strategies (particularly when they are exploring a new environment). Further, they may not even know the number of other players, let alone their strategies [34]. For instance, when a new and promising business model emerges, each startup company may be unaware of all of its competitors, and may lack a full understanding of the commercial environment and customer behavior in this new model, and hence its payoffs.

- In a two-sided matching marketplace, every individual has a preference over the agents of the other side, which is taken as the input of a matching algorithm. However, the individuals themselves may not know their (precise) preferences. For example, a man usually comes to realize how much he likes a woman only after they have hung out for a while. Another example is the job market: An applicant may not know precisely how much he or she would like the job in question because of a lack of information about the nature of the job, the company culture, and his or her future relations with colleagues. At the same time, it is generally quite difficult for recruiters to judge which candidates best fit the position. Indeed, decision makers quite often make hiring mistakes: "... a systematic and continuous approach to fitting the right person to the right job at the right time has long been the Holy Grail of workforce organization" [2].

- In the event of an infectious disease outbreak, due to an unknown virus, biochemists need to find diagnostic reagents that have no serious side effects. In a simplified formulation, this involves a search for a reagent that satisfies a collection of constraints (e.g., one constraint may be that the reagent should not contain certain medical ingredients composed in a certain way, as otherwise, its reaction with the virus could cause a severe headache). If the biochemists knew everything about the virus (e.g., its DNA sequence, chemical composition, etc.), then it would be much easier for them to find a diagnostic reagent. If the virus is largely unknown, however, then they are left with an effectively unknown-constraints formula to satisfy. Of course, they could try to employ modern DNA technologies to gain most information on the virus, but doing so usually takes a long time, and identifying a diagnostic reagent to control the ensuring pandemic is a matter of urgency.

In summary, an input, while it exists, may be *unknown* because of our limited knowledge, and control of the system or our lack of experience in a new environment. There are numerous other scenarios with unknown inputs, e.g., animal behavior studies, neural science, and hidden web databases, to name just a few.

1.1 Trial and Error

Trial and error is a basic methodology in problem solving and knowledge acquisition, and it has also been used extensively in product design and experiments [54]. Generally speaking, the approach proceeds by adaptively posing a sequence of candidate solutions and observing their validity. If a proposed candidate solution is found to be valid, then the mission is accomplished. Otherwise, an error is signaled from one of the characteristics of the studied object. An important feature of the approach is its solution orientation:

the goal is to find one solution, with little care paid to other considerations such as why the solution works [1].

Trial and error is also a commonly used approach in the aforementioned examples. In economics, individuals adopt and adaptively adjust their strategies based on observed market reactions. Such self-motivated, but self-regulating, types of behavior, as implied by Adam Smith's "invisible hand" theory, can converge to a socially desirable state (even without the individuals involved having any knowledge of one another). In a company, an employee will usually look for a more suitable position when dissatisfied with his or her current one, and senior management will usually encourage personnel adjustments to enhance performance. Biomedical scientists conduct clinical trials to test their designed reagents, and if an unacceptable side effect is observed, then they collect and analyze feedback data to help with future diagnostic reagent tests [37].

The most critical ingredient in the trial and error approach is how to employ previously returned errors to propose future trials. This procedure is algorithmic in nature, but it does not seem to have been formally addressed from an algorithmic perspective. This paper aims to investigate this approach on a broad category of problems with unknown inputs.

1.2 Model and Preliminaries

Motivated by the foregoing examples, we investigate the effects of the lack of input information from a computational viewpoint on the basis of the trial and error approach. The central question we consider is the following.

How much extra difficulty is introduced due to the lack of input knowledge?

In this paper, we explore this question in search problems. Suppose that on an input (instance) I, there is a set $S(I)$ of *solutions*. A search problem is to find a solution $s \in S(I)$ to input I.[1] Numerous problems arising from a variety of applications studied in algorithm design and computational complexity are search problems. Typical examples include searching for a Nash equilibrium in a multi-player game, searching for a satisfying assignment in a conjunctive normal form (CNF) formula, and finding a stable matching to pair individuals with preferences in a two-sided market.

All of these problems, in addition to the motivating examples discussed earlier, naturally fall into the broad category of *constraint satisfaction problems* (CSPs). Suppose that there is a space $\Omega = \{0,1\}^n$ of candidate solutions. Corresponding to an input I, there are a number of constraints $C_1, C_2, \ldots, C_m(,\ldots)$, where each $C_i \subseteq \{0,1\}^n$ is a relation on the solution variables defined on given domains.[2] The solutions of I are defined as those s that satisfy all constraints C_i, i.e., $s \in \bigcap_i C_i$. Note that the number of constraints can range from constant to polynomial, exponential, or even infinite. CSPs are a subject of intensive research in theoretical computer science, artificial intelligence, and operations research, and they provide a common basis for exploration of a large number of problems with both theoretical and practical importance.

This paper addresses the situation in which the input I is unknown. For a search problem A, we denote by $\mathsf{A_u}$ the same search problem with unknown inputs. For example, in the StableMatching problem, the

[1] One might wish to find more or even all solutions. Here, we follow the standard requirement for searching problems in complexity theory [31, 7] by asking for only one (arbitrary) solution.

[2] Note that it is possible for a problem to have different CSP definitions, which, depending on the available set of tools, may in turn lead to different complexities for solving the problem. For example, to identify an unknown substance, we can employ different (physical, chemical, etc.) test methods, which, in general, require different costs. This phenomenon reflects the intrinsic variety of a problem and the diversity of its solutions. Thus, to define an unknown-input problem, we need to indicate explicitly its corresponding CSPs. For the problems investigated in this paper, we arguably use their most natural definitions.

input contains the preference lists of all men and women; in $\mathsf{StableMatching_u}$, these preference lists are unknown to us. The constraints are that all man-woman pairs (m, w) are not blocking pairs, and the task is to find a solution that satisfies all constraints, namely a stable matching [30].

Similar to the way in which a biochemist proposes a chemical reagent and then performs clinical tests, here our method of searching for a solution of a CSP is also the trial and error approach. We propose a candidate solution s: If s is not a valid solution, then we are told so by a *verification oracle* V, and, what is more, V also gives us the *index* of one constraint that is not satisfied. Otherwise, s satisfies all constraints, and then we cannot observe any violated constraint; equivalently, V returns a confirmative answer, and our job is done. Some remarks are necessary:

- If more than one constraint is violated, then (the index of) any one of them can be returned by V. We make no assumption about which one, not only because worst-case analysis is standard in algorithm and complexity studies, but also because in many applications, such as drug tests, the verification oracle is carried out by Nature or human bodies, and thus how and which violation is returned is truly beyond our current understanding.

- Note that V does not reveal the constraint itself, but only its index or label. For example, we know something like "the third constraint is violated" in the proposed assignment of the $\mathsf{SAT_u}$ problem, or "the second player has a better mixed strategy" for the proposed strategy in the $\mathsf{Nash_u}$ problem, but the exact content of the constraint (i.e., the literals in the clause of $\mathsf{SAT_u}$ or the player's utility function of $\mathsf{Nash_u}$) is still unknown to us, which is consistent with our motivating examples. If a headache is observed in a drug development clinical trial, then we do not always know which components of the proposed reagent caused the problem: We have only a label of "headache" for the proposed reagent.

Surprisingly, despite this seemingly very little information and the worst-case assumption on the verification oracle, we still have efficient algorithms for many problems.

Given the verification oracle V, an algorithm is an interactive process with V. We choose candidate solutions (i.e., trials), and the oracle returns violations (i.e., errors). The process is adaptive, i.e., the newly proposed solution can be based on the historical information returned by the oracle.

Because our focus is on how much *extra* difficulty is introduced by the lack of input information for a search problem A, we single out this complexity by comparing the unknown-input and known-input scenarios. To this end, we equip our algorithms with another oracle, the *computation oracle*, which can solve the known-input version of the same problem A. Overall, our algorithms can access two oracles, the verification oracle and the computation oracle (we do not allow them to invoke each other).

As is standard in complexity theory, a query to either oracle has a unit time cost. The *time complexity* of a problem with unknown inputs is the minimum time needed for an algorithm to solve it for all inputs and all verification oracles consistent with the input. We employ the standard notation in computational complexity theory for complexity classes such as $\mathbf{P}$ and $\mathbf{NP}$ and also for oracles. For example, $\mathsf{A_u} \in \mathbf{P}^{\mathsf{V},\mathsf{A}}$ means that problem $\mathsf{A_u}$ can be solved by a polynomial-time algorithm with verification oracle V and the computation oracle that can solve the known-input version of A. If this occurs, then we consider the extra complexity (resulting from the unknown input) not to be very high. The central question can therefore be translated to the following. Given a search problem A, is $\mathsf{A_u} \in \mathbf{P}^{\mathsf{V},\mathsf{A}}$? If the given known-input problem A is in $\mathbf{P}$, then the computation oracle can be omitted, and the problem becomes "Is $\mathsf{A_u} \in \mathbf{P}^{\mathsf{V}}$?"

We also define the *trial complexity* of an unknown-input problem $\mathsf{A_u}$ as the minimum number of queries to the verification oracle that any algorithm needs to make, regardless of its computational power.[3] As is

[3]It is thus the "query complexity" to the verification oracle. Here, we adopt the term "trial complexity" to avoid any potential confusion of the two types of oracle queries (corresponding to the two oracles).

standard in query complexity theory, we can consider deterministic or (Las Vegas) randomized algorithms. The latter can be assumed to be error-free because of the verification oracle V, and we count the cost as the expected number of queries to V. We denote by $\mathsf{D}(A_{\mathsf{u}})$ and $\mathsf{R}(A_{\mathsf{u}})$ the deterministic and randomized trial complexities of A_{u}, respectively.

We investigate trial complexity not only because it provides a rigorous proof of computational hardness, but also because it measures the number of trials (or in another perspective, errors) that must be undertaken to find a solution. Note that in many scenarios, such as diagnostic reagent development, trials constitute the major expense, both financially and temporally, and in almost all of the motivating examples discussed earlier, an important goal is to design protocols or experiments with a small number of trials.

1.3 Our Results and Techniques

We consider a number of problems, that are motivated by the aforementioned examples, to investigate the trial and time complexities resulting from the lack of input knowledge. (The formal definitions of these problems and their natural formulation as CSPs are deferred to subsequent sections.)

Theorem 1. *For the following problems* A*, we have* $\mathsf{A}_{\mathsf{u}} \in \mathbf{P}^{\mathsf{V},\mathsf{A}}$*.*

- Nash*: Find a Nash equilibrium of a normal-form game.*
- Core*: Find a core of a cooperative game.*
- $\mathsf{StableMatching}$*: Find a stable matching of a two-sided market with preferences.*
- SAT*: Find a satisfying assignment of a CNF formula.*

Nash is a fundamental problem in game theory, and its complexity has recently been characterized (as **PPAD**-complete) [24, 21]. Core is also a fundamental problem in cooperative game theory [57]. Both problems are naturally defined as CSPs. Our algorithms for both $\mathsf{Nash}_{\mathsf{u}}$ and $\mathsf{Core}_{\mathsf{u}}$ employ the ellipsoid method, although for $\mathsf{Nash}_{\mathsf{u}}$ we shrink the input space, and for $\mathsf{Core}_{\mathsf{u}}$ we shrink the solution space. One technical difficulty is that the target space may degenerate to the case of containing at most one point. (In $\mathsf{Nash}_{\mathsf{u}}$, there is only one input point, and in $\mathsf{Core}_{\mathsf{u}}$ the core may contain only one point or even be empty.) Note that the standard perturbation approach, which proceeds by increasing the volume of the feasible region, is not applicable in our setting, because the linear constraints, as the input, are unknown. Here, we employ a more sophisticated ellipsoid method that works as long as the polyhedron can be specified by a *strong separation oracle*. As it turns out, this oracle can be constructed from the verification oracle V in both problems, and, crucially, the construction for $\mathsf{Nash}_{\mathsf{u}}$ uses the existence of a Nash equilibrium in *any* game.

$\mathsf{StableMatching}$ is a problem with interesting combinatorial structures and many applications, such as the pairing of graduating medical students with hospital residencies [64, 63]. SAT is a natural CSP, with the constraints being the OR of some literals. Considering the practical significance of $\mathsf{StableMatching}$ and SAT, we take a closer look at their trial complexities.

Theorem 2. *We have the following bounds for the trial complexity.*

- $\Omega(n^2) \le \mathsf{R}(\mathsf{StableMatching}_{\mathsf{u}}) \le \mathsf{D}(\mathsf{StableMatching}_{\mathsf{u}}) \le O(n^2 \log n)$*, where* n *is the number of agents.*
- *Given a formula with* n *variables and* m *clauses,* $\mathsf{R}(\mathsf{SAT}_{\mathsf{u}}) \le \mathsf{D}(\mathsf{SAT}_{\mathsf{u}}) = O(mn)$*. Further,* $\mathsf{R}(\mathsf{SAT}_{\mathsf{u}}) = \Omega(mn)$ *if* $m = \Omega(n^2)$*, and* $\mathsf{R}(\mathsf{SAT}_{\mathsf{u}}) = \Omega(m^{3/2})$ *if* $m = o(n^2)$*.*

The proofs of both lower bounds deviate from the standard method of applying Yao's min-max principle. Rather, they are obtained by arguing that, for an arbitrary but fixed *randomized* algorithm with an insufficient number of queries, there are input instances with disjoint solution sets between which the algorithm cannot distinguish. The existence of such input instances are proved by the probabilistic method for $\mathsf{StableMatching}_\mathsf{u}$, and by an adaptive construction procedure for SAT_u.

The proofs of the upper and lower bounds for $\mathsf{R}(\mathsf{StableMatching}_\mathsf{u})$ also employ order theory [17, 25]. A key step is to characterize how fast one can shrink the set of linear orders consistent with a partial order by worst-case pair violations. We identify the average height as the correct measure; the control of which allows us to bound the speed of the shrinkage. Along the way, we examine another natural problem, $\mathsf{Sorting}_\mathsf{u}$, and as a by-product, we completely pin down its trial complexity as $\Theta(n \log n)$.

It is somewhat surprising that knowing only the indices of violated constraints is already sufficient to admit quite a number of efficient algorithms. It is therefore natural to wonder whether the lack of input information adds any extra difficulty at all in finding a solution. We find that it does indeed: there are problems whose unknown-input versions are considerably more difficult than their known versions. Two representatives are GraphIso and GroupIso, the problems of deciding whether two given graphs or groups are isomorphic.

