Title	Interactive Search for One of the Top-k
Authors	Wang, Weicheng; Wong, Raymond Chi Wing; Xie, Min
Source	SIGMOD/PODS '21: Proceedings of the 2021 International Conference on Management of Data, New York, NY, United States : Association for Computing Machinery, 2021, p. 1920-1932
Version	Accepted Version
DOI	10.1145/3448016.3457322
Publisher	Association for Computing Machinery
Copyright	© 2021 Association for Computing Machinery. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in SIGMOD/PODS '21: Proceedings of the 2021 International Conference on Management of Data, https://doi.org/10.1145/3448016.3457322

Interactive Search for One of the Top-k

Weicheng Wang*, Raymond Chi-Wing Wong*, Min Xie+
*Hong Kong University of Science and Technology, +Shenzhen Institute of Computing Sciences, Shenzhen University
wwangby@connect.ust.hk,raywong@cse.ust.hk,xiemin@sics.ac.cn

ABSTRACT

When a large dataset is given, it is not desirable for a user to read all tuples one-by-one in the whole dataset to find satisfied tuples. The traditional top-k query finds the best k tuples (i.e., the top-k tuples) w.r.t. the user's preference. However, in practice, it is difficult for a user to specify his/her preference explicitly. We study how to enhance the top-k query with *user interaction*. Specifically, we ask a user several questions, each of which consists of two tuples and asks the user to indicate which one s/he prefers. Based on the feedback, the user's preference is learned implicitly and one of the top-k tuples w.r.t. the learned preference is returned. Here, instead of directly following the top-k query to return all the top-k tuples, since it requires heavy user effort during the interaction (e.g., answering many questions), we reduce the output size to strike for a trade-off between the user effort and the output size.

To achieve this, we present an algorithm *2D-PI* which asks an asymptotically optimal number of questions in a 2-dimensional space, and two algorithms *HD-PI* and *RH* with provable performance guarantee in a d-dimensional space ($d \geq 2$), where they focus on the number of questions asked and the execution time, respectively. Experiments were conducted on synthetic and real datasets, showing that our algorithms outperform existing ones by asking fewer questions within less time to return satisfied tuples.

CCS CONCEPTS

• **Information systems → Database query processing**.

KEYWORDS

top-k query; user interaction; data analytics

ACM Reference Format:
Weicheng Wang*, Raymond Chi-Wing Wong*, Min Xie+. 2021. Interactive Search for One of the Top-k. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457322

1 INTRODUCTION

Assisting users to find satisfied tuples in a large dataset is an important task in a variety of application domains [35, 39], including purchasing used cars, renting apartments and searching dating partners. For example, consider the scenario of purchasing used cars. Each car could be described by some attributes, e.g., price, horse power and used kilometers. Assume that Alice would like to buy a cheap used car with high horse power. In literature [22, 27, 36], Alice's preference is represented by a monotonic function, called a *utility function*. Based on the function, each car has a *utility* (i.e., the function score). It shows to what extent Alice favors the car, where a higher utility indicates the car is more favored by her.

Although various operators have been proposed for this scenario known as *multi-criteria decision-making*, they still have some weaknesses. Two representative operators are the top-k query [17, 26, 33] and the skyline query [7]. The first operator is the top-k query [17, 26, 33], which returns k tuples with the highest utilities, namely top-k tuples, w.r.t. the user's utility function. It assumes that a user knows his/her utility function precisely [22, 36]. However, in practice, it is very likely that Alice has difficulty in specifying that her utility function has weight 41.2% for price and 58.8% for horse power. Here, a higher weight indicates that the corresponding attribute is more important to Alice. If the weights given to the system differ slightly, the output can vary a lot. The second operator is the skyline query [7] which, instead of asking for an explicit utility function, considers all utility functions and returns tuples which have the highest utilities (i.e., top-1) w.r.t. at least one utility function. Unfortunately, the output size of the skyline query is uncontrollable and it often overwhelms users with excessive results [22, 23].

Motivated by the limitations of these operators, we propose a problem called *Interactive Search for One of the Top-k (IST)* which involves *user interaction* and asks a user as few easy questions as possible so that one of the top-k tuples is returned as the answer to the user where k is a positive integer. Problem IST has the following two advantages: (1) Unlike the top-k query, problem IST does not require a user to specify an explicit utility function; (2) Unlike the skyline query, problem IST has a controllable output size (i.e., only one tuple will be returned as the answer). Based on the user's feedback during interaction, the user's utility function is implicitly learned and one of the top-k tuples is guaranteed to be returned. To interact with the user, following [27, 36], *for each question, we present the user with two candidate tuples and ask the user which tuple s/he prefers.* This kind of interaction naturally appears in our daily life. For example, a car seller might show Alice two used cars and ask her which one she prefers. A matchmaker may present two candidates and ask Alice: which person would you like to date?

One characteristic of problem IST is that it involves a number of questions for asking a user. It is expected that the total number of questions asked should not be too many. In the literature of the marketing research [20, 28, 29], the maximum number of questions asked should be around 10. Besides, in real applications, involving excessive questions may not be good to the user. Consider the scenario about purchasing a used car where each car is unique in the market. If Alice looks for all (or some) of the top-k used cars,

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457322

she might spend days and weeks for selection. Due to the high-frequency of trading, it is possible that the car chosen by her has already been sold. Thus, a timely interaction and recommendation are crucial. Consider another scenario where Alice plans to rent an apartment for several months. Since this is a short-term need, it is not necessary for her to spend a lot of time cherry-picking. She could be frustrated due to the long selection process even if finally, several candidate (i.e., some of the top-k) apartments are returned.

To the best of our knowledge, we are the first to study problem IST. Some closely related problems are [14, 22, 27, 36] but they are different from ours. [14] also involves user interaction but it wants to find the ranking of *all* tuples (including the top-k tuples) by interacting with the user. Since the ranking of all tuples has to be determined, there are a lot of questions asked during user interaction. Besides, it has an assumption that all tuples are in the *general position* [30], which could not be applied in many cases. Roughly speaking, all tuples in the general position satisfies some geometry properties (e.g., no 3 points are co-linear), which is not realistic in real datasets. [22, 36] still involve user interaction but they aim at finding tuples in the answer such that a criterion called the *regret ratio*, evaluating how "regretful" a user is when seeing resulting tuples based on the concept of the top-k query, is minimized. It is observed that the returned tuples are satisfying one concept called "regret ratio", which is not quite intuitive as "one of the top-k" studied in our IST problem. Furthermore, a variant of [22, 36] could return the top-1 tuple as the answer. However, in our experiment, it needs more than 15 questions in many cases for asking a user, which is quite troublesome. As mentioned above, the maximum number of questions asked should be around 10. 15 is not quite acceptable to a user. The user study in our experimental study also verifies that people do not prefer being bothered with a lot of questions even the top-1 tuple is returned. Furthermore, [22] involves "fake" tuples (i.e., tuples not in the dataset) in the questions during user interaction. [27] involves user interaction and finds the user's utility function by interacting with the user. Similar to [14], since [27] has to find a precise utility function, there are a lot of redundant questions asked during user interaction, which may not be relevant to our desired answer (i.e., one of the top-k tuples).

Unfortunately, none of the existing algorithms could be adapted to solve problem IST satisfactorily. They generate a lot of questions to ask a user, which is quite troublesome to the user. In our experiments, when $k \geq 50$, most adapted algorithms [14, 22, 27, 36] require asking a user more than 20 questions, which are too many.

Our contributions are described as follows. Firstly, to the best of our knowledge, we are the first one to propose the problem of returning one of the user's top-k tuples by interacting with the user. We show a lower bound $\Omega(\log_2 \frac{n}{k})$ on the number of questions asked during the interaction. Secondly, we propose an algorithm *2D-PI* in a 2-dimensional space. It asks $O(\log_2 \lceil \frac{2n}{k+1} \rceil)$ questions to find one of the top-k tuples, which is asymptotically optimal in terms of the number of questions asked. Thirdly, we design an algorithm *HD-PI* with a provable guarantee in a d-dimensional space, which also performs well empirically. Fourthly, we propose an algorithm *RH* in a d-dimensional space. It guarantees a logarithmic number of questions in expectation, which is asymptotically optimal if the dimensionality is fixed. Fifthly, we conducted experiments to demonstrate the superiority of our algorithms. The results show that our algorithms are able to return one of the user's top-k tuples by requiring nearly half the number of questions compared with the existing algorithms when k is of medium size (e.g., $k \geq 50$).

In the following, we start by discussing the related work in Section 2. The formal definition of our problem is illustrated in Section 3. In Section 4, we propose an asymptotically optimal algorithm *2D-PI* in a 2-dimensional space. In Section 5, we propose two algorithms *HD-PI* and *RH* with provable guarantees on the number of questions asked in a d-dimensional space. Experiments are shown in Section 6 and finally, Section 7 concludes this paper.

2 RELATED WORK

Various queries were proposed to assist the multi-criteria decision making. Based on whether user interaction is involved, they can be classified into two categories: the *preference-based queries* and the *interactive queries*.

In addition to the top-k query and the skyline query described in Section 1, there are two other queries for preference-based queries, namely the similarity query [4, 32] and the regret minimizing query [10, 23, 25]. The similarity query [4, 32] finds tuples which are close to a given query tuple w.r.t. a given distance function. However, it relies on an assumption that the query tuple and the distance function are known in advance [5]. In practice, a user does not always know the query tuple or the distance function. Even if the query tuple or the distance function is known, the user needs to spend additional effort specifying them. The regret minimizing query [10, 23, 25], another type of preference-based query, which avoids this problem, defines a criterion called *regret ratio* which evaluates returned tuples and represents how regretful a user is when s/he sees the returned tuples instead of the whole dataset, and it aims to find tuples which minimize the regret ratio. However, it is hard to achieve a small output size and a small regret ratio simultaneously. When a small output size is fixed, the regret ratio is typically large [10, 36]. For further details, see [16, 37] and references therein.

To overcome the deficiencies of the preference-based queries, some existing studies [2, 3, 5, 6, 18, 22, 31, 36, 40] involve user interaction. [2, 3, 18] proposed the interactive skyline query, which tries to reduce the number of skyline tuples in the answer by interacting with the user. However, it only learns the user's preference on the values of attributes (e.g., values *red*, *yellow* and *blue* on the color attribute). Even if the preference on all values is obtained, the number of skyline tuples could still be arbitrarily large [23].

[5, 6, 31] proposed the interactive similarity query which learns the exact query tuple and the distance function with the help of user interaction. However, during the interaction, it requires a user to assign *relevance scores* for hundreds or thousands of tuples to learn how close the tuples are to the query tuple. From the user's perspective, requiring the user to give accurate scores for a lot of times is too demanding in practice. Besides, it is challenging to determine an initial query tuple guaranteeing a good performance because the initial query tuple affects the final output significantly.

[22] proposed the interactive regret minimizing query, which targets to reduce the regret ratio while maintaining a small output size by interacting with the user. It asks a user questions, each of which consists of several tuples and asks the user to tell which one

s/he prefers. However, it displays *fake tuples* during the interaction, which are artificially constructed (not selected from the dataset). This might produce unrealistic tuples (e.g., a car with 10 dollars and 50000 horse power) and the user can be disappointed if the displayed tuples with which s/he is satisfied do not exist [36]. To overcome the defect, [36] proposed algorithms *UH-Simplex* and *UH-Random*, which utilize *real tuples* (selected from the dataset) for the interactive regret minimization. However, they require heavy user effort: answering many questions and waiting a long time for algorithm processing. As shown in Section 6, our algorithms need fewer questions and less time compared with them (e.g., half the number of questions asked and execution time). Recently, [40] improved the algorithms of [36] and proposed *Sorting-Simplex* and *Sorting-Random*, which ask the user to give an order on the displayed tuples. However, this does not reduce the user effort essentially, since giving an order among tuples is equivalent to picking the favorite tuple several times. Note that both [36, 40] focus on finding the user's (close to) favorite tuple. To some extent, their problems [36, 40] can be seen as a special case of our problem when $k = 1$.

There are alternative approaches [15, 27] which focus on learning the user preference with the help of user interaction. [27] proposed algorithm *Preference-Learning* which approximates the user preference by pairwise comparison. Nevertheless, it aims at learning the preference rather than returning tuples, and thus it might ask the user unnecessary questions [36]. For example, if Alice prefers car p_1 to both p_2 and p_3, her preference between p_2 and p_3 is less interesting in our case, but this additional comparison might be useful in [27]. In addition, [15] learns the user preference on tuples with *undetermined* attributes, where there is no universal preference defined on values of those attributes for all users. For example, consider an attribute *color* containing 3 values: *red*, *green* and *blue*. The preferences on *red*, *green* and *blue* can vary dramatically from different users. In our problem, all attributes are determined.

In the literature of machine learning, our problem is related to the problem of *learning to rank* [12, 14, 19, 21], which learns the ranking of tuples by pairwise comparison. However, most of the existing methods [12, 19, 21] only consider the relations between tuples (where a relation means that a tuple is preferable to another tuple) and do not utilize their inter-relation (where attribute "price" is an example of an inter-relation showing that \$200 is better than \$500 since \$200 is cheaper), and thus, require more feedback from the user [36]. Algorithm *Active-Ranking* [14] considers the inter-relation between tuples to learn the ranking by interacting with the user. However, it assumes that all tuples are in the *general position* [30], which could not be applied in many cases. Besides, it focuses on deriving the order for all pairs of tuples, which requires asking unnecessary questions due to the similar reason stated for [27].

Our work focuses on returning one of the top-k tuples by interacting with the user on real tuples. This avoids the weaknesses of existing studies: (1) We do not require a user to provide an exact utility function (required in the top-k query) or a distance function (required in the similarity query). (2) We return one of the top-k tuples (but the skyline query has an uncontrollable output size) (3) We guarantee that the returned tuple must be among the top-k tuples (but the regret minimizing query does not have a clear and intuitive interpretation of the quality of the answer). (4) We only use real tuples during the interaction (unlike [22] which utilizes fake tuples). (5) We only involve a few easy questions since we return one of the top-k tuples. Firstly, existing studies like [12, 19, 21] ask a lot of questions since they require to learn a full ranking. Secondly, [12, 19, 21] do not utilize the inter-relation between tuples and thus, involve some unnecessary interaction. Thirdly, [40] requires a user to sort tuples and [5, 31] require a user to assign concrete scores. But, we just require a user to pick a favorite tuple between two candidates for each question, which is easier to handle.

Table 1 shows a comparison on the number of questions asked between our algorithms and several existing algorithms which can be adapted to our problem. Their adaptations are detailed in Section 6. It can be seen that our algorithm *HD-PI* is independent of the dimensionality in the optimal case and *RH* is asymptotically optimal in expectation when the dimensionality is fixed. Note that although algorithm *Active-Ranking* has an expected logarithmic bound, it is under the condition that all tuples are in the general position, but the bound of *RH* works for arbitrary cases.

3 PROBLEM DEFINITION

The input of our problem is a set D containing n tuples specified by d attributes. In the rest of this paper, we regard each tuple as a point in a d-dimensional space and use the words "tuple" and "point" interchangeably. For each point p, its i-th dimensional value is denoted by $p[i]$, where $i \in [1, d]$. Without loss of generality, following [36, 38], we assume that each dimension is normalized to $(0, 1]$ and a larger value is more favored by users in each dimension. Consider Table 2 as an example. It contains 5 points in a 2-dimensional space, where each dimension is normalized.

Following [1, 36], the user preference is modeled by a linear function $f(p) = \sum_{i=1}^{d} u[i]p[i]$, called the *utility function*, denoted by $f(p) = u \cdot p$ for simplicity, which is a mapping $f : \mathbb{R}_+^d \rightarrow \mathbb{R}_+$, where u is a d-dimensional non-negative vector, called the *utility vector*, and $f(p)$ is the *utility* of p w.r.t. f. For each $u[i]$, where $i \in [1, d]$, it represents to what extent the user cares about the i-th attribute. A larger value means that this attribute is more important to the user. In the rest of this paper, we use "utility vector" to refer to the user preference and call the domain of u the *utility space*. Given a utility vector u, the utility of each point can be computed and then the k points with the largest utility (i.e., the top-k points) can be found. Since the ranking of points remains unchanged with different scaled utility vectors [22, 36], for the ease of presentation, we assume that $\sum_{i=1}^{d} u[i] = 1$. Then, the utility space can be viewed as a $(d-1)$-dimensional polyhedron. For example, in a 2-dimensional space, the utility space is a line segment: $u[1] + u[2] = 1$ with $u[1], u[2] > 0$.

Example 3.1. Let $f(p) = 0.4p[1] + 0.6p[2]$ (i.e., $u = (0.4, 0.6)$). Consider Table 2. The utility of p_2 w.r.t. f is $f(p_2) = 0.4 \times 0.3 + 0.6 \times 0.7 = 0.54$. The utilities of the other points can be computed similarly. Assume $k = 2$. Points p_1 and p_3 with the highest utilities are the top-k points.

Given a point set D, our goal is to return a point, which is one of the user's top-k points, by interacting with the user for rounds. In each round, the system chooses a pair of points as a question presented to the user and asks the user to indicate which one s/he prefers. Based on the feedback, the user's preference is learned

Algorithm (d dimension)	Worst Case	Optimal Case	Expected Case
UH-Random [36]	$O(n)$	Unknown	Unknown
UH-Simplex [36]	$O(n)$	Unknown	$O(deg_{max}\sqrt[d]{n})$
Active-Ranking [14]	$O(n^2)$	$O(d\log n)$	$O(cd\log n)$, where $c > 1$
Preference-Learning [27]	$O(n^2)$	Unknown	Unknown
RH	$O(n^2)$	$O(d\log n)$	$O(cd\log n)$, where $c > 1$
HD-PI	$O(n)$	$O(\log n)$	Unknown

Table 1: Algorithm comparison ($k \geq 1$)

p	$p[1]$	$p[2]$	$f(p)$	rank
p_1	0	1	0.6	2
p_2	0.3	0.7	0.54	3
p_3	0.5	0.8	0.68	1
p_4	0.7	0.4	0.52	4
p_5	1	0	0.4	5

Table 2: Dataset($u = (0.4, 0.6)$)

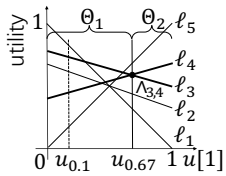


Figure 1: 2D Partitions

implicitly. When sufficient information has been collected, the interaction stops and the system returns the desired point to the user. Formally, we are interested in the following problem.

Problem 1. ***(Interactive Search for One of the Top-k (IST))*** *Given a point set D, we are to ask a user as few questions as possible to identify a point p in D, which is one of the user's top-k points.*

Theorem 3.2. *There exists a dataset of n points such that for any algorithm, it needs to ask a user $\Omega(\log_2 \frac{n}{k})$ questions in order to determine a point p, which is one of the user's top-k points.*

Proof Sketch. Consider a dataset D, where $\forall p \in D$, there are $k-1$ points $q \in D \setminus \{p\}$ such that $\forall i \in [1, d], p[i] = q[i]$. We show that any algorithm needs to ask $\Omega(\log_2 \frac{n}{k})$ questions to identify one of the user's top-k points on this dataset. □

4 2D ALGORITHM

In this section, we focus on 2-dimensional IST. We propose an asymptotically optimal algorithm, called *2D-PI*, which is able to return the desired point by asking a logarithmic number of questions.

4.1 Preliminary

Recall that in a 2-dimensional space, $u[1] + u[2] = 1$. The utility of each point p is written as $f(p) = u[1]p[1] + (1 - u[1])p[2]$ (i.e., it suffices to consider $u[1]$ only). From a geometric perspective, if we plot the utility of p as a function of $u[1]$, p can be mapped to a line segment ℓ: $f(p) = (p[1] - p[2])u[1] + p[2]$, whose slope is $(p[1] - p[2])$ and intercept is $p[2]$, and the utility space can be viewed as an interval, i.e., $u[1] \in [0, 1]$. For example, as shown in Figure 1, p_2 in Table 2 can be mapped or transformed to a line segment $\ell_2 : f(p_2) = -0.4u[1] + 0.7$. Similarly, other points p_i in Table 2 are also mapped to line segments ℓ_i shown in Figure 1.

By transforming points into line segments, the ranking of points w.r.t. a utility vector u can be easily visualized. Specifically, we build a vertical line t for each u, called the *utility line*, passing through $(u[1], 0)$ in the geometric space. The ranking of points w.r.t. u is the same as the order of the intersections (from top to bottom) between t and the transformed line segments. For example, in Figure 1, the ranking of points w.r.t. $u_{0.1} = (0.1, 0.9)$ is $< p_1, p_3, p_2, p_4, p_5 >$ and the utility line of $u_{0.1}$ also intersects ℓ_1, ℓ_3, ℓ_2, ℓ_4 and ℓ_5 in order.

Intuitively, our algorithm *2D-PI* consists of two steps: (1) utility space partitioning and (2) user interaction. We first divide the utility space $[0, 1]$ into a number of disjoint partitions, where the x-th partition, denoted by Θ_x, is an interval $[l_x, r_x]$ with $l_x = r_{x-1}$. Each partition Θ_x is associated with a point q_x which is among the top-k points w.r.t. any utility vector in Θ_x. For example in Figure 1, let $k = 2$. The utility space can be divided into two partitions $\Theta_1 = [0, 0.67]$ and $\Theta_2 = [0.67, 1]$, where p_3 is among the top-2 points w.r.t. any utility vector in Θ_1; similarly for p_4 in Θ_2. Then, we interact with the user by asking questions to locate the partition containing the user's utility vector and return the associated point.

In the following, we first present the method for utility space partitioning in Section 4.2, and then discuss the strategy of interacting with the user to quickly locate the target partition in Section 4.3.