Theorem 3. *We have the following hardness results.*

- *If* $\mathsf{GraphIso}_\mathsf{u} \in \mathbf{P}^{\mathsf{V},\mathsf{GraphIso}}$*, then the polynomial hierarchy (***PH***) collapses to the second level.*
- *If* $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_\mathsf{u} \in \mathbf{P}^{\mathsf{V}}$*, then* $\mathbf{P} = \mathbf{NP}$*. Here,* $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)$ *is the group isomorphism problem with the second group known as* $\mathbb{Z}_p$ *for a prime* p*.*

However, if SAT *is given as the computation oracle, then we have deterministic polynomial-time algorithms for* GraphIso *and* GroupIso*, i.e.,* $\mathsf{GraphIso}_\mathsf{u} \in \mathbf{P}^{\mathsf{V},\mathsf{SAT}}$ *and* $\mathsf{GroupIso}_\mathsf{u} \in \mathbf{P}^{\mathsf{V},\mathsf{SAT}}$*, with* $O(n^6)$ *and* $O(n^2)$ *trials, respectively.*

Note that $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)$ (with a known input) admits a simple polynomial-time algorithm by comparing the multiplication tables. Actually, GroupIso is in $\mathbf{P}$ if the two groups are Abelian [42]. However, if the multiplication table of the input group is unknown, then, surprisingly, the problem becomes $\mathbf{NP}$-hard. Interestingly, this substantial increase in computational difficulty occurs only for time complexity, not for trial complexity, which can be seen as a tradeoff (from below) between the two complexity measures—a phenomenon not commonly seen in other query models.

This hardness result for $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_\mathsf{u}$ is proved by a nonstandard reduction from the classic $\mathbf{NP}$-complete problem of finding a Hamiltonian cycle. We use an algorithm $\mathcal{A}$ for $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_\mathsf{u}$ to find a Hamiltonian cycle in a given graph G in the following way. Assuming the existence of a Hamiltonian cycle C, which does not change the $\mathbf{NP}$-completeness of the problem, we define a group H via C and run $\mathcal{A}$ on input $(H, \mathbb{Z}_p)$. An issue here is that because the reduction algorithm has only polynomial time, it cannot find such a Hamiltonian cycle for defining H. A related issue is how to provide the verification oracle V for $\mathcal{A}$ without knowing C. These issues can be overcome by (i) making use of the crucial property that $\mathcal{A}$ does not know its input, and (ii) designing an efficient simulator V' for the verification oracle V. Due to the time constraint, V' cannot perfectly mimic V to answer all of $\mathcal{A}$'s questions correctly. However, it is designed with the favorable property that whenever it produces an incorrect answer, a Hamiltonian cycle in G has just been found. ($\mathcal{A}$'s correctness on H may already have been compromised, but we are not further concerned with it. We simply use $\mathcal{A}$'s code to serve the purpose of finding a Hamiltonian cycle in G.)

Finally, beyond all of the foregoing problems that can be solved in $\mathbf{P}^{\mathsf{V},\mathsf{SAT}}$, we show via an information theoretical argument that certain other problems, such as Subset Sum, have an exponential lower bound for the randomized trial complexity.

Theorem 4. $\mathsf{R}(\mathsf{SubsetSum}_\mathsf{u}) = \Omega(2^n)$.

Our results illustrate the variety of time and trial complexities that arise from the lack of input information for different problems, and imply distinct levels of the cruciality of input information for different problems.

In addition to the specific techniques previously mentioned for each problem, a general remark is that, at a very high level, our algorithms are in line with the candidate elimination approach, similar to many existing learning algorithms [44]. However, our framework allows a space for possible inputs *and* a space for possible solutions—the interplay between them seems to be the main source of combinatorial structures, and how well the two spaces are combinatorially related accounts for the complexity of the problem. Some algorithms (e.g., that for Core_u) obtain their efficiency by directly shrinking the solution space. Even for those that shrink the input space (e.g., those for SAT_u and Nash_u), the key is to explore the relation of the two spaces and to design trials such that even a worst-case violation can be used to cut out a decent fraction of the input space.

1.4 Relation to Existing Work

Our model with unknown inputs bears a resemblance to certain other problems and models, e.g., learning, algorithm design in unknown environments, ellipsoid method, and query complexity. However, there are fundamental distinctions between these models and ours. We now provide a detailed discussion of our work's relation to these models and problems.

Learning. Our model has strong connections to various learning theories, but fundamental differences also exist.

1. Learning theories, in essence, aim to identify the unknown object *itself*, either exactly (as in concept learning) or approximately (sometimes in the form of a prediction, as in PAC learning and active learning). In our model, however, the ultimate goal is quite different: we attempt only to find a solution on an unknown object, without necessarily learning the object itself. For certain applications (such as the aforementioned development of diagnostic reagents), finding a solution is indeed the main mission.

2. It is important to note that in our model, a solution may be found long before the exact input is learnt. Further, in certain cases, such as SAT_u and Nash_u, the exact input may take an exponential number of queries or even be impossible to learn. Our algorithms, in contrast, are able to find a solution in polynomial time without learning the exact input.

3. Both similarities and differences abound in existing learning theories, and deciding which one to use in a specific application largely depends on the available method of accessing the unknown. As we have demonstrated, there are a fairly large number of scenarios in which the only available access to the unknown is provided by the verification oracle, but existing learning theories do not seem to address such situations.

In summary, with its solution-oriented objective and advantages in computational efficiency, the present work is hopefully to serve as a useful supplement to existing learning theories, particularly in contexts in

which the unknown object itself is impossible or unaffordable to learn and the only available access to the unknown is through a solution-verification process.

Next we give a detailed discussion on the relationship between our model and relevant learning theories.

- *Concept learning.* In concept learning, a concept is secretly drawn from a given concept class, and the task at hand is to identify it; see the survey by Angluin [5]. More precisely, given a domain set X and a collection C of concepts, each $c \in C$ maps $X \to \{0, 1\}$, defining a table with rows C and columns X. Further, there is an unknown concept $c^* \in C$. To identify the row/concept c^*, two types of queries are commonly used: (i) a *membership query*, where one queries a particular column/domain element $x \in X$, and an oracle returns its value $c^*(x)$, and (ii) an *equivalence query*, where one proposes a particular row/concept $c \in C$, and an oracle returns either a confirmative answer or a column/domain element x with $c(x) \neq c^*(x)$. If the proposed concept c is allowed to be any mapping from X to $\{0, 1\}$, then it is called an extended equivalence query.

  Despite having some similar features, our model cannot be cast into the membership or equivalence query frameworks. For a search problem, we can take the set of inputs as the concept class C, take all possible solutions as the domain X, and let $c(x) = 1$ if and only if $x \in X$ is a solution of the instance c. A membership query is thus similar to our model in that ours also proposes solutions $x \in X$. However, the goal in our problem is to find a solution x that satisfies $c^*(x) = 1$, where c^* is the hidden input, whereas the task in the membership query model is to identify the concept c^*. The second important difference is that other than knowing whether or not a proposed solution is valid, in our model, we also gain extra information on the index of a violated constraint. Both this extra information and the feature of not necessarily learning the input enable us to obtain (more) combinatorial structures and to design efficient algorithms.

  Alternatively, we can take the solution space as C and the set of constraints (corresponding to an input instance) as X; then, our problem is to find a solution c that satisfies $c(x) = 1$ for all $x \in X$, i.e., all constraints are satisfied. Our trial and error model then becomes similar to the equivalence query model: each time we propose a solution c and an oracle returns a violated constraint x where $c(x) = 0$. However, note that different input instances correspond to different sets of constraints, and thus induce different (C, X) tables. As a result, there are actually many such tables in our model, and we do not know which one we are faced with. Our task is still to find a row (i.e., a solution) in which all entries are 1 in the hidden table, and the algorithm needs to succeed for *any* of the possible input tables (C, X). Thus, both the unknowns and objectives of the two models are entirely distinct.

  We could also put all of the possible constraints together to form a much larger table, with the rows indexed by all possible solutions $c \in C$ and the columns indexed by all possible constraints $x \in X$, with $c(x) = 1$ if the solution c satisfies the constraint x, and $c(x) = 0$ otherwise. This approach would produce only one fixed table (as in concept learning), but the hidden input would now correspond to a hidden subset of columns. When a violation x is returned, it is not the identity of a column of the large table, but the column's relative position inside the hidden subset of columns. Further, the required task is also different. In the equivalence query model a row is to be identified, whereas in our model we are asked merely to return one row $c \in C$ that satisfies $c(x) = 1$ for all x from the hidden subset of columns.

  In addition, even more differences exist between our model and concept learning. In our model, a search problem instance may have several different solutions; thus, the correct output of an algorithm is not unique. However, in concept learning, the return of an algorithm has to be the unique hidden input c^*. Another difference lies in the complexity measure. In concept learning, the cost of an algorithm is measured in terms of $|C|$ and $|X|$. In our model, in constrast, the complexity is evaluated

in terms of the input size of the search problem, rather than the size of the solution space, which can be exponential or even unbounded.

- *Active learning.* A more general type of query learning model is active learning. Roughly speaking, there are two sets, L and U, whose data are labeled and unlabeled, respectively. Based on labeled set L, a learning algorithm interactively queries an oracle concerning certain data instances from unlabeled set U. The oracle then returns the labels of the queried data, which are added into L. See the survey by Settles [67] for a more detailed discussion. There are a number of similarities between active learning and our model. For example, both consider an iterative and adaptive process that involves posing selected queries to a given oracle, and both investigate the complexity of interacting with this oracle [23, 11]. However, the two oracles are fundamentally different. In active learning, the oracle always returns the *correct* answer (i.e., label), whereas in ours it always returns one of the *erroneous* constraints. In addition, the objective in active learning is to achieve a high degree of accuracy (for the prediction of the data in unlabeled set U) using as few labeled instances as possible, which is very different from the objective in ours.

- *PAC learning.* Another slightly related model is Valiant's probably approximately correct (PAC) learning model [69], in which from a probability distribution D over the domain set X, we can sample instances and be told whether or not they are supported by the unknown concept. Based on the training samples (the minimum number of which is called the sample complexity), the objective is to propose a concept (called a hypothesis) that approximates the unknown concept with a small probability of error (which can be used to predict future samples). Classic examples include DNF learning [38, 46, 19] and halfspace learning [45, 41, 47]. As in concept learning, PAC also aims to learn the unknown itself (although an approximation is allowed) rather than an induced solution, as in our model.

There are also many other learning models, e.g., decision tree learning, reinforcement learning [12], statistical learning [71], (semi-)supervised learning [10], and learning with errors [62]; see [44, 52] and the references therein for a more detailed discussion. The high-level philosophy of these models is also "sample and predict", which is very different from our trial and search (for a solution).

Algorithms in an unknown environment. Theorists have addressed the exploration of an unknown environment in several domains. In robot exploration and navigation, a robot is placed in an unknown geometric terrain with obstacles [14, 26, 35, 4, 36] or a graph with unknown edges [27, 58, 9, 3], and the robot's goal is to explore the entire unknown space starting from a given point. A similar problem is path planning, where again we are given an unknown environment, but the objective is to find a desired path between two specific locations in either graphs [61] or geometric spaces [50, 51, 6]. Readers are referred to [53] for a survey of these topics and results. A feature our model has in common with these is that the input object under study is unknown. However, as opposed to the proposal of entire solutions in our model, solution finding in the robot exploration and path planning models takes place via a "local search" type approach, where an agent looks for the next move on the basis of his or her current location and historical information. Despite being a very natural model for these applications, it seems difficult to generalize to scenarios without an underlying geometric structure.

Ellipsoid method. Our trial and error search model is, in spirit, similar to the ellipsoid method (see, e.g., [33]), in which a point is proposed as a trial, and a separating hyperplane is returned as an error. The ellipsoid method is an elegant approach for proving the polynomial time solvability of a class of

combinatorial optimization problems; it applies even when the explicit expressions of the constraints are unknown. However, our trial and error model includes a much broader class of search problems—not only convex optimization problems, but also many with pure combinatorial structures (e.g., the SAT, GroupIso, and GraphIso problems considered in our paper). From this perspective, the ellipsoid method is only one possible approach for the trial and error search problems in our model. (Indeed, the algorithms for the $\mathsf{Core_u}$ and $\mathsf{Nash_u}$ problems considered herein are built crucially on the ellipsoid method; but we also employ other approaches to solve other problems, including $\mathsf{SAT_u}$ and $\mathsf{StableMatching_u}$.) In addition, even if a problem can be solved using an ellipsoid-based approach, its trial and time complexities may be quite large (e.g., the ellipsoid method cannot compete with the simplex algorithm for practical calculations). Therefore, for problems with numerous applications, e.g., $\mathsf{StableMatching_u}$, a more efficient (combinatorial) algorithm is desirable (note that a stable matching instance can be written by a linear program [70]).

Complexity. In the query model (also known as decision tree model), an algorithm makes queries in the form of "$x_i =?$", and the task is to compute a function f on the unknown x by the minimum number of queries [20]. Although this area has the same flavor of computing a function without learning all input variables, it is quite different from our model in the form of queries allowed.[4] Therefore, our results on trial complexity can be viewed as an extension of the traditional query model by allowing a much larger class of queries with natural motivations.

In some cryptographic tasks, input instances are hidden from one party. For example, in instance-hiding proof systems [13], the verifier tries to compute a function f on input x by interacting with one or more provers, without leaking any information on x to the prover(s). Although this model is clearly very different from ours, an interesting research direction would be to explore the connections between our model and various cryptographic tasks.

2 Stable Matching

In a Gale-Shapley two-sided matching market model [30], we are given a set of men M and a set of women W, where $|M| = |W| = n$. Each man $m \in M$ has a strict and complete *preference list*,[5] denoted by $\succ_m$, ranking all the women in W, where $w_1 \succ_m w_2$ means that m prefers w_1 to w_2. The preference list $\succ_m$ is assumed to be transitive (i.e., if $w_1 \succ_m w_2$ and $w_2 \succ_m w_3$, then $w_1 \succ_m w_3$). The preference list $\succ_w$ of every woman $w \in W$ is defined similarly.

Given a matching between M and W, denoted by μ, we say that $m \in M$ and $w \in W$ form a *blocking pair* if both prefer each other to their matched partner in μ, i.e., $w \succ_m \mu(m)$ and $m \succ_w \mu(w)$, where $\mu(m)$ and $\mu(w)$ are the woman matched to m and the man matched to w in μ, respectively. Matching μ is called *stable* if it contains no blocking pair. The StableMatching problem is to find a matching that is stable, namely, a matching that satisfies the following set of constraints, labeled by man-woman pairs.

$$\text{Either } \mu(m) \succ_m w \text{ or } \mu(w) \succ_w m, \quad \forall m \in M \text{ and } \forall w \in W. \tag{1}$$

A stable matching always exists, and can be computed by Gale-Shapley's seminal deferred acceptance algorithm in time $O(n^2)$ [30].

[4] Other forms of queries were also considered, such as those in linear decision trees and algebraic decision trees, but they are still in a very restricted form of queries.

[5] We follow the model proposed by Gale and Shapley in their seminal work [30], where the number of men and women is the same, and every individual's preference is assumed to be complete and strict. All of these assumptions can be removed in our results, but for simplicity of exposition, we will adopt Gale and Shapley's original model.

In the unknown-input version of the stable matching problem, denoted by $\mathsf{StableMatching_u}$, we do not know the preference lists $\succ_m$ and $\succ_w$. What we can do is to propose candidate matchings as potential solutions. If a proposed matching is indeed stable, then the verification oracle V returns Yes, and the problem is solved. If it is not stable, then one constraint (i.e., a blocking pair) is revealed by V. Our first result is an $O(n^2 \log n)$ upper bound on the randomized trial complexity of $\mathsf{StableMatching_u}$.