4.2 Utility Space Partitioning

Since the user's utility vector can be located more quickly with *fewer* partitions, we divide the utility space into the *least* number of partitions by *plane sweeping* in the geometric space. Specifically, we sweep the utility line t from left to right by varying its $u[1]$-value from 0 to 1. During the process, we maintain two data structures: (1) a queue Q of size n, which stores all the points $p_i \in D$ based on the order (from top to bottom) of the interactions between t and l_i; (2) a min-heap $\mathcal{H}$, which records the intersection $\wedge_{i,j}$ between ℓ_i and ℓ_j of each pair of neighboring points, namely p_i and p_j, in Q, by using the distance between $\wedge_{i,j}$ and t as the key of the min-heap, provided that $\wedge_{i,j}$ is on the right of t and $\wedge_{i,j}[1] \leq 1$ (call an intersection satisfying this condition as a *valid* intersection). Furthermore, each point in D might be associated with a label T_x if it could be one of the top-k points w.r.t. the utility vector $(l_x, 1 - l_x)$.

At the beginning, the $u[1]$-value of t is set to 0. Start the first partition Θ_1 with $l_1 = 0$. We insert into Q all the points $p_i \in D$ based on the order of the intersections between t and ℓ_i and label the first k points with T_1. $\mathcal{H}$ is initialized with the valid intersections of all the neighboring pairs in Q. During sweeping, we stop t at any intersection popped from $\mathcal{H}$, updating the data structures and constructing partitions. Suppose that now, partition Θ_x is constructed. We pop the next intersection $\wedge_{i,j}$ from $\mathcal{H}$, which is the intersection between lines ℓ_i and ℓ_j, and move t to the right until it hits $\wedge_{i,j}$. Then, we swap p_i and p_j in Q, and insert into $\mathcal{H}$ two new intersections (i.e., p_i and its new neighbour in Q and p_j and its new neighbour in Q) if they are valid. If p_i and p_j are the k-th and $(k+1)$-th point in Q before swapping, we delete the label of p_i and label p_j with T_{x+1}. If p_i is the last point with label T_x deleted, we end partition Θ_x by setting $\Theta_x = [l_x, r_x]$ and $q_x = p_i$ such that l_x (resp. r_x) is the $u[1]$-value of t where we start (resp. end) partition Θ_x. Meanwhile, the construction of the next partition Θ_{x+1} is started with $l_{x+1} = r_x$. It can be easily verified that right now the top-k points in Q are already labeled with T_{x+1}. The algorithm continues until $\mathcal{H}$ is empty. The pseudocode is shown in Algorithm 1.

Example 4.1. Consider Figure 1 and let $k = 2$. Initially, the $u[1]$-value of t is 0. We begin Θ_1 with $l_1 = 0$, set $Q =< p_1, p_3, p_2, p_4, p_5 >$ with p_1 and p_3 labeled by T_1, and initialize $\mathcal{H} = \{\wedge_{1,3}, \wedge_{2,4}, \wedge_{4,5}\}$, where $\wedge_{2,3}$ is not included because it is not valid. Then, the closest intersection $\wedge_{1,3}$ to t is popped. We swap p_1 and p_3 in Q and insert into $\mathcal{H}$ a new intersection $\wedge_{1,2}$ (p_1 with its new neighbor p_2). Since p_3 has no new neighbor, no new intersection related to p_3 is inserted

Algorithm 1: Utility Space Partitioning

Input: A point set D
Output: $\Theta = \{\Theta_1, \Theta_2, ..., \Theta_m\}$, $\mathcal{E} = \{q_1, q_2, ..., q_m\}$

Initialize t with its $u[1]$-value as 0, $x \leftarrow 1$
$Q \leftarrow D$ based on the order of intersections between t and ℓ_i
Label the first k points in Q with T_x
Initialize $\mathcal{H}$ with valid intersections of all pairs in Q
while $|\mathcal{H}| > 0$ **do**
  Pop $\wedge_{i,j}$ from $\mathcal{H}$, move t to hit $\wedge_{i,j}$ and update $Q, \mathcal{H}$
  if *p_i and p_j are the k-th and $(k+1)$-th point in Q* **then**
    Delete the label of p_i and label p_j with T_{x+1}
    if *p_i is the last point with label T_x deleted* **then**
      $\Theta_x = [l_x, r_x]$, $q_x \leftarrow p_i$, $x \leftarrow x + 1$
return $\Theta = \{\Theta_1, \Theta_2, ..., \Theta_m\}$, $\mathcal{E} = \{q_1, q_2, ..., q_m\}$

into $\mathcal{H}$. Besides, the label of p_1 and p_3 are not updated since p_1 and p_3 are not the k-th and $(k+1)$-th point in Q before swapping.

Theorem 4.2. *Algorithm 1 runs in $O(n^2 \log n)$ time.*

Proof. We need to process the intersections of line segments w.r.t. all pairs of points in D. Since $|D| = n$, there are $O(n^2)$ intersections. At each intersection, we update $\mathcal{H}$ and Q in $O(\log n)$ and $O(1)$ time, respectively, and modify the labels in $O(1)$ time. Therefore, the total time complexity of Algorithm 1 is $O(n^2 \log n)$. □

Lemma 4.3. *Algorithm 1 divides the utility space into the least number of partitions.*

Proof Sketch. Use $\Theta'_i = [l'_i, r'_i]$ and $\Theta_i = [l_i, r_i]$ to denote the i-th partition of the optimal case and the i-th partition obtained by our algorithm. We show that $r'_i \leq r_i$, i.e., Θ_i always ends not "earlier" than Θ'_i. This means that the number of partitions obtained by our algorithm will not be more than that of the optimal case. □

Note that [1, 8] propose algorithms which achieve the similar purpose. However, [1] only presents an approximate algorithm, while our algorithm returns an exact solution. Besides, it is not easy to process the real implementation of [8] due to its complicated data structure with its theoretical result. To the best of our knowledge, there is no real implementation of [8] in the literature.

4.3 User Interaction

After the utility space is partitioned, we interact with the user to locate the partition containing the user's utility vector. Consider two points p_i and p_j. If the intersection $\wedge_{i,j}$ of their transformed line segments ℓ_i and ℓ_j exists and satisfies $0 < \wedge_{i,j}[1] < 1$, their ranking will change once when t sweeps from left to right, i.e., varies its $u[1]$-value from 0 to 1. If point p_i ranks higher than point p_j w.r.t. a utility vector u (i.e., $u \cdot p_i > u \cdot p_j$), where $u[1] < \wedge_{i,j}[1]$, p_i must rank lower than p_j w.r.t. any utility vector u such that $u[1] > \wedge_{i,j}[1]$. Then, if a user prefers p_i to p_j, the $u[1]$-value of the user's utility vector must be smaller than $\wedge_{i,j}[1]$. Otherwise, it should be larger than $\wedge_{i,j}[1]$. Based on this idea, we locate the partition containing the user's utility vector by *binary search*.

Algorithm 2: User Interaction

Input: $\Theta = \{\Theta_1, \Theta_2, ..., \Theta_m\}$, $\mathcal{E} = \{q_1, q_2, ..., q_m\}$
Output: A point q_x, which is one of the user's top-k points

$C \leftarrow < \Theta_1, \Theta_2, ..., \Theta_m >$, $left \leftarrow 1$, $right \leftarrow m$
while $|C| > 1$ **do**
  $x \leftarrow left - 1 + \lfloor \frac{right - left + 1}{2} \rfloor$
  Find points p_i and p_j, where the intersection $\wedge_{i,j}$ of ℓ_i and ℓ_j exists and satisfies $\wedge_{i,j}[1] = r_x$.
  Display p_i and p_j to the user
  if *the user prefers p_i to p_j* **then**
    $right \leftarrow x$
  else
    $left \leftarrow x + 1$
  $C \leftarrow < \Theta_{left}, ..., \Theta_{right} >$
return the associated point q_x of the only partition left in C

We interact with the user for rounds, while maintaining a list C to store the candidate partitions in order, initialized to be the partitions obtained from Algorithm 1. In each round, we prune half of partitions in C by asking the user a question. Specifically, we find the median partition $\Theta_x = [l_x, r_x]$ in C, and present the user with two points p_i and p_j, where the intersection $\wedge_{i,j}$ of ℓ_i and ℓ_j exists and satisfies $\wedge_{i,j}[1] = r_x$. Without loss of generality, assume that p_i ranks higher than p_j w.r.t. a utility vector u with $u[1] < r_x$. If the user prefers p_i to p_j, C is updated to be the first half of C. Otherwise, the remaining half of C is kept. The process continues until there is only one partition Θ_x left in C and the associated point q_x is returned. The pseudocode is shown in Algorithm 2.

Example 4.4. Continue Example 4.1. Initially, $C = < \Theta_1, \Theta_2 >$ and Θ_1 is the median partition. We present the user with points p_3 and p_4, since intersection $\wedge_{3,4}$ exists and satisfies $\wedge_{3,4}[1] = r_1$. Suppose the user prefers p_3 to p_4, C is updated to $< \Theta_1 >$. Because there is only one partition left in C, the associated point q_1 is returned.

Theorem 4.5. *Algorithm 2D-PI solves 2-dimensional IST by interacting with the user for $O(\log_2 \lceil \frac{2n}{k+1} \rceil)$ rounds.*

Proof Sketch. We show that there are at most $\lceil \frac{2n}{k+1} \rceil$ partitions if we divide the utility space by Algorithm 1. Since the number of candidate partitions is reduced by half in each round, we can locate the user's utility vector after $O(\log_2 \lceil \frac{2n}{k+1} \rceil)$ rounds. □

Corollary 4.6. *Algorithm 2D-PI is asymptotically optimal in terms of the number of questions asked for 2-dimensional IST.*

5 HIGH DIMENSIONAL ALGORITHM

In this section, we consider high dimensional IST. We first show some preliminaries in Section 5.1 and then develop two algorithms, namely *HD-PI* and *RH*, in Section 5.2 and Section 5.3, respectively. *HD-PI* enjoys good empirical performance. *RH* asks $O(cd \log_2 n)$ questions in expectation ($c \geq 1$ is a constant), which is asymptotically optimal w.r.t. the number of questions asked if d is fixed.

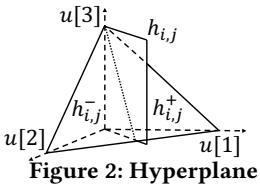


Figure 2: Hyperplane

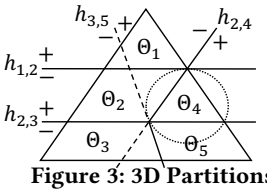


Figure 3: 3D Partitions

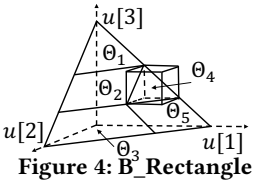


Figure 4: B_Rectangle

$h_{i,j}$	in $h^+_{i,j}$	in $h^-_{i,j}$	intersect	even score
$h_{1,2}$	Θ_1	$\Theta_2, \Theta_3, \Theta_4, \Theta_5$		1
$h_{2,3}$	$\Theta_1, \Theta_2, \Theta_4$	Θ_3, Θ_5		2
$h_{2,4}$	Θ_4, Θ_5	Θ_1, Θ_2	Θ_3	1.9
$h_{3,5}$	Θ_4, Θ_5	Θ_3	Θ_1, Θ_2	0.8

Table 3: The inital list Γ ($\beta = 0.1$)

5.1 Preliminaries

Recall that in a d-dimensional space $\mathbb{R}^d$, the utility space is a $(d-1)$-dimensional polyhedron. For example, as shown in Figure 2, the utility space is a triangular region when $d = 3$. For each pair of points $p_i, p_j \in D$, where $i, j \in [1, n]$, we can construct a hyperplane, denote by $h_{i,j}$, which passes through the origin with its unit normal in the same direction as $p_i - p_j$. Hyperplane $h_{i,j}$ divides the space $\mathbb{R}^d$ into two halves, called *halfspaces* [11]. The halfspace above (resp. below) $h_{i,j}$, denoted by $h^+_{i,j}$ (resp. $h^-_{i,j}$), contains all the utility vectors u such that $u \cdot (p_i - p_j) > 0$ (resp. $u \cdot (p_i - p_j) < 0$), i.e., p_i ranks higher (resp. lower) than p_j w.r.t. u [36].

In geometry, a polyhedron $\mathcal{P}$ is the intersection of a set of halfspaces [11] and a point p in $\mathcal{P}$ is said to be a *vertex* of $\mathcal{P}$ if p is a corner point of $\mathcal{P}$. We denote the set of all vertices of $\mathcal{P}$ by $\mathcal{V}$. Later, polyhedrons which possibly contain the user's utility vector are maintained. Every time a user provides feedback, a halfspace is built and is used to update the polyhedrons accordingly.

There are three kinds of relationship between a polyhedron $\mathcal{P}$ and a hyperplane $h_{i,j}$: (1) $\mathcal{P}$ is in $h^+_{i,j}$ (i.e., $\mathcal{P} \subseteq h^+_{i,j}$); (2) $\mathcal{P}$ is in $h^-_{i,j}$ (i.e., $\mathcal{P} \subseteq h^-_{i,j}$); (3) $\mathcal{P}$ intersects $h_{i,j}$. For instance, in Figure 3, polyhedron Θ_3 is in $h^-_{2,3}$ and intersects $h_{2,4}$. To determine the relationship between $h_{i,j}$ and $\mathcal{P}$, we use the vertices $\mathcal{V}$ of $\mathcal{P}$. If there exist $v_1, v_2 \in \mathcal{V}$ such that $v_1 \in h^+_{i,j}$ and $v_2 \in h^-_{i,j}$, $\mathcal{P}$ intersects with $h_{i,j}$. Otherwise, $\mathcal{P}$ lies in either $h^+_{i,j}$ or $h^-_{i,j}$.

However, since we may need to go through each vertex $v \in \mathcal{V}$, checking the relationship between $h_{i,j}$ and $\mathcal{P}$ requires $O(|\mathcal{V}|)$ time, which could be slow if $|\mathcal{V}|$ is large in a high dimensional space. Thus, we present two sufficient conditions to identify $\mathcal{P}$ in either $h^+_{i,j}$ or $h^-_{i,j}$ in $O(1)$ and $O(2^d)$ time, respectively.

We first introduce two concepts: the *bounding ball* and *bounding rectangle* of a polyhedron $\mathcal{P}$, which are a ball and a rectangle bounding $\mathcal{P}$. Examples are shown in Figures 3 and 4. For the bounding ball of $\mathcal{P}$, denoted by $\mathbb{B}(\mathcal{P})$, its center and its radius are defined to be $\mathbb{B}_c = \frac{\sum_{v \in \mathcal{V}} v}{|\mathcal{V}|}$ and $\mathbb{B}_r = \max_{v \in \mathcal{V}} \text{dist}(v, \mathbb{B}_c)$, respectively, where $\text{dist}(v, \mathbb{B}_c)$ denotes the Euclidean distance between v and $\mathbb{B}_c$. Then, $\mathbb{B}(\mathcal{P}) = \{p \in \mathbb{R}^d | \text{dist}(p, \mathbb{B}_c) \leq \mathbb{B}_r\}$. As for the bounding rectangle, we compute the maximum and minimum values of each dimension to be $max_i = \max_{v \in \mathcal{V}} v[i]$ and $min_i = \min_{v \in \mathcal{V}} v[i]$ ($i \in [1, d]$). Then, the bounding rectangle of $\mathcal{P}$, denoted by $\mathbb{R}(\mathcal{P})$, is defined to be $\mathbb{R}(\mathcal{P}) = \{p \in \mathbb{R}^d | \, p[i] \in [min_i, max_i], \forall i \in [1, d]\}$.

Lemma 5.1. *Given a polyhedron $\mathcal{P}$, if $\mathbb{B}(\mathcal{P}) \subseteq h^+_{i,j}$ or $\mathbb{R}(\mathcal{P}) \subseteq h^+_{i,j}$, we conclude that $\mathcal{P} \subseteq h^+_{i,j}$; similarly for $h^-_{i,j}$.*

Determining whether $\mathbb{B}(\mathcal{P}) \subseteq h^+_{i,j}$ can be done in $O(1)$ time by checking if $\mathbb{B}_c \in h^+_{i,j}$ and the distance from $\mathbb{B}_c$ to $h_{i,j}$ is larger than $\mathbb{B}_r$, and determining whether $\mathbb{R}(\mathcal{P}) \subseteq h^+_{i,j}$ can be done in $O(2^d)$ time by checking if each vertex of $\mathbb{R}(\mathcal{P})$ is in $h^+_{i,j}$; similarly for $h^-_{i,j}$. Although the bounding rectangle has a higher time complexity, it bounds $\mathcal{P}$ more "tightly". The performances of these two bounding methods are compared experimentally in Section 6.1.

We are ready to describe our d-dimensional algorithms. Intuitively, our algorithms follow the *interactive framework* from [36]: we interact with a user for rounds until we can find a point which is one of the user's top-k points. In each round,

- (**Point selection**) We select two points presented to the user and ask the user which one s/he prefers.
- (**Information Maintenance**) Based on the feedback, we update the maintained information.
- (**Stopping condition**) We check the stopping condition. If it is satisfied, we terminate and return the result.

In the following, we present the algorithms for high dimensional IST by elaborating the strategies of each component.

5.2 HD-PI

In this section, we present algorithm *HD-PI*, which is a high dimensional extension of *2D-PI* (shown in Section 4). It performs the best in our empirical study w.r.t. the number of questions asked.

5.2.1 Information Maintenance. We maintain two data structures: (1) a set C, which contains disjoint polyhedrons, called partitions, in the utility space, where the user's utility vector might be located. In particular, the union of partitions in C is defined to be the *utility range*, denoted by R, i.e., $R = \cup_{\Theta \in C} \Theta$; (2) a list Γ, recording the relationship between hyperplanes and partitions in C.

We initialize C by dividing the whole utility space into a number of partitions such that each partition Θ is associated with a point, which is among the top-k points w.r.t. any utility vector in Θ. One might want to divide the utility space into the least number of partitions, so that we can quickly locate the user's utility vector as explained in Section 4. However, as proved in Theorem 5.2, dividing the utility space into the least number of partitions is NP-hard.

Theorem 5.2. *Dividing the utility space into the least number of partitions such that each partition Θ is associated with a point, which is among the top-k points w.r.t. any utility vector in Θ, is NP-hard.*

To balance the time cost and the number of partitions, we propose a practical method to construct C. Specifically, we first find the set V of all *convex points* [11] in D based on some existing algorithms [11], which has the highest utilities (i.e., top-1) w.r.t. at least one utility vector in the utility space. In practice, we can use a sampling strategy for approximating V. Then, for each point $p_i \in V$, we create a partition $\Theta_i = \{u \in \mathbb{R}^d_+ | u \in h^+_{i,j}, \forall p_j \in V \setminus \{p_i\} \text{ and } \sum_{i=1}^d u[i] = 1\}$ and add it to C. It can be verified that p_i is the top-1 point w.r.t. any $u \in \Theta_i$, i.e., p_i is the associated point of Θ_i. Note that there are $O(n)$ partitions in C and initially $R = \cup_{\Theta \in C} \Theta = \{u \in \mathbb{R}^d_+ | \sum_{i=1}^d u[i] = 1\}$.

$h_{i,j}$	in $h^+_{i,j}$	in $h^+_{i,j}$	intersect	even score
$h_{2,4}$	Θ_5		Θ_3	-0.1
$h_{3,5}$	Θ_5	Θ_3		1

Table 4: List Γ given that p_3 is more preferred than p_2 ($\beta = 0.1$)

In list Γ, each row corresponds to a hyperplane $h_{i,j}$, constructed by points $p_i, p_j \in V$. It records the relationship between $h_{i,j}$ and the partitions in C (e.g., in $h^+_{i,j}$, in $h^-_{i,j}$ or intersect $h_{i,j}$). In addition, we store an *even score* for each $h_{i,j}$ in Γ to measure how evenly $h_{i,j}$ divides the partitions in C. The even score will be used later for point selection. We postpone its formal definition to the next section. Table 3 is the example of Γ for hyperplanes in Figure 3.

When a user answers more questions, more information about the user's utility vector is known and the data structures C and Γ are updated accordingly. Assume that a user prefers p_i to p_j in a question. We learn that the user's utility vector is in $h^+_{i,j}$. For each Θ in C, (1) if $\Theta \subseteq h^-_{i,j}$, Θ is removed from C since it cannot contain the user's utility vector according to the definition of $h_{i,j}$; and (2) if Θ intersects $h_{i,j}$, Θ is updated to be $\Theta \cap h^+_{i,j}$. Then, list Γ is updated based on the updated C: (1) partitions removed from C are deleted in each row of Γ; (2) if there is a hyperplane $h_{i',j'}$ in Γ such that R is in $h^+_{i',j'}$ (resp. $h^-_{i',j'}$), the row of $h_{i',j'}$ is removed from Γ since the user's utility vector is in R, $p_{i'}$ must rank higher (resp. lower) than $p_{j'}$ and thus, $h^+_{i',j'}$ and $h^-_{i',j'}$ cannot be used to update C later; and (3) the even scores of the remaining rows in Γ are recalculated.