Theorem 5. *There is a polynomial-time randomized algorithm solving* $\mathsf{StableMatching_u}$ *with* $O(n^2 \log n)$ *trails.*

Before describing the idea underlying the proof of this result, we first look at the unknown-input version of another basic problem, $\mathsf{Sorting}$, which has a close relationship with $\mathsf{StableMatching}$. In $\mathsf{Sorting_u}$, there is a set of n elements $S = \{a_1, a_2, \ldots, a_n\}$ in some underlying linear (total) order $\succ$, but this order is unknown to us, and the task is to discover it. We can propose a linear order $(a_{k_1}, a_{k_2}, \ldots, a_{k_n})$ each time. If it is indeed the desired hidden total order, i.e., $a_{k_1} \succ a_{k_2} \succ \cdots \succ a_{k_n}$, then the verification oracle returns Yes, and the problem is solved; otherwise, a pair of elements (a, b) is returned such that a is before b in the proposed order, whereas $b \succ a$ in the actual order.

It is well known that the time complexity of a comparison-based sorting problem is $\Theta(n \log n)$. Note that this time complexity for $\mathsf{Sorting}$ is completely different from our trial complexity for $\mathsf{Sorting_u}$. In the following, we will show that the trail complexity bound for $\mathsf{Sorting_u}$ is actually also $\Theta(n \log n)$.

Lemma 6. $\mathsf{Sorting_u}$ *can be solved by a polynomial-time randomized algorithm using* $O(n \log n)$ *trials. Meanwhile, any randomized algorithm that solves* $\mathsf{Sorting_u}$ *needs at least* $\Omega(n \log n)$ *trials even with unbounded computational power; that is,* $\mathsf{R}(\mathsf{Sorting_u}) = \Theta(n \log n)$.

Idea of the proof. The upper and lower bounds for $\mathsf{R}(\mathsf{Sorting_u})$ both use order theory [17, 25]. Both bounds critically depend on how fast the set of complete orders consistent with a partial order can be shrunk by *worst-case* pair violations. Due to the distinction of the oracles in the standard comparison model and in ours, the techniques in the comparison model do not straightforwardly carry over to solving our problem. It turns out that controling the measure of *average height* allows us to bound, in both directions, the worst-case shrinkage speed in our model. Although the average height is $\#\mathbf{P}$-hard to compute [18], we can efficiently estimate this value to a sufficient degree of precision by using a fully polynomial randomized approximation scheme (FPRAS) for another related problem [28]. □

Formal Proof of Lemma 6. First, we define the notation as follows. A *partially ordered set* (or *poset*) is a set S equipped with an irreflexive transitive relation $>$. A *linear extension* of a poset $(S, >)$ is a linear order $\succ$ over set S such that $a \succ b$ whenever $a > b$ in S. For any given poset $(S, >)$ and elements $a, b \in S$, we denote by $\Pr(a \succ b)$ the probability of $a \succ b$ where $\succ$ is chosen uniformly at random among linear extensions of $(S, >)$.

In the $\mathsf{Sorting_u}$ problem, suppose the unknown order is $\succ^*$. Notice that for each of our proposed orders $\succ$, the verification oracle returns a pair of elements $a, b \in S$, from which and $\succ$, we can infer the relation between a and b in the actual order $\succ^*$. Thus, at each point of the algorithm, the information collected so far forms a poset $(S, >)$, of which the underlying unknown order $\succ^*$ is a linear extension.

For any poset $(S, >)$, we call an order $(a_{k_1}, a_{k_2}, \ldots, a_{k_n})$ *good* if there is a constant $c < 1$, such that for any $i < j$, we always have $\Pr(a_{k_j} \succ a_{k_i}) < c$. Note that this is a very strong condition because it requires a constant shrinkage for *all* possible pairs (i, j). The idea of solving $\mathsf{Sorting_u}$ is that, at each step of the algorithm when our collected information forms a poset $(S, >)$, we propose a good order if it exists. The property of a good order guarantees that whatever the verification oracle returns, we can always reduce the number of candidate linear extensions by a constant fraction c. Note that at the beginning of the algorithm, the number of candidate linear extensions is $n!$ as we do not have any information about $\succ^*$.

And at the end of the algorithm the unique order $\succ^*$ is found and thus the number of linear extensions is 1. Therefore, the problem $\mathsf{Sorting_u}$ can be solved in polynomial time and by $O\big(\log_{1/c} n!\big) = O(n \log n)$ trials to the verification oracle, provided that

1. for any partially ordered set $(S, >)$, a good order always exists, and
2. we find a good order in polynomial time.

Next we shall address how to satisfy these two conditions. Given a poset $(S, >)$ and an element $a \in S$, define its *average height* $h(a)$ to be the average rank of a in all linear extensions of $(S, >)$, where the rank of a in a linear extension $\succ$ is the number of elements b such that $b \succ a$. It is easy to see that the average height of elements in S are all rational numbers between 0 and $n-1$. Kahn and Saks [40] showed that for any pair of elements $a, b \in S$ satisfying $h(a) - h(b) < 1$, we must have $\Pr(b \succ a) < \frac{8}{11}$. Thus, for any poset $(S, >)$, if we sort all elements of S in order $(a_{k_1}, a_{k_2}, \ldots, a_{k_n})$ such that $h(a_{k_1}) \le h(a_{k_2}) \le \cdots \le h(a_{k_n})$, then for any $i < j$ we will have $h(a_{k_i}) - h(a_{k_j}) \le 0 < 1$, and thus $\Pr(a_{k_j} \succ a_{k_i}) < \frac{8}{11}$. This implies that this is a good order as we want.

Therefore, it remains to compute $h(a)$ efficiently for every element a. However, it was shown in [18] that counting the number of linear extensions of a given poset is #**P**-complete, and determining the average height of an element of a poset is polynomially equivalent to the linear extension counting problem, thus is also #**P**-complete. Luckily, to our purpose of having a good order, an approximation of $h(a)$ to a small enough precision suffices. To this end, we first notice that there exists a fully polynomial randomized approximation scheme (FPRAS) for the problem of counting the number of linear extensions [28], where the algorithm finds, with probability $1-\delta$, a $(1+\epsilon)$-approximation of the number of the linear extensions in time $poly(n, 1/\epsilon, \log(1/\delta))$. Now given a poset $(S, >)$ and two elements $a, b \in S$, we can apply this algorithm to posets $(S, >)$, getting an output n_1, and apply the algorithm on $(S, >')$, getting an output n_2, where $>'$ is obtained from $>$ by incorporating an extra relation $(a > b)$. Then $\Pr(a \succ b)$ can be approximated by n_2/n_1 (with the precision $\frac{1+\epsilon}{1-\epsilon}$). It is easily seen from the definition of $h(a)$ and that of $\Pr(b \succ a)$ that

$$h(a) = \sum_{b \neq a} \Pr(b \succ a).$$

By setting $\epsilon = \frac{1}{5n}$ to approximate the value of each $\Pr(b \succ a)$ and using them to compute the value of $h(a)$, we can derive in polynomial time a value $h'(a)$ such that $1 - \frac{1}{2n} < \frac{h'(a)}{h(a)} < 1 + \frac{1}{2n}$ with high probability. Since $h(a) < n$, we have

$$|h'(a) - h(a)| < \frac{h(a)}{2n} < 0.5.$$

Using $h'(a)$ to sort all elements in S in order $(a_{k_1}, a_{k_2}, \ldots, a_{k_n})$, we have by the above inequality that $h'(a_{k_i}) < h'(a_{k_j})$ for any $i < j$ with arbitrarily high probability. This implies $h(a_{k_i}) - h(a_{k_j}) < 1$, and thus, $\Pr(a_{k_j} \succ a_{k_i}) < \frac{8}{11}$. Therefore, this is a good order for the current poset $(S, >)$. The whole process can be done in polynomial time, which completes the proof of the upper bound side.

For the lower bound side, it was also shown in [40] that for any poset $(S, >)$, there exist two elements $a, b \in S$ such that $\Pr(a \succ b) > \frac{3}{11}$ and $\Pr(b \succ a) > \frac{3}{11}$. Thus, we construct the verification oracle as follows. At any step, when the previously returned information forms a poset $(S, >)$, no matter what the current proposed order is, the oracle always returns such (a, b) (or (b, a), depending on their relative position in the proposed order) as a violation. Then after this trial, at least $\frac{3}{11}$ fraction of the possible linear extensions still remains. Therefore, we have $\mathsf{R}(\mathsf{Sorting_u}) \ge \log_{11/3} n! = \Omega(n \log n)$. □

Having the upper bound result for $\mathsf{Sorting_u}$, we can consider the preference of each individual as a sorting problem and solve these $2n$ $\mathsf{Sorting_u}$ problems together, which gives us the desired upper bound for the $\mathsf{StableMatching_u}$ problem.

Proof of Theorem 5. At each step, since our collected information gives a poset $(W, >_m)$ for each man m and a poset $(M, >_w)$ for each woman. We can use (part of) the above algorithm for $\mathsf{Sorting_u}$ to compute in polynomial time a good order $\succ_m$ for each man m and a good order $\succ_w$ for each woman w with respect to their current posets. Then we take these good orders as their preferences and compute a stable matching μ using Gale-Shapley's algorithm. Then we make a query μ to the verification oracle for $\mathsf{StableMatching_u}$. If it is not a stable matching, then V returns a blocking pair (m, w) satisfying $w \succ_m \mu(m)$ and $m \succ_w \mu(w)$. Notice that at least one of these two new relations conflicts with the assumed preferences $\{\succ_m, \succ_w\}$, which means that it can serve a valid verification oracle for $\mathsf{Sorting_u}$. Since (i) there are totally n man and n woman, (ii) at each step at least one man or woman gets a new pair of elements to update their posets, and (iii) by Lemma 6 we know that each poset will be updated at most $O(n \log n)$ times, we know that this stable matching algorithm calls the verification oracle $2n \cdot O(n \log n) = O(n^2 \log n)$ times. Finally, since all computation between trials can be done in polynomial time, the time complexity of the algorithm is in polynomial. □

Note that the same results for deterministic algorithms hold; that is, there is an exponential-time deterministic algorithm using $O(n \log n)$ trials for $\mathsf{Sorting_u}$ (and thus another algorithm using $O(n^2 \log n)$ trials for $\mathsf{StableMatching_u}$), namely, $\mathsf{D}(\mathsf{Sorting_u}) = O(n \log n)$ and $\mathsf{D}(\mathsf{StableMatching_u}) = O(n^2 \log n)$. The reason is simple: we can compute the average height $h(a)$ for every a by enumerating all linear extensions of the current partial order. See Section 9 for more discussions on polynomial time deterministic algorithms.

Note that our algorithm for $\mathsf{StableMatching_u}$ essentially runs $2n$ $\mathsf{Sorting_u}$ instances to learn the entire unknown input of the preference lists, and analysis of the trial cost simply involves adding the trials made on these instances. Although the $\Omega(n \log n)$ lower bound for $\mathsf{Sorting_u}$ implies a limit to this approach, there seems to be considerable room for improvement. It may be unnecessary to learn all of the input lists to find a solution, and there may be more sophisticated ways to solve $2n$ instances of $\mathsf{Sorting_u}$ to beat the naive upper bound by addition. However, the following theorem gives an almost matching lower bound for $\mathsf{StableMatching_u}$.

Theorem 7. *Any randomized algorithm solving* $\mathsf{StableMatching_u}$ *needs at least* $\Omega(n^2)$ *trials even with unbounded computational power.*

Proof. For each man m, we create a directed graph G_m with vertex set W and edge set initially empty; similarly create a directed graph G_w for each woman w. For each proposed matching μ, after a blocking pair (m, w) is returned by the verification oracle, we know that m prefers w to $\mu(m)$ and w prefers m to $\mu(w)$. We update the graphs G_w and G_m as follows. If there is no directed path from $\mu(m)$ to w, we add an edge $(\mu(m), w)$ in the graph G_m. Similarly, if there is no directed path from $\mu(w)$ to m, we add an edge $(\mu(w), m)$ in G_w. Since we have made k queries to verification oracle, there are at most k edges altogether in all graphs G_m and at most k edges in all graphs G_w. Note that these $2n$ graphs are all the information an algorithm gets from the trials and answers.

Our proof is by probabilistic method. Suppose we have already made k queries.We pick a pair (m, w) uniformly at random in all possible n^2 pairs, and will show the following property for any matching μ: If $k < (n^2 - n)/2$, then with a strictly positive probability, (m, w) is a blocking pair for μ on some input instance consistent with the queried information so far. Therefore, k queries to the verification oracle are not sufficient to guarantee to find a stable matching, as for any candidate output μ, there are still instances with blocking pairs for it.

The analysis of the probability goes as follows. First, with probability $1 - 1/n$, the pair is not in the current matching μ. Second, we claim that with probability at least $1 - k/n^2$, m could prefer w to $\mu(m)$. Here "could" means that there is an input instance consistent with the previous trials and answers but in the input m prefer w to $\mu(m)$. Indeed, this could not happen only if the known preferences of m already imply that w is less preferred than $\mu(m)$; namely there is a path from w to $\mu(m)$ in the graph G_m. But there are not many preferences known—on average, only k/n preferences known for m. Formally, for a randomly chosen w, the event that "w is known to be less preferred than $\mu(m)$ by m" happens with probability

$$\frac{1}{n} \cdot \big(\text{the number of nodes that can reach } \mu(m) \text{ in } G_m\big) \le \frac{1}{n} \cdot \big(\text{the number of edges in } G_m\big)$$

So averaging over all men gives that

$$\begin{aligned} \Pr\big[m \text{ prefers } w \text{ to } \mu(m)\big] &\ge 1 - \frac{1}{n} \cdot \frac{1}{n} \cdot \big(\text{the total number of edges in all men's graphs}\big) \\ &\ge 1 - k/n^2. \end{aligned}$$

Third, similar analysis shows that with probability at least $1 - k/n^2$, w could prefer m to her current assignment $\mu(w)$. Putting all these together, with probability at least $1 - \frac{1}{n} - \frac{2k}{n^2}$, the pair (m, w) can be a blocking pair. The probability is strictly positive if $k < (n^2 - n)/2$, meaning the existence of a blocking pair. Therefore k needs to be larger than $(n^2 - n)/2$. □

A final comment is that in the traditional stable matching problem, where the preferences are known, there is a tight lower bound $\Omega(n^2)$ for computing a stable matching [56]. However, this bound refers to the computational time complexity in a completely different meaning from our trial complexity lower bound.

3 SAT

Given a CNF formula ϕ with n variables and m clauses, the SAT search problem is to find a satisfying assignment to ϕ if one exists, or to return "ϕ is unsatisfiable." In the unknown-input version $\mathsf{SAT_u}$, the formula ϕ is unknown, and each time that a proposed assignment x is not a satisfying assignment, the verification oracle returns an index i such that the i-th clause of ϕ evaluates to FALSE on x.