Example 5.3. In Figure 3, initially, $C = \{\Theta_1, \Theta_2, \Theta_3, \Theta_4, \Theta_5\}$ and Γ is shown in Table 3. Suppose that a user prefers p_3 to p_2. We learn that his/her utility vector is in $h^-_{2,3}$ (or $h^+_{3,2}$). Partitions Θ_1, Θ_2 and Θ_4 in C are removed since they are in $h^+_{2,3}$ (or $h^-_{3,2}$) and the other partitions in C do not change since none of them intersect $h_{2,3}$. Next, partitions Θ_1, Θ_2 and Θ_4 are deleted in each row of list Γ. Finally, the rows of hyperplanes $h_{1,2}$ and $h_{2,3}$ are deleted from Γ since R is in $h^-_{1,2}$ and $h^-_{2,3}$, and the even scores of the remaining rows in Γ are renewed. The updated Γ is shown in Table 4.

Note that [1] presents an algorithm which focuses on the similar purpose of dividing the utility space into the least number of partitions. However, [1] only returns an approximate solution with a high time complexity. In detail, it involves two steps, where (1) the first step needs to find all the *k-sets* (note that since there are $O(n^{d-\epsilon})$ k-sets [30], where $\epsilon > 0$ is a small constant, the time complexity of this step is at least $O(n^{d-\epsilon})$) and (2) the second step is an NP-hard problem. But, our method only requires $O(n^{\lfloor d/2 \rfloor})$ time [9] to obtain a small number of partitions.

5.2.2 Point Selection. We aim to filter as many partitions in C as possible in each round so that the user's utility vector can be quickly located. In each round, if a user prefers p_i to p_j, we remove partitions in $h^-_{i,j}$. Otherwise, partitions in $h^+_{i,j}$ are removed. Thus, if a hyperplane evenly divides C into two halves, half of the partitions in C can be removed no matter what answer the user provides. Following this idea, in each round, we find two points p_i and p_j and display them to the user, where $h_{i,j}$ in Γ divides the partitions in C the *most evenly*. To evaluate the "evenness", we define the *even score* for each hyperplane in Γ as follows.

Definition 5.4. The *even score* of hyperplane $h_{i,j}$ is defined to be $\min\{N_+, N_-\} - \beta N$, where $\beta > 0$ is a balancing parameter, and N_+, N_- and N denote the number of partitions in C which are in $h^+_{i,j}$, are in $h^-_{i,j}$ and intersect $h_{i,j}$, respectively.

Intuitively, a higher even score means that the hyperplane divides the partitions in C more evenly. The first term $\min\{N_+, N_-\}$ shows that we want more partitions in $h^+_{i,j}$ or $h^-_{i,j}$ (and as even as possible). The second term $-\beta N$ gives a penalty for those partitions intersecting $h_{i,j}$ since they will not contribute to the reduction on the size of C. In each round, we present the user with points p_i and p_j, where hyperplane $h_{i,j}$ in Γ has the highest even score.

5.2.3 Stopping Condition. They are defined based on C.

Stopping Condition 1. If there is only one partition Θ left in C, we stop and return the associated point of Θ to the user.

Stopping Condition 2. Recall that $R = \cup_{\Theta \in C} \Theta$. We stop if there exists a point in D which is one of the top-k points w.r.t. any utility vector in R. The following lemma gives a sufficient condition to check whether a given point p_i is qualified to be returned.

Lemma 5.5. *Given utility range R and a point $p_i \in D$, p_i is among the top-k points w.r.t. any $u \in R$ if $|\{p_j \in D \setminus \{p_i\} | R \not\subseteq h^-_{j,i}\}| < k$.*

Intuitively, given two points p_i and p_j, if $R \not\subseteq h^-_{j,i}$, it means that there could be a utility vector u in R such that p_j ranks higher than p_i w.r.t. u. If the number of such kind of points is less than k, p_i is guaranteed to be one of the top-k points w.r.t. any $u \in R$.

To determine whether there exists a qualified point to be returned, a naive idea is to check each $p_i \in D$ using Lemma 5.5. However, it could be time-consuming if D is large. To reduce the burden, we randomly sample a utility vector u in R and only check each of the top-k points w.r.t. u using Lemma 5.5. It is easy to verify that it suffices to check those k points. Note that if there are multiple qualified points, we return an arbitrary one.

5.2.4 Summary. The pseudocode of algorithm *HD-PI* is presented in Algorithm 3. Its theoretical analysis is summarized as follows.

Theorem 5.6. *Algorithm HD-PI solves IST by interacting with the user for $O(n)$ rounds. In particular, if the selected hyperplane can divide C into equal halves without any intersecting partitions in each round (i.e., the optimal case), IST can be solved in $O(\log n)$ rounds.*

Proof. According to the definition of hyperplane $h_{i,j}$, we can remove at least one partition from C in each round. Since there are $O(n)$ partitions, there is one partition left after $O(n)$ rounds and stopping condition 1 is satisfied. If the selected hyperplane divides C into equal halves without any intersecting partitions in each round, we can prune half partitions. After $O(\log n)$ rounds, there exists only one partition and stopping condition 1 is satisfied. □

5.3 RH

In this section, we propose the second algorithm *RH*. It solves IST by asking the user $O(cd \log_2 n)$ ($c > 1$ is a constant) questions in expectation, which is asymptotically optimal if d is fixed.

5.3.1 Information Maintenance. We maintain a polyhedron R, called the *utility range*, in the utility space, which contains the user's utility vector. Initially, R is the entire utility space, i.e., $R = \{u \in \mathbb{R}^d_+ | \sum_{i=1}^d u[i] = 1\}$. In each round, based on the user's preference between p_i and p_j, R is updated to be $R \cap h^+_{i,j}$ (or $h^-_{i,j}$).

5.3.2 Stopping Condition. Since we only maintain the utility range R in *RH*, stopping condition 1 in Section 5.2.3 is no longer applicable. Fortunately, stopping condition 2 in Section 5.2.3 still holds. Besides, we define an additional stopping condition based on R.

Stopping Condition 3. Given two points p_i and p_j in D, if hyperplane $h_{i,j}$ does not intersect with R, the user's utility vector u_0 (note that $u_0 \in R$) must be in either $h^+_{i,j}$ or $h^-_{i,j}$. The user's preference between p_i and p_j is known. If this holds for all pairs of points in D, the ranking of all points in D is known. In this case, we stop and arbitrarily return one of the top-k points.

5.3.3 Point Selection. In each round, hyperplane $h_{i,j}$, which consists of the two presented points, divides R into two smaller halves. Depending on the user's feedback, R becomes smaller by keeping one half left (i.e., $R \cap h^+_{i,j}$ or $R \cap h^-_{i,j}$). The intuition behind our point selection strategy is that if R is smaller, it is easier to meet the stopping conditions. Therefore, in each round, we select two points p_i and p_j, where hyperplane $h_{i,j}$ divides R the most *"evenly"*, hoping that we can reduce the size of R by half after the question. Since it is expensive to compute the exact size of R, we adopt the following heuristic. Denote the center of R by $R_c = \sum_{v \in \mathcal{V}_R} v / |\mathcal{V}_R|$, where $\mathcal{V}_R$ is the set of vertices of R. We present the user with points p_i and p_j, where hyperplane $h_{i,j}$ is the closest to R_c.

However, to select the hyperplane with the minimum distance to R_c, we need to check $O(n^2)$ hyperplanes, which could be costly if n is large. To reduce the number of hyperplanes considered, we initialize a random order of all points in D. With a slight abuse of notations, denote by p_i the i-th point in the random order and define a set H_i of hyperplanes to be $H_i = \{h_{i,j} | \forall j, j < i\}$. The hyperplane selection starts from H_2 (since $H_1 = \emptyset$) and moves to the next H_{i+1} if the current H_i does not contain any hyperplanes intersecting R since only the hyperplanes intersecting R can be used to make R smaller. In each round, we select the hyperplane in the current H_i, which intersects R and has the smallest distance to R_c.

5.3.4 Summary. The pseudocode of algorithm *RH* is shown in Algorithm 4. Its theoretical analysis is presented in Theorem 5.7.

Theorem 5.7. *Algorithm RH solves IST by interacting with the user for $O(cd \log n)$ rounds in expectation, where $c > 1$ is a constant.*

Proof Sketch. We first show that if the probabilities of all the possible rankings are equal, *RH* asks $O(d \log n)$ questions in expectation and then prove that in the general case, the expected number of questions asked is $O(cd \log n)$, where $c > 1$ is a constant. □

Corollary 5.8. *Algorithm RH is asymptotically optimal in terms of the number of questions asked in expectation for IST if d is fixed.*

Algorithm 3: The *HD-PI* Algorithm

Input: A point set D
Output: A point p, which is one of the user's top-k points
Divide the utility space into several partitions Θ_x
$C \leftarrow \{\Theta_1, \Theta_2, ..., \Theta_m\}$
Initialize the utility range R and the list Γ
while $|C| > 1$ & $\nexists p \in D$ *satisfying Lemma 5.5* **do**
  Select hyperplane $h_{i,j}$ in Γ with the highest even score
  Display points p_i and p_j to the user
  Update R, C and Γ based on the user's feedback
return a point p, which is one of the user's top-k points

Algorithm 4: The *RH* Algorithm

Input: A point set D
Output: A point p, which is one of the user's top-k points
Initialize a random order of points in D, $i \leftarrow 2$
Initialize utility range R and set H_i
while *the stopping conditions are not satisfied* **do**
  while $\nexists h_{i,j} \in H_i$ *intersecting* R **do**
    Initialize set H_{i+1}, $i \leftarrow i + 1$
  Find $h_{i,j} \in H_i$ intersecting R with the min-distance to R_c
  Display points p_i and p_j to the user
  Update R based on the user's feedback
return a point p, which is one of the user's top-k points

6 EXPERIMENTS

We conducted experiments on a machine with 3.10GHz CPU and 16GB RAM. All programs were implemented in C/C++.

Datasets. The experiments were conducted on synthetic and real datasets which are commonly used in existing studies [13, 16, 24, 36]. Specifically, the synthetic datasets are *anti-correlated* [7] and the real datasets are *Island*, *Weather*, *Car* and *NBA*. *Island* contains 63, 383 2-dimensional geographic locations and *Weather* includes 178, 080 tuples described by four attributes. *Car* is 4-dimensional, which consists of 68, 010 used cars after it is filtered by only keeping the cars whose attribute values are in normal range. *NBA* involves 16, 916 players after the records with missing values are deleted. Six attributes are used to describe the performance of each player. For all the datasets, each dimension is normalized to $(0, 1]$. Note that in existing studies [8, 36], they preprocessed datasets to contain skyline points only (which are all possible top-1 points for any utility function) since they look for (close to) top-1 point. Consistent with their setting, we preprocessed all the datasets to include k-skyband points (which are all possible top-k points for any utility function) [13] since we are interested in one of the top-k points.