First, we clarify that the computation oracle for SAT is defined as follows. On a query ϕ which is a CNF formula, SAT returns a satisfying assignment x to ϕ, or reports that such x does not exist. So this SAT oracle solves the search problem instead of the decision problem. This is for the fair comparison since the target $\mathsf{SAT_u}$ is also a search problem. Also note that the search and the decision problems for SAT are roughly the same due to the standard self-reducibility of SAT.

We have the following algorithm to solve the $\mathsf{SAT_u}$ problem.

ALG-SAT

Unknown input: A CNF formula ϕ of n variables and m clauses.

Let $L_1 = L_2 = \cdots = L_m = \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$.
loop
  Let $\phi' = \bigwedge_{i=1}^{m} (\vee_{\ell \in L_i} \ell)$.
  Query the computation oracle SAT on ϕ'.
  if the computation oracle says that ϕ' is not satisfiable, **then**
    return ϕ is unsatisfiable (and terminate the program).
  else
    Suppose the computation oracle gives a satisfying assignment x of ϕ'.
    Ask the assignment x to the verification oracle V.
    if V confirms that $\phi(x) = 1$ **then**
      return x (and terminate the program).
    else
      Suppose V returns an index i.
      Let $L_i \leftarrow \{\ell \in L_i : \ell(x) = 0\}$ (i.e., those literals in L_i evaluating FALSE on x).
    end if
  end if
end loop

The algorithm initially includes all literals, i.e., $x_1 \vee \bar{x}_1 \vee \cdots \vee x_n \vee \bar{x}_n$, in each clause. It proceeds by employing the SAT computation oracle to propose an assignment x consistent with the current knowledge of the clauses. If a clause's index is returned by the verification oracle upon a trial, then we know that x_i cannot be in the clause if $x_i = 1$ and that $\bar{x}_i$ cannot be in the clause if $x_i = 0$ in the assignment of the trial. We therefore remove the literals from the clause, and continue the process until either a satisfying assignment is found or the computation oracle returns the result that no satisfying assignment exists.

One may feel resemblance to the standard self-reducibility approach of finding a satisfying assignment by queries to an oracle solving the decision version of SAT. Note that our algorithm directly shrinks the space of possible inputs (i.e., formulas), instead of the space of possible solutions (i.e., assignments). (While some variables may have their values fixed along the process of the algorithm, this may not necessarily be the case and is surely not the reason why our algorithm works within $O(mn)$ queries.)

Theorem 8. ALG-SAT *solves the* $\mathsf{SAT}_{\mathsf{u}}$ *problem in polynomial time using* $O(mn)$ *trials, where* n *and* m *are the numbers of variables and clauses, respectively.*

Proof. Correctness. The set L_i maintains a collection of possible literals for the i-th clause in the unknown formula ϕ. Note that if ϕ is satisfiable, so is ϕ', at any step of the algorithm. Indeed, at the beginning all literals are included in each clauses and ϕ' is trivially satisfiable. Each time L_i is updated, the literals that are removed from L_i are exactly those ℓ with $\ell(x) = 1$. But the literals in the actual i-th clause C_i of ϕ all evaluate to 0, because $C_i(x) = \vee_{\ell \in C_i} \ell(x) = 0$. Thus, all the literals in C_i are kept when L_i is updated. Therefore a satisfying assignment x to ϕ also satisfies ϕ'.

Also note that if $\phi'(x) = 1$, then there exists at least one $\ell \in L_i$ such that $\ell(x) = 1$. So the size $|L_i|$ decreases by at least 1 each time L_i is updated. Since ϕ' is always satisfiable and $\sum_i |L_i|$ decreases by at least 1 in each iteration, the algorithm finally stops and outputs an x with $\phi(x) = 1$. On the other hand, if ϕ is unsatisfiable, then $\phi(x)$ never evaluates to 1, thus finally the algorithm outputs that ϕ is unsatisfiable.

Complexity. Since initially $|L_1| = \cdots = |L_m| = 2n$, the algorithm runs in at most $2nm$ rounds. In each round finding a satisfying assignment of ϕ' takes one query to the computation oracle, so the trial complexity are $O(nm)$. The time complexity of the algorithm is $poly(n)$ as all computations between queries can be done in polynomial time. □

We also have a lower bound on the randomized trial complexity of $\mathsf{SAT_u}$, which matches the upper bound at least when $m = \Omega(n^2)$.

Theorem 9. *If $m = \Omega(n^2)$, any randomized algorithm for the $\mathsf{SAT_u}$ problem needs at least $\Omega(mn)$ trials, even with unbounded computational power. If $m = o(n^2)$, the lower bound is $\Omega(m^{3/2})$.*

The proof of the lower bound is combinatorially involved, in which we construct instances such that only one candidate literal can be eliminated in the foregoing process. This is enforced with the help of certain clauses that confine all satisfying assignments to a very special form; see appendix for details.

Proof. We will first show the lower bound of $\Omega(n^3)$ for the special case when $m = \Theta(n^2)$. Then we will consider the cases when $m = \omega(n^2)$ and $m = o(n^2)$.

Let $m^* = \binom{n/3}{2} + \frac{n^2}{9} + 3$. We will construct a family of CNF formulas with n variables and m clauses, where $m^* \le m \le O(n^2)$. Without loss of generality, we assume that 3 divides n. Divide $[n]$ into three equal blocks: $B_1 = \{1, 2, \ldots, \frac{n}{3}\}$, $B_2 = \{\frac{n}{3} + 1, \frac{2n}{3} + 2, \ldots, \frac{2n}{3}\}$, and $B_3 = \{\frac{2n}{3} + 1, \frac{2n}{3} + 2, \ldots, n\}$. We, as an adversary constructing the verification oracle, maintain a set T of triples (i_1, i_2, i_3), with T initially containing (i_1, i_2, i_3) with all $i_1 \in B_i$, $i_2 \in B_2$ and $i_3 \in B_3$. Each triple (i_1, i_2, i_3) represents an assignment, denoted by $x(i_1, i_2, i_3)$, where $x_i = 1$ if and only if $i \in \{i_1, i_2, i_3\}$. We will show that some of the formulas in the constructed family have a unique satisfying assignment of the form $x(i_1, i_2, i_3)$, which can only be found by at least $(\frac{n}{3})^3$ queries for any randomized algorithm and a carefully designed verification oracle.

The hidden formula ϕ has $\Theta(n^2)$ clauses, divided into two parts, each with $\Theta(n^2)$ clauses. The first part ensures that any satisfying assignment of ϕ needs to have exactly one variable $x_i = 1$ in the last block. This can be done by $\Theta(n^2)$ clauses because we use one clause $\vee_{i \in B_3} x_i$ to enforce that there is at least one $x_i = 1$ in that block, and then use $\bar{x}_i \vee \bar{x}_j$ to enforce that at most one of x_i and x_j is assigned to be 1. Thus, this part of formula

$$(\vee_{i \in B_1} x_i) \bigwedge (\vee_{i \in B_2} x_i) \bigwedge (\vee_{i \in B_3} x_i) \bigwedge_{i,j \in B_3 : i \neq j} (\bar{x}_i \vee \bar{x}_j)$$

ensures that a satisfying assignment has at least one $x_i = 1$ in each of the first two blocks, and exactly one $x_i = 1$ in the last block. This part has $\binom{n/3}{2} + 3 = \Theta(n^2)$ clauses.

The second part of the formula consists of $\frac{n^2}{9}$ clauses, indexed by the pairs (i_1, i_2) for each $i_1 \in B_1$ and $i_2 \in B_2$. The clause associated with (i_1, i_2) is either $(\bar{x}_{i_1} \vee \bar{x}_{i_2})$ or $(\bar{x}_{i_1} \vee \bar{x}_{i_2} \vee x_{i_3})$ for some $i_3 \in B_3$. For an arbitrary randomized algorithm, on each query x, if x does not satisfy the first part of formula, then we return the corresponding clause in that part. (This first part can be even revealed to the algorithm for free.) Now assume that x satisfies the first part and it has three positions i_1, i_2, i_3 being 1, one in each block. (If there are multiple 1's in the first two blocks, let i_1 and i_2 denote the positions of the first 1's in the corresponding blocks.) Let the verification oracle return the clause with index (i_1, i_2); note that this implies that the clause cannot be of the form $(\bar{x}_{i_1} \vee \bar{x}_{i_2} \vee x_{i_3})$. We then update the candidate set T by removing one element (i_1, i_2, i_3) from it. Note that this is also all the information the algorithm obtains from this query, that is, for any $(i'_1, i'_2, i'_3) \in T$, $(i'_1, i'_2, i'_3) \neq (i_1, i_2, i_3)$, $x(i'_1, i'_2, i'_3)$ can still be a possible satisfying assignment.

This process continues; we claim that as long as $|T| \ge 2$, namely it contains at least two distinct triples (i_1, i_2, i_3) and (i'_1, i'_2, i'_3), then the formula can be either with a unique satisfying assignment $x(i_1, i_2, i_3)$ or with a unique satisfying assignment $x(i'_1, i'_2, i'_3)$, thus the algorithm does not know what to output yet. Indeed, all previous queries by the algorithm cannot distinguish the following two possibilities: The second part of the formula can be either

$$\phi_1 = \bigwedge_{(j_1,j_2) \neq (i_1,i_2)} (\bar{x}_{j_1} \vee \bar{x}_{j_2}) \bigwedge (\bar{x}_{i_1} \vee \bar{x}_{i_2} \vee x_{i_3}) \quad \text{or} \quad \phi_2 = \bigwedge_{(j_1,j_2) \neq (i'_1,i'_2)} (\bar{x}_{j_1} \vee \bar{x}_{j_2}) \bigwedge (\bar{x}_{i'_1} \vee \bar{x}_{i'_2} \vee x_{i'_3})$$

where j_k is in block B_k.

We claim that both formulas are satisfiable, and actually each has a unique satisfying assignment, namely $x(i_1, i_2, i_3)$ and $x(i'_1, i'_2, i'_3)$, respectively. Let us consider ϕ_1 as an example. First, it is easy to see that $x(i_1, i_2, i_3)$ does satisfy ϕ_1. Second, the assignment $x(i_1, i_2, i_3)$ is the only satisfying assignment. Suppose x satisfies ϕ_1, then to satisfy $(\vee_{i\in B_1} x_i) \bigwedge (\vee_{i\in B_2} x_i) \bigwedge_{(j_1,j_2)\neq(i_1,i_2)} (\bar{x}_{j_1} \vee \bar{x}_{j_2})$, we have $x_{i_1} = x_{i_2} = 1$ and $x_i = 0$ for $i \in B_1 \cup B_2 - \{i_1, i_2\}$. Otherwise, if $x_{j_1} = 1$ for any $j_1 \neq i_1$ in B_1, then $x_{j_2} = 0$ for all $j_2 \in B_2$, violating $\vee_{i\in B_2} x_i$. Thus, $x_{j_1} = 0$ for all $j_1 \neq i_1$ in B_1; the clause $(\vee_{i\in B_1} x_i)$ then forces $x_{i_1} = 1$. Similarly we can show that i_2 is the only position in B_2 with assignment 1. Now to satisfy the last clause in ϕ_1, x_{i_3} must be 1, and the first part of formula guarantees that i_3 is the only position i in B_3 with $x_i = 1$. Thus, $x(i_1, i_2, i_3)$ is the only satisfying assignment.

Since initially $|T| = (n/3)^3$ and each query decreases $|T|$ by 1, we know that the algorithm needs at least $(n/3)^3 = \Omega(n^3)$ queries in the worst case.

Now consider the case that $m = \omega(n^2)$. Suppose $c > 2$ is the minimum integer that $m \leq (\frac{n}{2c})^c$. Divide the variables into $c + 1$ blocks, with the first c blocks of size $k_1 = (\frac{m}{2})^{1/c} < \frac{n}{2c}$, and the last block of size $k_2 = n - ck_1$. Note that $k_1 c \leq \frac{n}{2}$ and thus $k_2 \geq \frac{n}{2}$. By a similar setting of the previous formulas, we can use $\binom{k_2}{2} + 1$ clauses to ensure that all the satisfying assignments have exactly one $x_i = 1$ in the last block, and c clauses to ensure that there is at least one $x_i = 1$ in each of the first c blocks. We also use k_1^c clauses to hide a unique tuple $(i_1, \ldots, i_{c+1})$, with each i_ℓ in block ℓ, such that the assignment x with $x_i = 1$ if and only if $i = i_\ell$, for $\ell \in [c+1]$, is the unique satisfying assignment. Then we use $k_1^c + \binom{k_2}{2} + 1 + c < m$ clauses to hide the satisfying assignment, to find which needs $k_1^c k_2 \geq \frac{m}{2} \cdot \frac{n}{2} = \Omega(mn)$ queries.

Finally, for the case of $m < m^*$, we only use $\Theta(\sqrt{m})$ variables. The problem is then reduced to the first case, which gives a lower bound of $\Omega(m\sqrt{m}) = \Omega(m^{3/2})$. □

Note that the above proof can be easily adapted to show the same lower bound for the decision problem, namely to decide whether a formula (with unknown clauses) is satisfiable. Actually the verification oracle in the above proof leaves the possibility of ϕ being satisfiable until the very last step: By choosing to use two extra clauses, one is x_1 and the other is $\bar{x}_1$, we can make the formula unsatisfiable. We will not reveal these two clauses until exhausting all previous m clauses. Hence, the same lower bound holds in this decision problem as well.

4 Group Isomorphism

Given two groups, G and G', of the same size, the group isomorphism (GroupIso) problem is to find an isomorphism between G and G' or to report that "$G \ncong G'$". More precisely, given two groups, G and G', by their multiplication tables, $T_{n\times n}$ and $T'_{n\times n}$, our task is to output a bijection $\pi : G \to G'$ such that

$$\pi(a \circ b) = \pi(a) \circ' \pi(b) \tag{2}$$

for all $a, b \in G$ (where $\circ$ and $\circ'$ are the multiplications of G and G', respectively), or to report that "$G \ncong G'$".

Whether a polynomial time algorithm exists for the general GroupIso problem is a long-standing open question. Compared with another well-known problem, Graph Isomorphism, however, GroupIso has more group structures for potential use, and indeed, GroupIso can be solved in polynomial time if the given groups are Abelian, and (the decision version of) GroupIso is in **NP** ∩ **co-NP** (under certain complexity assumptions) if the given groups are solvable [8].

The problem is by nature a constraint satisfaction problem, searching for a bijection π that satisfies the n^2 constraints Eq.(2). In the unknown-input model, we can consider the case of both groups being

unknown and that of exactly one group, say, G, being unknown. In either case, we can propose bijections π to the verification oracle V. If π is not an isomorphism, then V gives a pair (a, b) with the forregoing equality violated. Our upper bound result in this section applies to the general case in which both groups are unknown, and our lower bound result applies even to the restricted case in which one group is known.

A very natural attempt to solve $\mathsf{GroupIso}_{\mathsf{u}}$ is to keep proposing bijections π that are consistent with our current knowledge of the restrictions of a valid bijection. More precisely, whenever V returns a pair of elements (a, b) on a proposed π, it adds the restriction that we cannot *simultaneously* map the three elements $(a, b, a \circ b)$ to $(\pi(a), \pi(b), \pi(a) \circ' \pi(b))$. Thus, there are no more than $O(n^6)$ forbidden rules. Note that finding a bijection that avoids a list of forbidden pairs of triples is easy with an **NP** computation oracle. Thus, $\mathsf{GroupIso}_{\mathsf{u}}$ can be solved using SAT as the computation oracle in polynomial time and via $O(n^6)$ trials.

An immediate question that arises from this claim is the following. Does it hold only with a computation oracle $\mathsf{GroupIso}$ instead of SAT? (After all, only comparison with $\mathsf{GroupIso}$ reveals the extra difficulty arising from the unknown input on the problem.) This seems quite conceivable that this is the case, as group theory by definition handles triples in the form $(\pi(a), \pi(b), \pi(a) \circ' \pi(b))$, and we have not yet exploited the group structures in the given tables. Surprisingly, this intuition turns out to be misleading, as shown by the following hardness result, which stands even if one group G' is known to us and is a very simple group $\mathbb{Z}_p$ for a prime p. Denote the known-input version of this problem by $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)$. Note that because it is solvable in polynomial time, the computation oracle is not needed (module a polynomial factor in running time) in the unknown-input model.

Theorem 10. *If* $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_{\mathsf{u}}$ *can be solved in polynomial time, i.e.,* $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_{\mathsf{u}} \in \mathbf{P}^{\mathsf{V}}$*, then* $\mathbf{P} = \mathbf{NP}$*. More specifically, if* $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_{\mathsf{u}}$ *can be solved in time* $t(p)$*, then* $\mathsf{HamiltonianCycle}$ *can be solved in time* $O(t(p) \cdot p)$*, where* p *is the order of the given group* $\mathbb{Z}_p$*.*

Idea of the proof. The hardness result is shown by a reduction that is not very standard. Given a graph H with p vertices, we employ an algorithm $\mathcal{A}$ for $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_{\mathsf{u}}$ to find a Hamiltonian cycle in H in the following way. Assuming the existence of a Hamiltonian cycle C (it can be seen that the primality of the size of the graph and the assumption of the existence of one Hamiltonian cycle do not alter the hardness of the Hamiltonian cycle problem), define a group T via C as follows. Let $(b_0, b_1, \ldots b_{p-1}, b_0)$ be a Hamiltonian cycle C in graph H. We can fix a vertex a and assume that $b_1 = a$, which is always achievable by a cyclic shift in the labels in the Hamiltonian cycle if necessary. For simplicity, we use the vertices of H to denote the elements of group T. Now, for any b_i and b_j in group T, define their multiplication by $b_i \circ b_j = b_{i+j \bmod p}$. It is easy to see that T is a cyclic group with p elements (where b_1 is a generator of the group).

The main idea of the reduction is to run algorithm $\mathcal{A}$ on input $(T, \mathbb{Z}_p)$, and translate the output of $\mathcal{A}$, which is an isomorphism from T to $\mathbb{Z}_p$, to a Hamiltonian cycle in the given graph H. However, one problem immediately arises: Because the reduction algorithm has only polynomial time, and it cannot find such a Hamiltonian cycle, thus cannot construct the multiplication table T.

To get around this issue, we employ the crucial fact that $\mathcal{A}$ does not know its first input—all of $\mathcal{A}$'s information about T comes from interactions with its verification oracle V. Thus, it is sufficient to construct a V that answers $\mathcal{A}$'s trials. However, doing so again requires the information on T, which is exactly what we do not have. Here, the idea is to efficiently construct a *simulator* V' to take the place of V. Given the shortage of running time, it is inevitable that we lose something in our simulator V', and, in our final construction it turns out to be the correctness, which is the seemingly the most critical component. In other words, V' cannot answer all of $\mathcal{A}$'s questions correctly. What makes it still qualified for our purpose is the following key property. On any π proposed by $\mathcal{A}$, V'

1. either provides a correct response to π, or

2. finds a Hamiltonian cycle in H.

This property means that the first time V' gives a wrong answer to $\mathcal{A}$, it has just found a Hamiltonian cycle in H. ($\mathcal{A}$'s output is admittedly now out of control now, but we no longer care about the correctness of A; we have used part of $\mathcal{A}$'s code to solve our HamiltonianCycle problem.) □

Formal Proof of Theorem 10. In the HamiltonianCycle problem, we are given a graph G with p vertices and are asked if it has at least one Hamiltonian cycle[6]. Suppose that there exists an algorithm $\mathcal{A}$ that solves the $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_\mathsf{u}$ problem in time $t(p)$. We now construct an algorithm $\mathcal{B}$ for the following variation of the HamiltonianCycle problem: Given a graph G with p vertices and the condition that it contains at least one Hamiltonian cycle, find a Hamiltonian cycle in G; call the problem PromisedHamiltonianCycle.

Now we are given a graph G with p nodes, we label the nodes in G as $a_1, a_2, \ldots, a_p$ in an arbitrary way. We will construct an algorithm $\mathcal{B}$ that uses algorithm $\mathcal{A}$ to find a Hamiltonian cycle in G, assuming one exists.

First we describe a cyclic group T[7] with elements $a_1, \ldots, a_p$. Let $(b_0, b_1, \ldots b_{p-1})$ be a Hamiltonian cycle in graph G. If there is more than one Hamiltonian cycle, pick an arbitrary one. We can further impose that $b_1 = a_1$, which is always achievable by a cyclic shift if necessary, since a Hamiltonian cycle contains every vertex. Now for any b_i and b_j in group T, define their multiplication by $b_i \circ b_j = b_{i+j \bmod p}$ (all additions hereafter are module p). It is easy to check that T is a cyclic group with p elements.

The algorithm $\mathcal{B}$ basically runs the algorithm $\mathcal{A}$ on input $(T, \mathbb{Z}_p)$. If finally $\mathcal{A}$ outputs a correct bijection π mapping T to $\mathbb{Z}_p$, then we can identify all b_i's and thus find a Hamiltonian cycle. However, we do not know how to provide a valid verification oracle, because in polynomial time we cannot find $(b_0, b_1, ..., b_p)$ and define the multiplication table of T as above. The idea here is to construct a simulator V' of the verification oracle in such a way that V'

1. either provides correct responses to $\mathcal{A}$'s trials, or

2. finds a Hamiltonian cycle for $\mathcal{B}$.

The V' and the algorithm $\mathcal{B}$ are given below.

Several explanations of the algorithms are in order. First, note that in the course of the algorithm, $\mathcal{B}$ defines T and runs $\mathcal{A}$ on input $(T, \mathbb{Z}_p)$. However, as we have mentioned, $\mathcal{B}$ actually does not know how to find a Hamiltonian cycle in polynomial time and to define T. Here we use the key property that $\mathcal{A}$ *does not* know its input: by saying "$\mathcal{B}$ runs $\mathcal{A}$", we mean to let $\mathcal{B}$ run $\mathcal{A}$'s code between trials; whenever $\mathcal{A}$ makes a trial π, $\mathcal{B}$ uses V' to simulate the true verification oracle V to give an answer.

This immediately raises the second issue: Our designed V' is not a valid verification oracle for the $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_\mathsf{u}$ problem, since it may return Yes for some wrong bijection. (That is, even if a bijection π does not really map T to $\mathbb{Z}_p$, our oracle V' may say Yes.) But what we can guarantee are the following two properties of V'. The first is roughly the soundness for the algorithm $\mathcal{A}$.

Claim 1. *If V' returns a violation (a_i, a_j) to a proposed π, it is indeed a violation.*

[6]The primality of the graph size does not change the hardness of this problem. For a given graph with n vertices for a general number n, one can first find a prime in $[n, 2n]$ (which takes time $n \cdot poly(\log n)$) and then for an edge (u, v) in the given graph G, add a path from u to v with $(p - n)$ extra vertices to form a new graph G'. It is easy to see that G' has a Hamiltonian cycle if and only if G has a Hamiltonian cycle using the edge (u, v). Fix an arbitrary u, try all neighbors v in G using the algorithm for the new instances and verify the solutions, we will know that whether G has a Hamiltonian cycle.

[7]Precisely, T is defined according to G and should be written as $T(G)$. As all our discussions are with respect to the given graph G, for simplicity we will use T in the proof.

Simulator V' given (G, π)

Input: Graph G, bijection $\pi : T \to \mathbb{Z}_p$

Let $x = \pi(a_1)$.
if $x = 0$ **then**
  return (a_1, a_1)
else
  if $\exists i \in \mathbb{Z}_p$ *s.t.* $(\pi^{-1}(ix), \pi^{-1}((i+1)x)) \notin E(G)$ **then**
    return $(\pi^{-1}(ix), a_1)$ for the first such i
  else
    return Yes
  end if
end if

Algorithm $\mathcal{B}$ for the PromisedHamiltonianCycle problem

Input: Graph G with at least one Hamiltonian Cycle

Suppose there is a Hamiltonian Cycle $(b_0, b_1, ..., b_{p-1})$; circularly shift the cycle to make $b_1 = a_1$.
Define a group T with the multiplication table given by $T(b_i, b_j) = b_{i+j \bmod p}$.
Run $\mathcal{A}$ on input $(T, \mathbb{Z}_p)$, during which:
if $\mathcal{A}$ makes a query π to the verification oracle V **then**
  run $\mathsf{V}'(G, \pi)$ to simulate V to give either Yes or a pair (a_i, a_j) as an answer
  if the answer is Yes **then**
    $x = \pi(a_1)$
    return a Hamiltonian cycle found: $\left(\pi^{-1}(0), \pi^{-1}(x), \pi^{-1}(2x), \ldots, \pi^{-1}((p-1)x)\right)$
  end if
end if
if $\mathcal{A}$ outputs σ **then**
  return a Hamiltonian cycle found: $\left(\sigma^{-1}(0), \sigma^{-1}(\sigma(a_1)), \sigma^{-1}(2\sigma(a_1)), \ldots, \sigma^{-1}((p-1)\sigma(a_1))\right)$
end if

Proof. If V' returns (a_1, a_1) in the outer **if** statement, then it is a violation because, as π is a bijection,

$$\pi(a_1 \circ_T a_1) = \pi(b_2) \quad \text{but} \quad \pi(a_1) + \pi(a_1) = 0 + 0 = 0 = \pi(a_1) = \pi(b_1) \neq \pi(b_2).$$

(Here we use $\circ_T$ to emphasize that it is a multiplication of group T).

The other possibility is that V' returns $(\pi^{-1}(ix), a_1)$: Suppose it is not a violation, then

$$\begin{aligned}
&(\pi^{-1}(ix), a_1) \text{ is not a violation} && && (3)\\
\Rightarrow\ &\pi(\pi^{-1}(ix) \circ_T a_1) = \pi(\pi^{-1}(ix)) + \pi(a_1) && && (4)\\
\Rightarrow\ &\pi(\pi^{-1}(ix) \circ_T a_1) = ix + x && (\pi(a_1) = x) && (5)\\
\Rightarrow\ &\pi^{-1}(ix) \circ_T a_1 = \pi^{-1}((i+1)x) && (\text{taking } \pi^{-1}) && (6)\\
\Rightarrow\ &(\pi^{-1}(ix), \pi^{-1}((i+1)x)) \in E && && (7)
\end{aligned}$$

where the last line is because $\circ_T\ b_1$ is defined as going to the next vertex along the Hamiltonian cycle $(b_0, b_1, ..., b_{p-1})$. □

The second property is roughly the soundness for $\mathcal{B}$; it relies on the fact that all non-zero elements of $\mathbb{Z}_p$ can generate the whole $\mathbb{Z}_p$.

Claim 2. *If V' returns Yes to a proposed π, the vertices $(\pi^{-1}(0), \pi^{-1}(x), \pi^{-1}(2x), \ldots, \pi^{-1}((p-1)x))$ in that order, as later outputted by $\mathcal{B}$, indeed form a Hamiltonian cycle of the graph G, regardless of whether π is a correct bijection.*

Proof. When V' returns Yes, it comes to the **else** branches of both the outer and inner **if-then-else** statements. The inner statement implies that all edges $(\pi^{-1}(ix), \pi^{-1}((i+1)x))$ exist. The outer statement implies that $x \neq 0$, and therefore $\{0, x, 2x, ..., (p-1)x\} = \{0, 1, ..., p-1\}$ since each non-zero is a generator of $\mathbb{Z}_p$ for prime p. Combining the two gives a claimed Hamiltonian cycle. □

By Claim 1, we know that before V' returns Yes, all answers of V' to $\mathcal{A}$'s trials are valid. And Claim 2 guarantees that once V' returns Yes, $\mathcal{B}$ already finds a Hamiltonian cycle (and terminates the program). Note that when V' returns Yes, it may not be a correct response to $\mathcal{A}$, but now we do not care the correctness or even the completeness of the execution of $\mathcal{A}$ any more, because we have already used $\mathcal{A}$'s code to serve our purpose of finding a Hamiltonian cycle for G.

The analysis so far shows that the algorithm $\mathcal{B}$ correctly outputs a Hamiltonian cycle as long as V' answers Yes. To finish the proof, we need to address the case that $\mathcal{A}$ halts before V' answers Yes. First, since the graph G does have a Hamiltonian cycle by promise, the group T as defined does exist (despite the fact that $\mathcal{B}$ could not really find it) and it is indeed isomorphic to $\mathbb{Z}_p$. Hence, a correct algorithm $\mathcal{A}$ cannot output No.

Now the only left case is that $\mathcal{A}$ may somehow infer, from the violations returned by V', a valid bijection σ that maps T to $\mathbb{Z}_p$ before V' returns Yes. In such a case, we can actually identify each $b_i = \sigma^{-1}(i\sigma(a_1))$: First, $b_0 = \pi^{-1}(0)$ since that is the only element which does not change by multiplying itself. Then, for all $i \geq 1$, we have

$$\begin{aligned}
\sigma(b_i) &= \sigma(\underbrace{b_1 \circ ... \circ b_1}_{i\ b_1\text{'s}}) && (\text{def of } \circ \text{ in } T) && (8)\\
&= \underbrace{\sigma(b_1) + ... + \sigma(b_1)}_{i \text{ times}} && (\sigma \text{ is a isomorphism}) && (9)\\
&= i \cdot \sigma(b_1) = i \cdot \sigma(a_1) && && (10)
\end{aligned}$$

Thus, $\left(\sigma^{-1}(0), \sigma^{-1}(\sigma(a_1)), \sigma^{-1}(2\sigma(a_1)), \ldots, \sigma^{-1}((p-1)\sigma(a_1))\right) = (b_0, b_1, \ldots, b_{p-1})$, as outputted by algorithm $\mathcal{B}$, is a Hamiltonian cycle.

Finally, the runtime of algorithm $\mathcal{B}$ is easily upper bounded by that of $\mathcal{A}$ times that of V'. So it takes $\mathcal{B}$ at most $O(t(p) \cdot p)$ time to solve the PromisedHamiltonianCycle problem. Now given a HamiltonianCycle problem instance with p nodes, we can run algorithm $\mathcal{B}$ on this instance and incorporate its polynomial time bound as a time limit. After the time limit, we check the output of the algorithm $\mathcal{B}$, and accept if it is indeed a valid Hamiltonian cycle. If algorithm $\mathcal{B}$ outputs a wrong Hamiltonian cycle or it fails to output one, we reject it. Thus in this way the HamiltonianCycle problem can be solved in $O(t(p) \cdot p)$ time too. This completes the proof. □

A few remarks about the above proof are in order.