Algorithms. We evaluated our 2-dimensional algorithm: *2D-PI* and d-dimensional algorithms: *HD-PI* and *RH*. As mentioned in Section 5.2.1, the sampling strategy is utilized to accelerate finding convex points for *HD-PI*. Specifically, the utility space is uniformly sampled and the top-1 point w.r.t. each sampled utility vector is found. Based on the different strategies for finding convex points, *HD-PI* is distinguished into two versions: accurate and sampling. The competitor algorithms are: *Median* [36], *Hull* [36], *Active-Ranking* [14], *UtilityApprox* [22], *UH-Random* [36], *UH-Simplex* [36] and *Preference-Learning* [27]. Since none of them can solve our problem directly, we adapted them as follows:

- Algorithms *Median* and *Hull* (only work in a 2-dimensional space) return the user's top-1 point by interacting with the user. We keep these two algorithms and create a new version of them, namely *Median-Adapt* and *Hull-Adapt*, by modifying their *point deletion condition* to that a point is deleted if it cannot be one of

the user's top-k (originally top-1) points according to the learnt information, and their *stopping condition* to that the algorithm stops if there are fewer than k points left (instead of 1 point left).

- Algorithm *Active-Ranking* focuses on learning the full ranking of all points by interacting with the user. We arbitrarily return one of the top-k points when the ranking is obtained.
- Algorithm *UtilityApprox* reduces the regret ratio (used for evaluating returned points) [22] by interacting with the user. It terminates when the regret ratio satisfies a given threshold ϵ. We set $\epsilon = 1 - f(p_k)/f(p_1)$, where p_1 and p_k are points with the first and k-th largest utility w.r.t. the user's utility vector, respectively. In this way, if the regret ratio of the returned point is smaller than ϵ, the returned point must be one of the top-k points.
- Algorithms *UH-Simplex* and *UH-Random* return one point, where either its utility is the largest w.r.t. the user's utility vector or its regret ratio satisfies a given threshold ϵ, by interacting with the user. We set ϵ the same as that in algorithm *UtilityApprox*. Besides, we create another version of *UH-Simplex* and *UH-Random*, namely *UH-Simplex-Adapt* and *UH-Random-Adapt*, by modifying their point deletion condition and their stopping condition in the same way we handle algorithms *Median-Adapt* and *Hull-Adapt*.
- Algorithm *Preference-Learning* learns the user's utility vector by interacting with the user. According to the experimental results in [27], the utility vector learnt is very close to the theoretical optimum if the error threshold ϵ of the learnt utility vector is set to a value *below* 10^{-5} (e.g., 10^{-6}). In our experiment, we set ϵ to 10^{-6} (since the learnt utility vector could achieve the optimum), and arbitrarily return one of the top-k points w.r.t. the learnt utility vector.

Parameter Setting. We evaluated the performance of each algorithm by varying different parameters: (1) different bounding strategies; (2) parameter β, which is a balancing parameter in the even score shown in Section 5.2.2; (3) the dataset size n; (4) the dimensionality d; (5) the parameter k. Unless stated explicitly, for each synthetic dataset, the number of points was set to 100, 000 (i.e., $n = 100,000$) and the dimensionality was set to 4 (i.e., $d = 4$).

Performance Measurement. We evaluated the performance of each algorithm by two measurements: (1) *execution time* which is the processing time; (2) *the number of questions asked* which is the number of rounds interacting with the user. Each algorithm was conducted 10 times with different generated user utility vectors and the average performance was reported.

In the following, the parameter setting of our algorithms is studied in Section 6.1. The performance of all algorithms on the synthetic and real datasets is presented in Section 6.2 and Section 6.3, respectively. In Section 6.4, a user study under a purchasing used car scenario is demonstrated. Section 6.5 shows our motivation study, and finally, the experiments are summarized in Section 6.6.

6.1 Performance Study of Our Algorithms

We compared different bounding strategies applied to our algorithm *HD-PI*. To evaluate their performance, we included two measurements: (1) *effective ratio* which is the ratio N_B/N_I, where N_I is the number of times to identify the relationship between hyperplanes and partitions, and N_B is the number of times that the relationship between hyperplanes and partitions can be identified by bounding ball/rectangle strategy; (2) *execution time* which is the execution time of *HD-PI* with different bounding strategies. For the first measurement, we compared 2 variants of *HD-PI*, namely *HD-PI(Ball)* and *HD-PI(Rectangle)*. *HD-PI(Ball)* (*HD-PI(Rectangle)*) is *HD-PI* using the bounding ball (rectangle) strategy. For the second measurement, we additionally included one variant called *HD-PI(NoBall-NoRectangle)* which is *HD-PI* without using any bounding strategy. As shown in Figure 5, the bounding rectangle strategy can identify more relationships than the bounding ball strategy, where their effective ratios are around 30% and 20%, respectively. However, the execution time by applying the bounding ball strategy is smaller since it only needs $O(1)$ time to check each relationship. Thus, we stick to the bounding ball strategy in the rest of the experiments.

We studied the balancing parameter β in the even score (shown in Definition 5.4), on algorithm *HD-PI* in Figure 6, through evaluating the execution time and the number of questions asked. The result shows that the two measurements increase when β increases. Thus, we set $\beta = 0.01$ in the rest of the experiments.

In Figure 7, we evaluated the result quality of algorithm *HD-PI* with the sampling strategy for finding convex points. Following [8, 10], we define the *accuracy* of the returned point p to be $f(p)/f(p_k)$ if $f(p) < f(p_k)$ (otherwise the accuracy is set to 1), where point p_k has the k-th largest utility in D w.r.t. the user's utility vector. It can be seen that the accuracy of the returned point on different datasets are close to 1. This concludes that the sampling strategy for finding convex points affects little to the result quality.

6.2 Performance on Synthetic Datasets

We compared our 2-dimensional algorithm *2D-PI*, against *Median*, *Hull*, *Median-Adapt* and *Hull-Adapt* on a 2-dimensional dataset by varying the parameter k. For completeness, our d-dimensional algorithms, *HD-PI* and *RH*, and existing ones were also involved by setting $d = 2$ (Since the performance of d-dimensional algorithms is almost the same as that on a 4-dimensional dataset, we analyze them later). Figures 8(a) and (b) show the execution time and the number of questions asked of each algorithm, respectively. All the algorithms finish within a few seconds. Note that *Median* and *Hull* are slightly faster than *2D-PI*. But, they ask much more questions. When $k \geq 60$, they ask three times as many questions as *2D-PI*. Although *2D-PI* spends slightly more time, its execution time is small and reasonable given that it requires the least number of questions for arbitrary k. For *Median-Adapt* and *Hull-Adapt*, although their modified stopping conditions are easier to achieve, the adaptation of the point deletion condition reduces the effectiveness of deleting points, which results in a long execution time and a large number of questions asked (even increase when k increases).

Figures 8 and 9 demonstrate the performance of our algorithms, *HD-PI* and *RH*, and the existing d-dimensional algorithms on a 2-dimensional dataset and 4-dimensional dataset, respectively. Algorithm *Active-Ranking* asks the largest number of questions with a long execution time, e.g., more than 1000 questions and 500 seconds on the 4-dimensional dataset if $k \geq 50$, since it learns the ranking of all points. It even runs gradually slower and asks more questions when k increases, because of its sensitivity to the input size (increases with k due to the k-skyband preprocessing). The execution times of *UH-Random*, *UH-Simplex* and *Preference-Learning* are larger

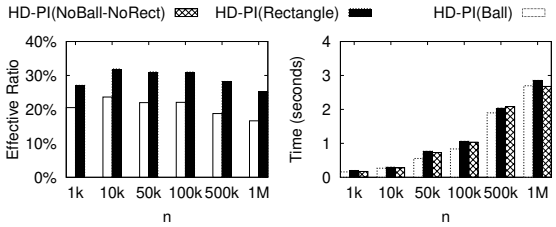


Figure 5: Bounding Strategies

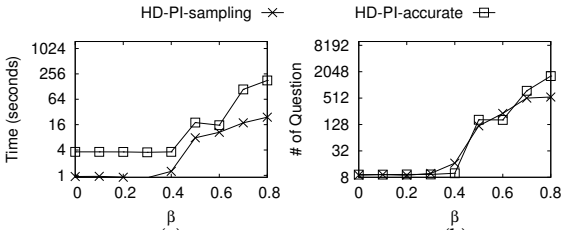


Figure 6: Performances on different β

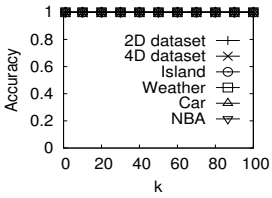
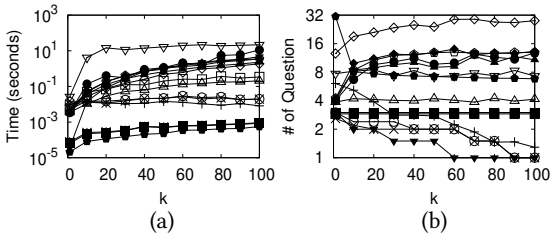


Figure 7: Accuracy



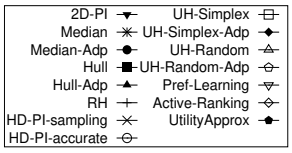


Figure 8: 2D Dataset

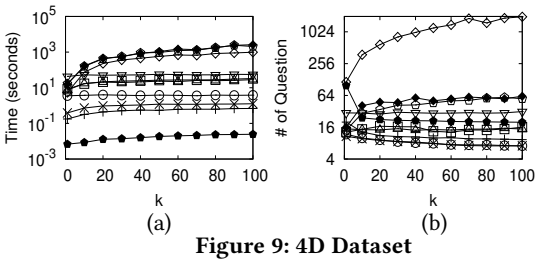
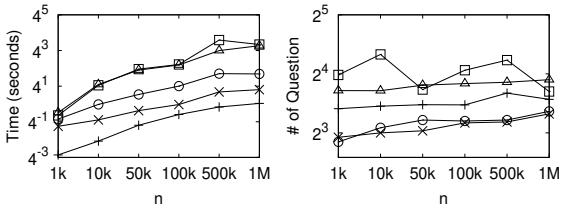


Figure 9: 4D Dataset



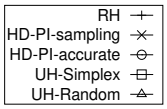


Figure 10: Vary n (k=20, d=4)

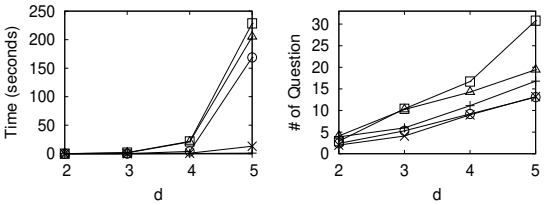
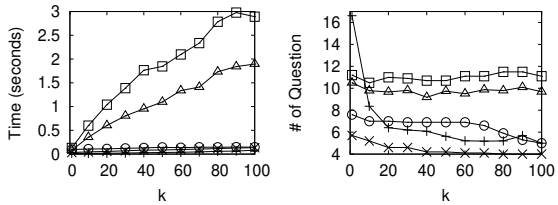


Figure 11: Vary d (k=20, n=100k)