1. The polynomial-time algorithm $\mathcal{B}$ cannot find a certificate of an **NP** statement but can create a simulator for its own purpose. This phenomenon is reminiscent of the simulator paradigm in some cryptographic notions such as zero-knowledge proofs. However, differences are also clear, because simulators in zero-knowledge proofs are used to show that the interaction to the prover is in some sense "useless", while the simulator in our proof is actually useful for forcing $\mathcal{A}$ to leak information of a witness (in our case, a Hamiltonian cycle).

2. One can generalize the theorem by allowing the algorithm $\mathcal{A}$ to also invoke a computation oracle C. Then our algorithm $\mathcal{B}$ can invoke the same oracle C during simulating $\mathcal{A}$, and thus the conclusion becomes that "if $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_{\mathsf{u}}$ can be solved in time $t(n)$ with computation oracle C and verification oracle V, then HamiltonianCycle can be solved in time $O(t(n)n^2)$ with computation oracle C". This implies that $\mathsf{GroupIso}(\cdot, \mathbb{Z}_p)_{\mathsf{u}}$ is not likely to be solved in polynomial time by the help of a computation oracle weaker than **NP**.

5 Graph Isomorphism

In the graph isomorphism (GraphIso) search problem, we are given two undirected graphs G_1 and G_2, and we are asked to find a bijection $\pi : V(G_1) \to V(G_2)$ *s.t.* $\forall i, j \in V(G_1)$,

$$(i, j) \in E(G_1) \Leftrightarrow (\pi(i), \pi(j)) \in E(G_2), \tag{11}$$

if such a permutation exists, and to output "$G_1 \ncong G_2$" otherwise. The graph isomorphism is a well-known **NP** problem whose complexity is still open.

In our unknown-input version, $\mathsf{GraphIso}_{\mathsf{u}}$, we can consider both the case that the two graphs are unknown, and the case that exactly one graph, say G_1, is unknown. The latter case corresponds to the situations where we want to compare an unknown object (such as a new chemical compound) to a known one (such as a known chemical compound).

In either setting, we can propose a bijection $\pi : V(G_1) \to V(G_2)$. If it is indeed an isomorphism, the verification oracle returns Yes; otherwise, it returns a pair $i, j \in V(G_1)$ violating the above equivalence Eq.(11).[8] Our upper bound result in this section applies to the model when both graphs are unknown, and our lower bound result applies to the model when one graph is known. Therefore, both of our upper and lower bound results are at the stronger sense.

[8]Note that a return of V only implies that exactly one edge $(i, j) \in E(G_1)$ or $(\pi(i), \pi(j)) \in E(G_2)$ exists, but does not tell which one. One may consider to define a stronger V revealing this further piece of information, but we will show that this distinction does not matter: Our algorithm works with the weaker V, and our hardness result holds even for the stronger V.

A natural try for an algorithm is, as in the algorithms for StableMatching and SAT, to keep proposing bijections π that are consistent with the current knowledge of the edge information of the two graphs. More precisely, each returned violation (i, j) implies that a homomorphism, if one exists, should not *simultaneously* map i to $\pi(i)$ and map j to $\pi(j)$ (or i to $\pi(j)$ and j to $\pi(i)$). Actually we can do better: Consider a bipartite graph H with $\binom{n}{2}$ nodes at each side, where the left and right hand side nodes are indexed by all pairs of different vertices (i, j) in G_1 and G_2, respectively. Starting from empty, H is updated as follows. Each time we propose a bijection π and V returns a pair (i, j), we add an edge $((i, j), (\pi(i), \pi(j)))$ in H. Then an efficient algorithm tries to propose a bijection π *s.t.* (i, j) and $(\pi(i), \pi(j)))$ are not in the same connected component in the current H, so that any newly returned (i, j) gives a new piece of information. More precisely, each added edge in H decreases the number of connected components by 1. Since initially all $2\binom{n}{2}$ nodes in H are isolated and finally the nodes form at least one components, the algorithm stops after at most $2\binom{n}{2} - 1$ trials.

The above analysis gives a trial-efficient algorithm to solve $\mathsf{GraphIso_u}$. When it comes to the time complexity, we need to address the question of how to find a π to avoid a collection of forbidden pairs. This can surely be done if we are given an **NP** oracle since *checking* a bijection avoiding a list of forbidden pairs is easy. This leads to the following proposition.

Proposition 11. $\mathsf{GraphIso_u}$ *can be solved using* SAT *as the computation oracle in polynomial time and by* $O(n^2)$ *trials.*

Of course, as in $\mathsf{SAT}_u \in \mathbf{P}^{\mathsf{V},\mathsf{SAT}}$, one naturally desires an algorithm using only GraphIso as the computation oracle for $\mathsf{GraphIso_u}$. It looks quite achievable: After all, graphs is by nature a collection of binary relations, and the well-developed graph theory is a large source of tools. Indeed, it is not hard to show by a simple probabilistic argument that one can propose a π to avoid $\Theta(n)$ existing forbidden pairs, so it is "merely" the matter of whether the process can continue to handle more forbidden pairs. However, these intuitions turn out to be wrong, as refuted by the following theorem about the necessity of the SAT computation oracle.

Theorem 12. *If* $\mathsf{GraphIso_u}$ *can be solved in time* $t(n)$ *with computation oracle* A *and verification oracle* V*, then* Clique *can be solved in time* $O(t(n)n^2)$ *with oracle* A*.*

Proof. We prove the claim for the model when one graph is known. We assume that there is an algorithm $\mathcal{A}$ that solves $\mathsf{GraphIso_u}$ in polynomial time. We will use it to design another algorithm $\mathcal{B}$ to solve the Clique problem — given a graph G and a number k, find a clique of size k in G, or claim that it does not exist.

For the given G, we construct a $\mathsf{GraphIso_u}$ instance as follows: Let the known graph $G_2 = G$ and the unknown graph be G_1 where $V(G_1) = \{1, \ldots, n\}$ and $E(G_1) = \{(i, j) \mid 1 \le i < j \le k\}$, *i.e.*, G_1 is composed of a k-clique with extra $n - k$ isolated nodes. We now apply algorithm $\mathcal{A}$ on the instance (G_1, G_2) with the following specific verification oracle $\mathcal{O}$ upon a query: Given a bijection π from $V(G_1)$ to $V(G_2)$, if it is indeed an isomorphism, then return Yes. Otherwise $\mathcal{O}$ returns a pair (i, j) that minimizes $\max\{i, j\}$ where the minimization is over all pairs (i, j) *s.t.* $(i, j) \in E(G_1)$, and $(\pi(i), \pi(j)) \notin E(G_2)$. If no such pair exists, it returns an arbitrary violated pair (for such a pair (i, j), it must be $\max\{i, j\} > k$). Note that $\mathcal{A}$ runs in polynomial time on the instance and $\mathcal{O}$ is efficiently implementable.

We next construct an algorithm $\mathcal{B}$ for the k-clique problem.

If G is precisely a k-clique plus $n - k$ isolated nodes, then G_1 is isomorphic to $G = G_2$, and $\mathcal{A}$ finally returns a correct bijection; thus $\mathcal{B}$ also finds the clique. If G contains a k-clique as well as some other edges, then G_1 is not isomorphic to G and $\mathcal{A}$ finally outputs No. This looks undesirable since $\mathcal{B}$ may also output "no k-clique" in the **else** branch. However, we claim that the verification oracle $\mathcal{O}$ always returns a

Algorithm $\mathcal{B}$ for k-clique

Input: Graph G

1: $G_1 \leftarrow ([n], \{(i,j) \mid 1 \le i < j \le k\})$, $G_2 \leftarrow G$.
2: Run algorithm $\mathcal{A}$ on (G_1, G_2) on oracle $\mathcal{O}$.
3: **if** ($\mathcal{A}$ outputs a bijection π) or ($\mathcal{O}$ returns a pair (i,j) with $\max\{i,j\} > k$ on $\mathcal{A}$'s query π) **then**
4: Return "$(\pi(1), \ldots, \pi(k))$ is a clique in G".
5: **else**
6: Return "no k-clique".
7: **end if**

pair i and j with $\max\{i,j\} > k$ before $\mathcal{A}$ can conclude with an answer No. Indeed, to conclude that G_1 is not isomorphic to G, $\mathcal{A}$ has to detect at least one pair (i,j) with $\max\{i,j\} > k$ and (i,j) is not an edge in G_1. (Otherwise, $\mathcal{A}$ only see edges within the first k nodes in G_1 and it is still possible in $\mathcal{A}$'s point of view that G_1 is isomorphic to G. Thus $\mathcal{A}$ cannot make any decision yet.) But the only way that $\mathcal{A}$ detects such a pair (i,j) is when $\mathcal{O}$ returns a pair (i,j) where $(\pi(i), \pi(j)) \in E(G)$ but $(i,j) \notin E(G_1)$. By the design of our oracle, this can happen only if $\max\{i,j\} > k$, as claimed.

Once $\mathcal{O}$ returns a pair (i,j) with $\max\{i,j\} > k$, all the edges $(\pi(i), \pi(j))$ in G exist; thus, $\mathcal{B}$ already finds a k-clique and outputs it.

On the other hand, if G does not contain a k-clique, then $\mathcal{A}$ never outputs a bijection. Further, for any π, there is at least one pair (i,j), $1 \le i < j \le k$, such that $(i,j) \in E(G_1)$ and $(\pi(i), \pi(j)) \notin E(G)$, and the oracle $\mathcal{O}$ as defined always returns one of such pairs. Hence, the algorithm $\mathcal{B}$ never outputs a k-clique before $\mathcal{A}$ finishes, at which time $\mathcal{B}$ gets to the last line and outputs "no k-clique".

Finally, for the time cost, each execution of $\mathcal{O}$ takes time $k^2 \le n^2$ and the number of queries to $\mathcal{O}$ in $\mathcal{A}$ is at most its time complexity $t(n)$, so the total time on $\mathcal{O}$ is $O(t(n)n^2)$. Other time cost mainly includes the non-query part of $\mathcal{A}$, which is at most $t(n)$, so claimed time bound holds. □

We have the following immediate corollary.

Corollary 13. *For any given computation oracle L (which is a class of languages),* $\mathsf{GraphIso_u} \in \mathbf{P}^{\mathsf{V},L}$ *if and only if* $\mathbf{NP} \subseteq \mathbf{P}^L$.

Proof. First, if $\mathbf{NP} \subseteq \mathbf{P}^L$, which means $\mathbf{P}^{\mathbf{NP}} \subseteq \mathbf{P}^L$, would imply $\mathsf{GraphIso_u} \in \mathbf{P}^{\mathsf{V},\mathbf{NP}} \subseteq \mathbf{P}^{\mathsf{V},L}$. Second, Theorem 12 will directly give us that if $\mathsf{GraphIso_u} \in \mathbf{P}^{\mathsf{V},L}$, then $\mathsf{Clique} \in \mathbf{P}^L$, which means $\mathbf{NP} \subseteq \mathbf{P}^L$. □

It was shown in [16] that if GraphIso is **NP**-complete, then the polynomial hierarchy (**PH**) collapses to the second level. The proof can be easily adapted to show a slightly stronger result that **PH** collapses to the second level even if GraphIso is **NP**-complete under Turing reduction[9]. This gives the following corollary. We include the proof for completeness.

Corollary 14. *If* $\mathsf{GraphIso_u} \in \mathbf{P}^{\mathsf{V},\mathsf{GraphIso}}$*, then the polynomial hierarchy (***PH***) collapses to the second level.*

Proof. It is sufficient to prove that $\Sigma_2 = \Pi_2$. We will show the inclusion $\Sigma_2 \subseteq \Pi_2$, and the other direction is similar. For any formula $\phi(x,y)$, where $x, y \in \{0,1\}^n$, we want to construct another formula $\phi'(r_1, \ldots, r_k; x, a_1, \ldots, a_k)$, where k and the lengths of all r_i's and a_i's are $poly(n)$, s.t.

$$\exists x \forall y\ \phi(x,y) = 1 \Leftrightarrow \forall (r_1 \ldots r_k) \exists (x, a_1 \ldots a_k)\ \phi'(r_1, \ldots, r_k, x, a_1, \ldots, a_k) = 1. \tag{12}$$

[9] Eric Allender later pointed out that this stronger result, as we guessed, was indeed known, e.g., in [66].

Next is the construction.

By Theorem 12, if $\mathsf{GraphIso_u} \in \mathbf{P}^{\mathsf{V,GraphIso}}$, then $\mathbf{NP} \subseteq \mathbf{P}^{\mathsf{GraphIso}}$, *i.e.*, any **NP** problem can be solved by a polynomial-time algorithm calling the GraphIso oracle at most $k = poly(n)$ times. By flipping the answer, the algorithms can also solve **co-NP** problems. Note that $\forall y\ \phi(x, y) = 1$ is a **co-NP** statement, so it can be solved by a polynomial-time algorithm with the GraphIso oracle.

It is well-known that there is an **AM** protocol for GraphNonIso with perfect completeness and soundness error less than 2^{-m}, for any m polynomial in the length of the input. Also note that there is a trivial **NP** proof for GraphIso, which is a special case of an **AM** protocol with perfect completeness and perfect soundness. So we can design a protocol of $2k$ rounds to solve a **co-NP**-complete problem. Basically, the verifier simulates the algorithm mentioned in the above paragraph for the **co-NP** problem. When it comes to the i-th query to the GraphIso oracle, the verifier sends r_i as if it is the **AM** protocol for GraphNonIso. Since it is a public-coin protocol, the verifier's code is deterministic except for the public random coins sent to the prover. So each time the prover knows the pair of graphs currently in the verifier's mind (as the input for the GraphIso oracle). So the prover is supposed to solve the graph isomorphism problem and to return the one-bit answer, followed by a proof of that answer.

Now we will define a polynomial-time verification process V' on input $(r_1, \ldots, r_k, x, a_1, \ldots, a_k)$, and then take ϕ' to be the formula induced by V' as in the standard Cook-Levin reduction, and show Eq.(12). Let V denote the predicate which the assumed algorithm for the **co-NP** problem uses, after all queries to the GraphIso oracle, to decide acceptance/rejection. Now let V' on $(r_1, \ldots, r_k, x, a_1, \ldots, a_k)$ be the following: Check each a_i is a valid answer respect to r_i, and if all pass, output $V(r_1, \ldots, r_k, x, a_1, \ldots, a_k)$.