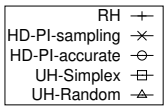


Figure 12: Weather

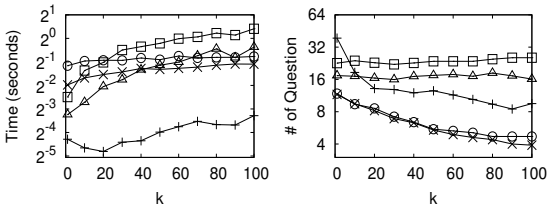


Figure 13: NBA

than those of our algorithms. Specifically, on the 2-dimensional dataset, *UH-Random* and *UH-Simplex* run 2-10 times as much time as our algorithms, and *Preference-Learning* takes about 2-3 orders of magnitude longer than our algorithms. On the 4-dimensional dataset, *UH-Random* and *UH-Simplex* take more than 4 times and *Preference-Learning* takes more than 10 times as much time as our algorithms. For the number of questions asked, *UH-Random* and *UH-Simplex* perform well when $k = 1$, since they are designed specially for returning the top-1 point. However, the number of questions asked almost remains unchanged when k increases. Algorithm *Preference-Learning* also asks much more questions than our algorithms when k is slightly large. For *UH-Random-Adapt* and *UH-Simplex-Adapt*, same as *Median-Adapt* and *Hull-Adapt*, the adaptation of the point deletion condition decreases the effectiveness of deleting points, which results in a long execution time and many questions asked. In particular, they take more than 3 and 2000 seconds and require around 13 and 60 questions when $k = 100$ on the 2-dimensional and the 4-dimensional datasets, respectively. Except for *UtilityApprox*, our algorithms *RH* and *HD-PI* take the least time. For arbitrary k, they only require within 0.02 second and 4 seconds on the 2-dimensional and 4-dimensional datasets, respectively. Note that *UtilityApprox* is faster than our algorithm, since it constructs fake points and does not rely on the dataset. However, as we argue in Section 2, displaying fake points is not suitable for real systems. Although our algorithms take slightly more time, our execution times are small and reasonable since our algorithms use real points (points from datasets) and ask the least number of questions. For example, when $k = 100$ on the 4-dimensional dataset, our algorithms only require half the number of questions asked by the best existing algorithm. Moreover, except for *UtilityApprox*, there is no existing algorithm whose number of questions asked decreases significantly when k increases, while our algorithms can have at least 32% reduction. According to the results, in the following, we use the state-of-the-art existing d-dimensional algorithms *UH-Random* and *UH-Simplex* for comparison, and include algorithms *Median* and *Hull* for the 2-dimensional case additionally.

In Figure 10, we studied the scalability on the dataset size n. Our algorithms *HD-PI* and *RH* scale the best in terms of the execution time and the number of questions asked. For example, our execution times are less than 10 seconds even if $n \geq 500,000$, while the others run up to 100 seconds. Our algorithms also ask at least 2 fewer questions than the others for arbitrary n. In Figure 11, we evaluated the scalability on the dimensionality d. Compared with the existing ones, *RH* and *HD-PI* consistently require fewer questions and less execution time for arbitrary d. For instance, when $d = 4$, the number of questions required by *UH-Random* and *UH-Simplex* are around 15 and 17, while our algorithms need at most 11 questions. When $d = 5$, *UH-Simplex* and *UH-Random* run about 230 and 205 seconds, respectively, while *RH* finishes within only 1 seconds.

6.3 Performance on Real Datasets

We studied the performance of our algorithms against existing algorithms, namely *Median* (for 2D), *Hull* (for 2D), *UH-Random* and *UH-Simplex*, on 4 real datasets. Due to the lack of space, we only show the performance on *Weather* (with the largest data size) and *NBA* (with the largest dimensionality) in Figure 12 and Figure 13, respectively. The results on *Island* and *Car* can be found in the technical report [34]. Except that *HD-PI* takes similar time with *UH-Random* on dataset *NBA*, our algorithms perform much better than the others both on the number of questions asked and the execution time on *Weather* and *NBA*. For instance, when $k \geq 50$, both *HD-PI* and *RH* need nearly half the number of questions asked by the others on both datasets. As for the execution time, both *HD-PI* and *RH* spend at most 0.15 seconds on *Weather* when $k \geq 50$, while the others take more than 1 second.

6.4 User Study

We conducted a user study on the used car dataset *Car* to see the impact of user mistakes on the final results, since users might make mistakes or provide inconsistent feedback during the interaction. Following the setting in [27, 36], 1000 candidate cars were randomly selected from the dataset described by 4 attributes, namely price, year of purchase, power and used kilometers. 30 participants were recruited and their average result was reported. We compared our algorithms *RH* and *HD-PI* in sampling and accurate versions, against 4 existing algorithms, namely *UH-Random*, *UH-Simplex*, *Preference-Learning* and *Active-Ranking*. Each algorithm aims at finding one of the user's top-20 cars. Since the user's utility vector is unknown, we re-adapted algorithms *UH-Random*, *UH-Simplex* and *Preference-Learning* (instead of the way described previously).

- Algorithm *Preference-Learning* maintains an estimated user's utility vector u_e during the interaction. We compared the user's answer of some randomly selected questions with the prediction w.r.t. u_e. If 75% questions [27] can be correctly predicted, we stop and return one of the top-20 cars w.r.t. u_e.
- For *UH-Random* and *UH-Simplex*, we set the threshold $\epsilon = 0$ and this guarantees that the returned car is one of the top-20 cars.

Each algorithm was measured via: (1) *The number of questions asked*; (2) *Degree of boredness* which is a score from 1 to 10 given by each participant. It indicates how bored the participant feels when s/he sees the returned car after being asked several questions (1 denotes the least bored and 10 means the most bored). (3) *Rank* which is the ranking given by each participant. Specifically, since the participants sometimes gave the same score for different algorithms, to distinguish them clearly, we asked each participant to give a ranking of the algorithms. Note that to reduce the workload, we only asked for the ranking of 5 algorithms with the least number of questions asked and the lowest degree of boredness: *RH*, *HD-PI* in sampling and accurate versions, *UH-Random* and *UH-Simplex*.

Figure 16 shows the results. The number of questions required by *HD-PI-sampling*, *HD-PI-accurate* and *RH* are 4.1, 4.8 and 7.1, respectively, while existing algorithms ask more than 8.4 questions. In particular, the number of questions asked by *Preference-Learning* and *Active-Ranking* are up to 20.3 and 45.4, respectively. Besides, our algorithms *HD-PI-sampling*, *HD-PI-accurate* and *RH* are the least boring and rank the best. Their degrees of boredness are 1.9, 2.13 and 3, respectively. In comparison, the degrees of boredness of existing algorithms are more than 3.75 and the most boring algorithm *Active-Ranking* is up to 7.7, especially.

6.5 Motivation Study

In this section, we gave experimental results/studies to show why problem IST is effective in practice. In Section 6.5.1, we first compared the result in problem IST (returning *one* of the top-k points) with the result in a variant of problem IST returning *all* the top-k points. We find that the result in problem IST is convincing since our problem IST could involve less user interaction and execution time. Then, in Section 6.5.2, we performed a user study to compare the result in problem IST with the result in another variant of problem IST returning *some* of the top-k points. We find that the result in problem IST is also convincing since participants in the user study felt the most satisfied with the result in problem IST.

6.5.1 User Effort Comparison. We compared the user effort required by two cases: returning one of the top-k points vs. returning all the top-k points by interacting with the user. Our algorithms *RH* and *HD-PI* are modified to return all the top-k points as follows. (1) Their stopping conditions are changed. Recall that we maintain a utility range R in the utility space which contains the user's utility vector. The algorithms stop if we can find k points which are the top-k points w.r.t. any utility vector in R, i.e., if there are k points which fulfill Lemma 5.5. (2) The information maintenance part of *HD-PI* is extended. A point set V_d is maintained. When there is only one partition Θ left in set C and the modified stopping condition is not satisfied, the associated point of partition Θ is added to V_d. Then, partition Θ is divided into several smaller partitions Θ_i added to C such that $\Theta_i = \{u \in \Theta | u \in h^+_{i,j}, \forall p_j \in V \setminus \{p_i\}\}$, where V contains all the convex points in $D \setminus V_d$.

The original version (returning one of the top-k points), denoted by *RH*, *HD-PI-sampling* and *HD-PI-accurate*, and the modified version (returning all the top-k points), denoted by *RH-AllTopK*, *HD-PI-sampling-AllTopK* and *HD-PI-accurate-AllTopK*, were conducted on the 6 datasets described before. Due to the lack of space, we only show in Figures 14 and 15 the performance on the 4-dimensional synthetic dataset and the *NBA* dataset (with the largest dimensionality). The results on the other datasets can be found in the technical report [34]. It can be seen that the modified version is costly. It runs $1-2$ orders of magnitude longer and requires 4-10 times more

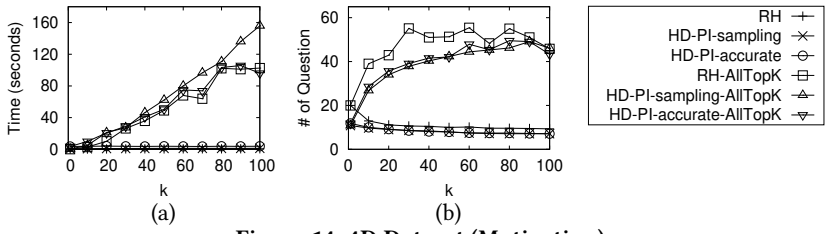


Figure 14: 4D Dataset (Motivation)

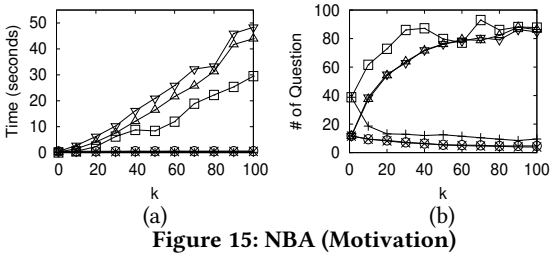


Figure 15: NBA (Motivation)

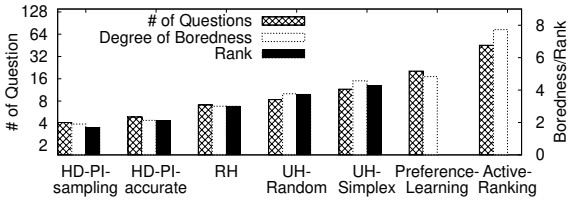


Figure 16: User Study

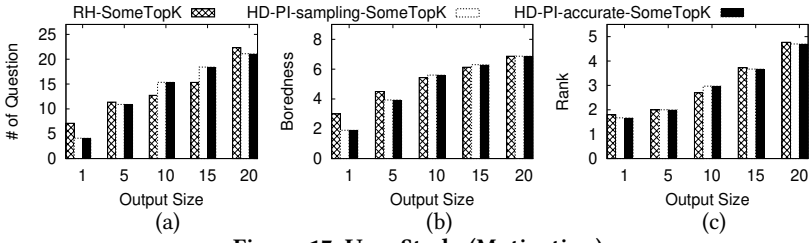


Figure 17: User Study (Motivation)

questions than the original version when $k \geq 20$. For example, *RH* asks about 9 questions within 0.08 seconds, while *RH-AllTopK* needs around 86 questions within 22 seconds on the *NBA* dataset when $k = 80$. This verifies our claim that returning more than one of the top-k comes with the price of additional user effort and thus, for achieving a good trade-off between the output size and the user effort, our current problem setting is one best option.