Since no matter whether the answer is 0 or 1, the protocol always has perfect completeness. Therefore, for the Yes instances of the original Σ_2 language, we have

$$\exists x \forall y\ \phi(x, y) = 1 \tag{13}$$
$$\Rightarrow \exists x \forall r_1 \exists a_1 \ldots \forall r_k \exists a_k \quad V'(x, \phi, r_1, \ldots, r_k, a_1, \ldots, a_k) = 1 \qquad \text{(due to perfect completeness)} \tag{14}$$
$$\Rightarrow \exists x \forall r_1 \ldots \forall r_k \exists a_1 \ldots \exists a_k \quad V'(x, \phi, r_1, \ldots, r_k, a_1, \ldots, a_k) = 1 \qquad \text{(use the honest prover)} \tag{15}$$
$$\Rightarrow \forall r_1 \ldots \forall r_k \exists x \exists a_1 \ldots \exists a_k \quad V'(x, \phi, r_1, \ldots, r_k, a_1, \ldots, a_k) = 1 \qquad \text{(use the fixed } x\text{)} \tag{16}$$

On the other hand, for the No instances of the Σ_2 language, we have

$$\forall x \exists y\ \phi(x, y) = 0 \tag{17}$$
$$\Rightarrow \forall x, \text{for } (1 - 2^{-m})\text{-fraction of } r_1, \forall a_1, \ldots, \text{for } (1 - 2^{-m})\text{-fraction of } r_k, \forall a_k,$$
$$V'(x, \phi, r_1, \ldots, r_k, a_1, \ldots, a_k) = 0 \qquad \text{(small soundness error for each round)} \tag{18}$$
$$\Rightarrow \text{for } (1 - k2^{n-m})\text{-fraction of } (r_1, \ldots, r_k), \forall x \forall a_1 \ldots \forall a_k$$
$$V'(x, \phi, r_1, \ldots, r_k, a_1, \ldots, a_k) = 0 \qquad \text{(union bound)} \tag{19}$$
$$\Rightarrow \exists r_1 \ldots \exists r_k \forall x \forall a_1 \ldots \forall a_k \quad V'(x, \phi, r_1, \ldots, r_k, a_1, \ldots, a_k) = 0 \quad \text{(whenever } m > n + \log_2 k\text{)} \tag{20}$$

So if we pick $m = n + \lceil \log_2 k \rceil + 1$, then Eq.(12) holds, as desired. □

6 Nash Equilibrium

In a normal-form game, there are n players. Each player i has a strategy space S_i and a payoff function $u_i : S_1 \times \cdots \times S_n \mapsto \mathbb{Q}$, which gives the utility that i obtains for every strategy profile $(s_1, \ldots, s_n) \in S_1 \times \cdots \times S_n$. A joint probability distribution $(p_1, \ldots, p_n)$ on $S_1 \times \cdots \times S_n$ is called a *(mixed) Nash equilibrium* if for any

player i and any probability distribution p_i' on S_i, we have

$$\sum_{(s_1,\ldots,s_n)} \prod_j p_j(s_j) \cdot u_i(s_1,\ldots,s_n) \geq \sum_{(s_1,\ldots,s_n)} p_i'(s_i) \cdot \prod_{j\neq i} p_j(s_j) \cdot u_i(s_1,\ldots,s_n). \tag{21}$$

Note that the number of constraints given by the foregoing inequality is unbounded. It is well-known that a two-player game admits a mixed Nash equilibrium with polynomial size rationals, whereas games with three or more players may only have equilibria in irrational numbers [22]. The Nash problem is to find a Nash equilibrium in a normal-form game.

In the unknown-input version of a given game, denoted by $\mathsf{Nash_u}$, the payoff functions $u_i(\cdot)$ are unknown. We can query a mixed strategy $(p_1,\ldots,p_n)$ each time. If it is not an equilibrium, then the oracle will return a player i and one of his better responses p_i' where the foregoing inequality fails to hold.[10] Our trial and error model considers how fast a Nash equilibrium can be found from the viewpoint of a centralized authority, which is quite different from the learning models investigated in [29, 72] whose focuses are on the strategic dynamics formed by the behavior of individual players. We have the following result.

Theorem 15. *For any two-player game, there is a polynomial-time algorithm solving* $\mathsf{Nash_u}$ *given a computation oracle solving* Nash.

Idea of the proof. The proof is built on the existence of a Nash equilibrium in any game [55]. Assume that each player has m strategies. There are a total of $2m^2$ values in the two payoff matrices. Note that the Nash equilibrium solution space may not be convex; thus, we cannot employ the ellipsoid method to search for an equilibrium in the solution space. One observation is that the $2m^2$ values in the matrices correspond to a point U in the space $\mathbb{R}^{2m^2}$, which can also be seen as a degenerate polyhedron in $\mathbb{R}^{2m^2}$. For any given point $X \in \mathbb{R}^{2m^2}$, we can consider it as two payoff matrices of some game. If we compute a Nash equilibrium with respect to this game using the computation oracle and query it to the verification oracle, then the returned information (if it is not a Yes) actually gives us a hyperplane that separates X from the true point U. It is now tempting to claim that the problem is solved by the ellipsoid method. However, there is a remaining issue: in our problem, the solution polyhedron degenerates to a point and has volume 0. The standard approach in the ellipsoid method for handling such degenerated cases is to add perturbations to the constraints to introduce a positive volume of the feasible solution polyhedron. However, this approach is not applicable in our context, as we do not know the constraints explicitly. Luckily, we are able to employ the much more involved machinery developed by Grötschel, Lovász, and Schrijver [32, 33] to overcome this issue and thus solve the problem. □

Formal Proof of Theorem 15. Assume without loss of generality that each player has m strategies. Notice that there are totally $2m^2$ values in the payoff matrices given by u_1 and u_2; these values correspond to a point U in the space $\mathbb{R}^{2m^2}$. If we are able to construct a separation oracle for $\mathbb{R}^{2m^2}$ (which can also be seen as a degenerated polyhedron), then again we can apply the algorithm for Core to find this point and thus solve this problem. Therefore the remaining problem is how to construct such a separation oracle in polynomial time.

For any given point $X \in \mathbb{R}^{2m^2}$, consider it as two utility matrices x_1, x_2 of another game. We can first compute a Nash equilibrium (p_1, p_2) with respect to the two new utility matrices. Next we query this mixed strategy (p_1, p_2) to the verification oracle. If (p_1, p_2) is already a Nash equilibrium to the unknown utility matrices u_1 and u_2, then the verification oracle tells us so, and we have thus solved the problem.

[10] Note that the full set of strategies S_i may also be unknown; that is, in the process of trials, we query a probability distribution over those strategies that we have already observed. A deviation from a player can be either from the known strategies or "new" unknown strategies.

Now we assume that (p_1, p_2) is not a Nash equilibrium to the utility matrices u_1 and u_2. In this case we know that $X \neq U$. Suppose without loss of generality that the verification oracle returns that the first player has a better response p_1'. This means that

$$\sum_{(s_1,s_2)} p_1(s_1) \cdot p_2(s_2) \cdot u_1(s_1, s_2) < \sum_{(s_1,s_2)} p_1'(s_1) \cdot p_2(s_2) \cdot u_1(s_1, s_2)$$

Also notice that (p_1, p_2) is a Nash equilibrium to utility matrices x_1, x_2, thus we have

$$\sum_{(s_1,s_2)} p_1(s_1) \cdot p_2(s_2) \cdot x_1(s_1, s_2) \geq \sum_{(s_1,s_2)} p_1'(s_1) \cdot p_2(s_2) \cdot x_1(s_1, s_2)$$

This means that the vector $(p_1 - p_1') \otimes p_2$ (whose (s_1, s_2)-entry is $(p_1(s_1) - p_1'(s_1))p_2(s_2)$) can serve as the returned vector for the strong separation oracle (separating point U from point X), and we can use an approach developed by Grötschel, Lovász, and Schrijver [32, 33] that based on the ellipsoid method to solve this problem. Formally, the approach can be used to provide a polynomial-time algorithm for the strong nonemptiness problem for well-described polyhedra K given by a strong separation oracle ([33], Theorem 6.4.1). Here a strong nonemptiness problem is to decide whether K is empty, and if not, finding a point in K. Our (degenerated) polyhedron U is a single point and is thus well-described. A strong separation oracle is one that on a given point $y \in \mathbb{R}^n$ and a polyhedron K, finds a vector c such that $c^T y > \max\{c^T x : x \in K\}$. As argued above, we can construct such a strong separation oracle for point U easily in polynomial time (thanks to the computation oracle for solving Nash); thus, we can identify the exact value of these two utility matrices u_1, u_2 (or find a Nash equilibrium in some middle step). Once we find u_1, u_2, we call the computation oracle one more time to solve it. □

We would like to comment that here the polyhedron U is degenerated and has volume 0. In the known-input case, one can use the standard perturbation approach in the ellipsoid method to introduce a positive volume to U for handling this issue. However, the approach is not applicable in our context, as we do not know the constraints explicitly. Therefore, we have to use the much more involved machinery in [32, 33], as applied in the above proof, to find halfspaces that contain U, and do a sequence of dimension reductions. Fortunately and very interestingly, what is provided by the verification oracle just fits what the method requires.

For completeness, we briefly describe the idea of this general method. Given the separation oracle provided by the verification oracle, we use the ellipsoid method to ask queries iteratively: Initially we pick an ellipsoid that covers the entire domain $[0, N]^n$ and query the center of the ellipsoid. If a constraint is violated from the verification oracle, we establish a separation oracle and compute the next ellipsoid. (Note that a remarkable property of the ellipsoid method is that we do not even need to know the explicit expression of the violated constraint, and a separation oracle suffices for the algorithm to continue.) The process continues until either we find a point which is in F or the volume of the ellipsoid is sufficiently small such that F must have volume 0.

Note that if the system of inequalities has a solution, the numerator and denominator of all its components are bounded by $(nN)^n$. Thus, there is a lower bound on the volume of F if it is positive. After the volume of the ellipsoid gets smaller than that lower bound, which can be done in polynomial to n and $\log N$ number of steps, we can conclude that F is not full-dimensional, i.e., all points in F are lie on a hyperplane H. If we can find the hyperplane H in polynomial time, we know an extra (explicit) constraint and can reduce our problem to an $(n-1)$-dimensional case; then we can use the same method to solve it recursively.

Thus the remaining problem is how to identify the hyperplane H in polynomial time. This problem is solved by Grötschel, Lovász, and Schrijver in [32]. The general idea is that, having the ellipsoid with small enough volume that contains the solution polyhedra F, the ellipsoid must be very "thin" in the direction perpendicular to H. That is to say, if we take the shortest axis of this ellipsoid, let u be the unit vector which is parallel to this axis and v be the center of the ellipsoid, the hyperplane $u^T x = u^T v$ must be very close to our target hyperplane H. Then next we use the simultaneous diophantine approximation algorithm, which is a technique to round real numbers by rational numbers with relatively small sized denominator, to round the coefficients of $u^T x = u^T v$; this will finally give us the hyperplane H.

Although the aforementioned claim applies only to two-player games, our approach can also be generalized to n players to obtain an ϵ-approximate mixed Nash equilibrium for any constant $\epsilon > 0$. (Note that we cannot hope to compute an exact Nash equilibrium when $n \geq 3$, as such a solution may consist of irrational numbers.) However, there is one potential issue here: the input size of a game can be as large as $O(m^n)$, where m is the number of strategies of each player. Thus, for general games, our algorithm may require running time polynomial to $O(m^n)$ (which is still polynomial in the input size). However, for some multi-player games that admit concise representations, e.g., graphical games [43] on constant degree graphs, we can find an ϵ-approximate Nash equilibrium in time polynomial to m and n.

Our result implies that even if players are not completely aware of the rules of a game, we can still find a Nash equilibrium efficiently. Further, even if a Nash equilibrium has been achieved, the game itself can still remain a mystery (because beyond this point, the verification oracle cannot return any further information). The Internet provides one such example, as Scott Shenker remarked: "*The Internet is an equilibrium, we just have to identify the game*" [60].

In addition, we note that our approach can also be adopted to solve some other similar problems such as correlated equilibrium in games and competitive equilibrium in matching markets with prices (the details are quite similar to $\mathsf{Nash_u}$ and thus are omitted here).

7 Core of Cooperative Games

In a cost-sharing game, we are given a set A of n agents, and a cost function $c : 2^A \mapsto \mathbb{R}^+ \cup \{0\}$. Basically, the cost function gives the cost for any subset $S \subseteq A$ in order to let every agent in S be served. For example, $c(S)$ can be the cost to build a network that connects everyone in set S to the Internet. We assume that $c(\emptyset) = 0$ and the function is monotone, i.e., $c(S) \leq c(T)$ if $S \subseteq T$. We say a vector $\boldsymbol{\alpha} = (\alpha_i)_{i \in A} \in \mathbb{R}^n$ is in the *core* of the game if it satisfies the following two conditions:

- $\sum_{i \in A} \alpha_i = c(A)$.
- $\sum_{i \in S} \alpha_i \leq c(S)$ for every $S \subset A$.

Core is a central notion in cooperative game theory. The classic Bondareva-Shapley Theorem [15, 68] says that the core of a cooperative game is non-empty if and only if the cost function is fractionally subadditive. (A function is fractionally subadditive if there is a set of linear functions $f_1, \ldots, f_m$ such that $v(S) = \max\{f_1(S), f_2(S), \ldots, f_m(S)\}$ for any $S \subseteq A$.) Fractionally subadditive functions form a pretty general class which includes, e.g., additive functions, gross substitutes functions, and submodular functions as special cases [49].

In the corresponding unknown-input problem, denoted by $\mathsf{Core_u}$, the cost function $c(\cdot)$ is unknown; the information that we have is the number of agents n and an integer N which bounds the encoding length of every $c(S)$. (We assume that each $c(S)$ is given by two rational numbers presenting its numerator and denominator, respectively, whose values are therefore bounded by 2^N.) We can propose a vector

$\boldsymbol{\alpha} \in [0, N]^n$. If it is in the core, the verification oracle returns a Yes answer; otherwise, it returns the index of a violated constraint. Here the set of constraints contains precisely the linear constraints used to define the core, except that we replace the inequality $\sum_{i \in A} \alpha_i = c(A)$ by two inequalities: $\sum_{i \in A} \alpha_i \geq c(A)$ and $\sum_{i \in A} \alpha_i \leq c(A)$.[11]

Theorem 16. *There is an algorithm solving the* $\mathsf{Core_u}$ *problem in time polynomial in the input size.*

Note that when the cost function is additive, i.e., $\sum_{i \in A} c(i) = c(A)$, there is a unique solution to the core, i.e., $\alpha_i = c(i)$. An implication of our result is that we are able to identify these number $c(i)$'s precisely given the verification oracle. Further, together with Bondareva-Shapley Theorem [15, 68], our result immediately implies that we are able to determine if a given function is in the class of fractionally subadditive in polynomial trials.

Proof of Theorem 16. According to the definition, the core of a cost-sharing game is just the set F of solutions described by the following system of linear inequalities:

$$\begin{aligned} \sum_{i \in A} -x_i &\leq -c(A). \\ \sum_{i \in A} x_i &\leq c(A). \\ \sum_{i \in S} x_i &\leq c(S), \qquad \forall S \subset A. \end{aligned}$$

This is a system of linear inequalities with n variables and $2^n + 2$ constraints.

Each time when we propose a vector $\boldsymbol{\alpha}$, either we know that $\boldsymbol{\alpha} \in F$, which implies that $\boldsymbol{\alpha}$ is in the core and we are done), or we get a subset S where $\sum_{i \in S} \alpha_i > c(S)$. (The case that the first inequality is violated can be handled similarly.) Note that the value of $c(S)$ is still unknown to us, but we are able to get a strong separation oracle for $(F, \boldsymbol{\alpha})$. Because our polyhedron F has short representation and is thus well-described. Hence again we can apply the ellipsoid algorithm either to find a point $x \in F$ or to conclude that F is empty, solving the problem $\mathsf{Core_u}$. □

8 Subset Sum

Given a set $S = \{a_1, \ldots, a_n\}$ of n integers, the subset sum problem is to find a partition of S into two subsets S_1 and S_2 such that $\sum_{a \in S_1} a = \sum_{b \in S_2} b$, or report that such a partition does not exist.

In the unknown input version of subset sum problem, denoted by $\mathsf{SubsetSum_u}$, the values of these n integers are unknown to us. Each time we can propose a partition S_1 and S_2. If it is indeed a solution to the subset sum problem, the verification oracle will return Yes. Otherwise, the oracle will return which subset has a larger total value. That is, the violation is one of the following two constraints.

- $\sum_{a \in S_1} a \geq \sum_{b \in S_2} b$
- $\sum_{a \in S_1} a \leq \sum_{b \in S_2} b$

Theorem 17. *The* $\mathsf{SubsetSum_u}$ *problem has an exponential lower bound on trial complexity.*

[11] This is necessary to admit a polynomial time algorithm. For instance, when there is only one agent, the problem degenerates to find a given unknown rational number using queries. If the query is of the form "Is $x = y$?", it will take an exponential number of queries in the worst case; but if the query is of the form "Is $x \leq y$?", polynomial time algorithms are known [59, 48].

Proof. Consider the following instance of SubsetSum: $S = \{a_1, a_2, \ldots, a_{2n}\}$ of $2n$ integers, which can be divided into three categories:

- $a_1 = M + n + 2$.
- $a_2 = \cdots = a_n = M + 2$.
- $a_{n+1} = \cdots = a_{2n} = M + 3$.

Here M is a sufficiently large integer, such that in any partition, if the two sets have different sizes (number of integers), the larger set will always have a larger sum. Thus, it is easy to see that the only valid partition is given by $S_1 = \{a_1, a_2, \ldots, a_n\}$ and $S_2 = \{a_{n+1}, \ldots, a_{2n}\}$.

Given an algorithm for $\mathsf{SubsetSum}_\mathsf{u}$, since the instance S has a valid and unique partition solution, it should be able to find out the subset $\{a_{n+1}, \ldots, a_{2n}\}$ precisely, i.e., the exact partition (S_1, S_2). For any query (T_1, T_2), if T_1 and T_2 have different sizes, the oracle will always return the set with bigger size (which has a larger sum), and we cannot derive any information from this query. If T_1 and T_2 have the same size but are not exactly S_1 and S_2, it is always the case that the set containing a_1 has a larger sum. Thus, the only information we can derive is a subset of candidates of a_1. Therefore, in order to find the subset $\{a_{n+1}, \ldots, a_{2n}\}$, even if we already know the membership of a_1, in the worse case one needs as much as $\binom{2n-1}{n}$ queries; this gives the desired exponential lower bound. □

9 Concluding Remarks

In this paper, we propose a trial and error model to investigate search problems with unknown input. We consider a number of natural problems, and show that the lack of input knowledge may introduce different levels of extra difficulty in finding a valid solution. Our complexity results range from polynomially solvable, to **NP**-complete and exponential. Our model and results demonstrate the value of input information in solution finding from the computational complexity viewpoint.

The present work showcases a number of algorithms and lower bounds. Meanwhile, a number of important questions are left for future exploration. Closing the small gaps in Theorem 2 and examining more CSP problems are the obvious and specific ones. The following is a list of more problems and directions for further research.

- Information processing on hidden inputs is a common phenomenon in many scenarios, and the present work tries to address the related computational complexity issues in a specific and natural framework. What other general frameworks could be employed for systematic studies of hidden inputs from an algorithmic perspective?

- Our complexity results focus on either the trial or the time cost. It would be intriguing to consider the tradeoff between them. For instance, in $\mathsf{Sorting}_\mathsf{u}$ and $\mathsf{StableMatching}_\mathsf{u}$, our deterministic upper bounds $O(n \log n)$ and $O(n^2 \log n)$ for trial complexity are established by exponential-time algorithms. If *polynomial time* computation alone were allowed, then we would not have any bound better than $O(n^2)$ and $O(n^3)$ for $\mathsf{Sorting}_\mathsf{u}$ and $\mathsf{StableMatching}_\mathsf{u}$, respectively. (Note that the classic argument of graph entropy for sorting under a partial order [39] is not directly applicable here, as their allowable queries are of the standard form of pair comparison.)

  On the lower bound side, a natural question is whether any lower bound better than $\Omega(n \log n)$ and $\Omega(n^2 \log n)$ can be proven for $\mathsf{Sorting}_\mathsf{u}$ and $\mathsf{StableMatching}_\mathsf{u}$ with polynomial time computation power. Note that the bound, if possible, would probably be very difficult to prove because it implies that $\#\mathbf{P} \neq \mathbf{FP}$ (and thus $\mathbf{P} \neq \mathbf{PP}$).

- It is well known in decision tree complexity that deterministic and randomized complexities can be polynomially separated [20], and a fundamental open question is whether the gap exhibited by the NAND tree [65] is the largest possible. What separation between deterministic and randomized trial complexities could we have in our model? This question could also be considered in the polynomial-time computation framework.

- An algorithm in our model can access two oracles, verification and computation. In this paper, we consider only the complexity that interacts with the verification oracle. It is natural to ask about the query complexity of the other oracle (the problem is of particular importance when the computational complexity of the problem itself is quite large).

Acknowledgments

We are grateful to Shang-Hua Teng, Leslie Valiant, Umesh Vazirani, and Andrew Yao for their helpful discussions and comments. We also thank Eric Allender for bringing [66] to our attention, and to Graham Brightwell and Kazuo Iwama for pointing out [18] and [56], respectively.

References

[1] http://en.wikipedia.org/wiki/Trial_and_error.

[2] V. Agrawal, J. Manyika, and J. Richard. Matching people and jobs. *McKinsey Quarterly*, 2003.

[3] S. Albers and M. Henzinger. Exploring unknown environments. *SIAM Journal on Computing*, 29(4):1164–1188, 2000.

[4] S. Albers, K. Kursawe, and S. Schuierer. Exploring unknown environments with obstacles. *Algorithmica*, 32(1):123–143, 2002.

[5] D. Angluin. Queries revisited. *Theoretical Computer Science*, 313(2):175–194, 2004.

[6] D. Angluin, J. Westbrook, and W. Zhu. Robot navigation with distance queries. *SIAM Journal on Computing*, 30(1):110–144, 2000.

[7] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[8] V. Arvind and J. Torán. Solvable group isomorphism is (almost) in NP ∩ coNP. *ACM Transactions on Computation Theory*, 2(2):1–22, 2011.

[9] B. Awerbuch, M. Betke, R. Rivest, and M. Singh. Piecemeal graph exploration by a mobile robot. *Information and Computation*, 152(2):155–172, 1999.

[10] M. Balcan and A. Blum. A discriminative model for semi-supervised learning. *Journal of the ACM*, 57(3), 2010.

[11] M. Balcan, S. Hanneke, and J. Wortman. The true sample complexity of active learning. *Machine Learning*, 80:111–139, 2010.

[12] A. Barto and R. Sutton. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[13] D. Beaver, J. Feigenbaum, R. Ostrovsky, and V. Shoup. Instance-hiding proof systems. Technical Report 65, DIMACS, Rutgers University, 1993.

[14] A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrain. *SIAM Journal on Computing*, 26(1):110–137, 1997.

[15] O. Bondareva. Some applications of linear programming to cooperative games. *Problemy Kibernetiki*, 10:119–139, 1963.

[16] R. Boppana, J. Hastad, and S. Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25(2):127–132, 1987.

[17] G. Brightwell. Balanced pairs in partial orders. *Discrete Mathematics*, 201(1-3):25–52, 1999.

[18] G. Brightwell and P. Winkler. Counting linear extensions is #P-complete. In *23rd Annual ACM Symposium on Theory of Computing*, pages 175–181, 1991.

[19] N. Bshouty, E. Mossel, R. O'Donnell, and R. Servedio. Learning DNF from random walks. *Journal of Computer and System Sciences*, 71(3):250–265, 2005.

[20] H. Buhrman and R. D. Wolf. Complexity measures and decision tree complexity: A survey. *Theoretical Computer Science*, 288(1):21–43, 2002.

[21] X. Chen, X. Deng, and S. Teng. Settling the complexity of computing two-player Nash equilibria. *Journal of the ACM*, 56(3), 2009.

[22] R. W. Cottle and G. B. Dantzig. Complementary pivot theory of mathematical programming. *Linear Algebra and its Applications*, 1:103–125, 1986.

[23] S. Dasgupta. Coarse sample complexity bounds for active learning. In *19th Annual Conference on Neural Information Processing Systems*, 2005.

[24] C. Daskalakis, P. Goldberg, and C. Papadimitriou. Computing a Nash equilibrium is PPAD-complete. *SIAM Journal on Computing*, 39(1):195–259, 2009.

[25] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[26] X. Deng, T. Kameda, and C. Papadimitriou. How to learn an unknown environment I: The rectilinear case. *Journal of the ACM*, 45(2):215–245, 1998.

[27] X. Deng and C. Papadimitriou. Exploring an unknown graph. In *31st Annual Symposium on Foundations of Computer Science*, pages 355–361, 1990.

[28] M. Dyer, A. Frieze, and R. Kannan. A random polynomial time algorithm for approximating the volume of convex bodies. In *21st Annual ACM Symposium on Theory of Computing*, pages 375–381, 1989.

[29] D. Fudenberg and D. Levine. *The Theory of Learning in Games*. The MIT Press, 1998.

[30] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962.

[31] O. Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

[32] M. Grötschel, L. Lovász, and A. Schrijver. Geometric methods in combinatorial optimization. *Progress in Combinatorial Optimization*, pages 167–183, 1984.

[33] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer, 1988.

[34] J. Halpern. Beyond Nash equilibrium: Solution concepts for the 21st century. In *27th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–10, 2008.

[35] F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. The polygon exploration problem. *SIAM Journal on Computing*, 31(2):577–600, 2001.

[36] C. Icking, R. Klein, E. Langetepe, S. Schuierer, and I. Semrau. An optimal competitive strategy for walking in streets. *SIAM Journal on Computing*, 33(2):462–486, 2004.

[37] A. Indrayan. Elements of medical research. *Indian Journal of Medical Research*, 119:93–100, 2004.

[38] J. Jackson. An efficient membership-query algorithm for learning DNF with respect to the uniform distribution. *Journal of Computer and System Sciences*, 55(3):414–440, 1997.

[39] J. Kahn and J. H. Kim. Entropy and sorting. *Journal of Computer and System Sciences*, 51(3):390–399, 1995.

[40] J. Kahn and M. Saks. Balancing poset extensions. *Order*, 1(2):113–126, 1984.

[41] A. Kalai, A. Klivans, Y. Mansour, and R. Servedio. Agnostically learning halfspaces. *SIAM Journal on Computing*, 37(6):1777–1805, 2008.

[42] T. Kavitha. Linear time algorithms for abelian group isomorphism and related problems. *Journal of Computer and System Sciences*, 73(6):986–996, 2007.

[43] M. Kearns, M. Littman, and S. Singh. Graphical models for game theory. In *17th Conference in Uncertainty in Artificial Intelligence*, pages 253–260, 2001.

[44] M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[45] A. Klivans, R. O'Donnell, and R. Servedio. Learning intersections and thresholds of halfspaces. *Journal of Computer and System Sciences*, 68(4):808–840, 2004.

[46] A. Klivans and R. Servedio. Learning DNF in time $2^{\tilde{O}(n^{1/3})}$. *Journal of Computer and System Sciences*, 68(2):303–318, 2004.

[47] A. Klivans and A. Sherstov. Cryptographic hardness for learning intersections of halfspaces. *Journal of Computer and System Sciences*, 75(1):2–12, 2009.

[48] S. Kwek and K. Mehlhorn. Optimal search for rationals. *Information Processing Letters*, 86(1):23–26, 2003.

[49] B. Lehmann, D. Lehmann, and N. Nisan. Combinatorial auctions with decreasing marginal utilities. *Games and Economic Behavior*, 55:270–296, 2006.

[50] A. Lopez-Ortiz and S. Schuierer. Going home through an unknown street. In *4th Workshop on Algorithms and Data Structures*, pages 135–146, 1995.

[51] A. Lopez-Ortiz and S. Schuierer. Generalized streets revisited. In *4th Annual European Symposium on Algorithms*, pages 546–558, 1996.

[52] J. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[53] J. Mitchell. *Geometric Shortest Paths and Network Optimization*. Elsevier Science Publishers, 1998.

[54] D. Montgomery. *Design and Analysis of Experiments*. Wiley, 7th edition, 2008.

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

[56] C. Ng. Lower bounds for the stable marriage problem and its variants. In *30th Annual Symposium on Foundations of Computer Science*, pages 129–133, 1989.

[57] N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[58] P. Panaite and A. Pelc. Exploring unknown undirected graphs. In *9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 316–322, 1998.

[59] C. Papadimitriou. Efficient search for rationals. *Information Processing Letters*, 8(1):1–4, 1979.

[60] C. Papadimitriou. Algorithms, games, and the internet. In *33rd Annual ACM Symposium on Theory of Computing*, pages 749–753, 2001.

[61] C. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84(1):127–150, 1991.

[62] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6), 2009.

[63] A. Roth. Deferred acceptance algorithms: History, theory, practice, and open questions. *International Journal of Game Theory*, pages 537–569, 2008.

[64] A. Roth and M. Sotomayor. *Two-Sided Matching: A Study in Game-Theoretic Modeling and Analysis*. Cambridge University Press, 1992.

[65] M. Saks and A. Wigderson. Probabilistic boolean decision trees and the complexity of evaluating game trees. In *27th Annual Symposium on Foundations of Computer Science*, pages 29–38, 1986.

[66] U. Schöning. Graph Isomorphism is in the low hierarchy. In *4th Annual Symposium on Theoretical Aspects of Computer Science*, pages 114–124, 1987.

[67] B. Settles. Active learning literature survey. *Computer Sciences Technical Report 1648, University of Wisconsin-Madison*, 2009.

[68] L. Shapley. On balanced sets and cores. *Naval Research Logistics Quarterly*, 14:453–460, 1967.

[69] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[70] J. Vande Vate. Linear programming brings marital bliss. *Operations Research Letters*, 8(3):147–153, 1989.

[71] V. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, 1998.

[72] H. Young. Learning by trial and error. *Games and Economic Behavior*, 65(2):626–643, 2009.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de