6.5.2 User Satisfactory Study. We conducted a user study on our algorithms *RH* and *HD-PI* in sampling and accurate versions to verify whether users are willing to spend more effort for larger output in real scenarios (i.e., whether users would like to obtain *some* (possibly, more than one) of the top-k points). Following the setting in Section 6.4, we selected 1000 candidate cars randomly from dataset *Car* and recruited 30 participants. Our algorithms are modified to return k' ($k' \leq 20$) out of the top-20 cars. Specifically, each algorithm stops if there are k' points which are the top-k points w.r.t. any utility vector in utility range R, i.e., if there are k' points which fulfill Lemma 5.5. The modified algorithms are denoted by *RH-SomeTopK*, *HD-PI-SomeTopK* and *HD-PI-accurate-SomeTopK*.

We studied each algorithm under 5 cases: returning 1, 5, 10, 15, 20 cars out of the top-20 cars by interacting with the user and evaluated each case with three measurements. (1) *The number of questions asked*; (2) *Degree of boredness* (mentioned in Section 6.4); (3) *Rank* which is the 5 cases' ranking given by each participant. For the latter two measurements, participants were asked to consider both the number of returned cars and their effort spent.

Figure 17 shows the results. With the increasing output size, the number of questions asked increases dramatically, and the degree of boredness and the rank also increase. Nevertheless, returning one of the top-20 cars asks the least number of questions, has the lowest degree of boredness and ranks the best. This indicates that returning one of the top-k points achieves the best user satisfactory level.

6.6 Summary

The experiments showed the superiority of our algorithms over the best-known existing ones: (1) We are effective and efficient. Algorithms *RH* and *HD-PI* ask fewer questions within less time than existing algorithms (e.g., half the number of questions asked on the 2-dimensional dataset compared with *UH-Random* and *UH-Simplex* when $k \geq 80$). (2) Our algorithms scale well on the dimensionality and the dataset size (e.g., ours ask at most 11 questions when $n = 1,000,000$ on the 4-dimensional dataset, while *UH-Random* and *UH-Simplex* need around 15 and 17 questions). (3) The bounding strategies are useful. The bounding ball strategy can identify more than 20% relationship between hyperplanes and partitions. In summary, *2D-PI* asks the least number of questions in a 2-dimensional space with a small execution time (e.g., one third of questions asked compared with *Median* and *Hull* when $k \geq 60$). In a d-dimensional space, *RH* runs the fastest (e.g., it runs within 1 second, while *UH-Random* and *UH-Simplex* need more than 200 seconds when $d = 5$) and *HD-PI* asks the least number of questions (e.g., half the number of questions asked on the 4-dimensional dataset compared with *UH-Random* and *UH-Simplex* when $k \geq 50$).

7 CONCLUSION

In this paper, we present interactive algorithms for searching one of the user's top-k tuples, pursing as little user effort as possible. In a 2-dimensional space, we propose algorithm *2D-PI*, which is asymptotically optimal w.r.t. the number of questions asked. In a d-dimensional space, two algorithms *RH* and *HD-PI* are presented, and perform well w.r.t. the execution time and the number of questions asked. In particular, for *HD-PI*, we propose two versions: sampling and accurate, which target at speed and accuracy, respectively. Extensive experiments showed that our algorithms are both efficient and effective in finding one of the user's top-k tuples by asking few questions within little time. As for future work, we consider the situation that users might make mistakes when answering questions.

8 ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their constructive comments on this paper. The research of Weicheng Wang and Raymond Chi-Wing Wong is supported by *IRS17EG25*. The research of Min Xie is supported by *LHKJCXJCYJ202003*.

REFERENCES

[1] Abolfazl Asudeh, Azade Nazi, Nan Zhang, Gautam Das, and H. V. Jagadish. 2019. RRR: Rank-Regret Representative. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 263–280.

[2] Wolf-Tilo Balke, Ulrich Güntzer, and Christoph Lofi. 2007. Eliciting Matters – Controlling Skyline Sizes by Incremental Integration of User Preferences. In *Advances in Databases: Concepts, Systems and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 551–562.

[3] Wolf-Tilo Balke, Ulrich Güntzer, and Christoph Lofi. 2007. User Interaction Support for Incremental Refinement of Preference-Based Queries. In *Proceedings of the First International Conference on Research Challenges in Information Science*. 209–220.

[4] Ilaria Bartolini, Paolo Ciaccia, Vincent Oria, and M. Tamer Özsu. 2007. Flexible integration of multimedia sub-queries with qualitative preferences. *Multimedia Tools and Applications* 33, 3 (2007), 275–300.

[5] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. 2014. Domination in the Probabilistic World: Computing Skylines for Arbitrary Correlations and Ranking Semantics. *ACM Transactions on Database System* 39, 2 (2014).

[6] Ilaria Bartolini, Paolo Ciaccia, and Florian Waas. 2001. FeedbackBypass: A New Approach to Interactive Similarity Query Processing. In *Proceedings of the 27th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 201–210.

[7] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *Proceedings of the International Conference on Data Engineering*. 421–430.

[8] Wei Cao, Jian Li, Haitao Wang, Kangning Wang, Ruosong Wang, Raymond Chi-Wing Wong, and Wei Zhan. 2017. k-Regret Minimizing Set: Efficient Algorithms and Hardness. In *20th International Conference on Database Theory*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 11:1–11:19.

[9] Bernard Chazelle. 1993. An Optimal Convex Hull Algorithm in Any Fixed Dimension. *Discrete Computational Geometry* 10, 4 (1993), 377–409.

[10] Sean Chester, Alex Thomo, S. Venkatesh, and Sue Whitesides. 2014. Computing K-Regret Minimizing Sets. In *Proceedings of the VLDB Endowment*, Vol. 7. VLDB Endowment, 389–400.

[11] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. 2008. *Computational geometry: Algorithms and applications*. Springer Berlin Heidelberg.

[12] Brian Eriksson. 2013. Learning to Top-k Search Using Pairwise Comparisons. In *Proceedings of the 16th International Conference on Artificial Intelligence and Statistics*, Vol. 31. PMLR, Scottsdale, Arizona, USA, 265–273.

[13] Yunjun Gao, Qing Liu, Baihua Zheng, Li Mou, Gang Chen, and Qing Li. 2015. On processing reverse k-skyband and ranked reverse skyline queries. *Information Sciences* 293 (2015), 11–34.

[14] Kevin G. Jamieson and Robert D. Nowak. 2011. Active Ranking Using Pairwise Comparisons. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 2240–2248.

[15] Bin Jiang, Jian Pei, Xuemin Lin, David W. Cheung, and Jiawei Han. 2008. Mining Preferences from Superior and Inferior Examples. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, USA, 390–398.

[16] Georgia Koutrika, Evaggelia Pitoura, and Kostas Stefanidis. 2013. Preference-Based Query Personalization. *Advanced Query Processing* (2013), 57–81.

[17] Jongwuk Lee, Gae-won You, and Seung-won Hwang. 2009. Personalized top-k skyline queries in high-dimensional space. *Information Systems* 34 (2009), 45–61.

[18] Jongwuk Lee, Gae-Won You, Seung-Won Hwang, Joachim Selke, and Wolf-Tilo Balke. 2012. Interactive skyline queries. *Information Sciences* 211 (2012), 18–35.

[19] Tie-Yan Liu. 2010. Learning to Rank for Information Retrieval. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, NY, USA, 904.

[20] Alchemer LLC. 2021. https://www.alchemer.com/resources/blog/how-many-survey-questions/

[21] Lucas Maystre and Matthias Grossglauser. 2017. Just Sort It! A Simple and Effective Approach to Active Preference Learning. In *Proceedings of the 34th International Conference on Machine Learning*. 2344–2353.

[22] Danupon Nanongkai, Ashwin Lall, Atish Das Sarma, and Kazuhisa Makino. 2012. Interactive Regret Minimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 109–120.

[23] Danupon Nanongkai, Atish Das Sarma, Ashwin Lall, Richard J. Lipton, and Jun Xu. 2010. Regret-Minimizing Representative Databases. In *Proceedings of the VLDB Endowment*, Vol. 3. VLDB Endowment, 1114–1124.

[24] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive Skyline Computation in Database Systems. *ACM Transactions on Database Systems* 30, 1 (2005), 41–82.

[25] Peng Peng and Raymond Chi-Wing Wong. 2014. Geometry approach for k-regret query. In *Proceedings of the International Conference on Data Engineering*. 772–783.

[26] Peng Peng and Raymong Chi-Wing Wong. 2015. K-Hit Query: Top-k Query with Probabilistic Utility Function. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 577–592.

[27] Li Qian, Jinyang Gao, and H. V. Jagadish. 2015. Learning User Preferences by Adaptive Pairwise Comparison. In *Proceedings of the VLDB Endowment*, Vol. 8. VLDB Endowment, 1322–1333.

[28] QuestionPro. 2021. https://www.questionpro.com/blog/optimal-number-of-survey-questions/

[29] Melanie Revilla and Carlos Ochoa. 2017. Ideal and Maximum Length for a Web Survey. *International Journal of Market Research* 59, 5 (2017), 557–565.

[30] J.-R. Sack and J. Urrutia. 2000. *Handbook of Computational Geometry*. North-Holland, Amsterdam.

[31] Gerard Salton. 1989. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley Longman Publishing Co., Inc., USA.

[32] Thomas Seidl and Hans-Peter Kriegel. 1997. Efficient User-Adaptable Similarity Search in Large Multimedia Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 506–515.

[33] Mohamed A. Soliman and Ihab F. Ilyas. 2009. Ranking with uncertain scores. In *Proceedings of the International Conference on Data Engineering*. 317–328.

[34] Weicheng Wang, Raymond Chi-Wing Wong, and Min Xie. 2021. *Interactive Search for One of the Top-k*. Technical Report. http://www.cse.ust.hk/~raywong/paper/interactiveOneOfTopK-technical.pdf

[35] Min Xie, Tianwen Chen, and Raymond Chi-Wing Wong. 2019. FindYourFavorite: An Interactive System for Finding the User's Favorite Tuple in the Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 2017–2020.

[36] Min Xie, Raymond Chi-Wing Wong, and Ashwin Lall. 2019. Strongly Truthful Interactive Regret Minimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 281–298.

[37] Min Xie, Raymond Chi-Wing Wong, and Ashwin Lall. 2020. An experimental survey of regret minimization query and variants: bridging the best worlds between top-k query and skyline query. *VLDB Journal* 29, 1 (2020), 147–175.

[38] Min Xie, Raymond Chi-Wing Wong, Jian Li, Cheng Long, and Ashwin Lall. 2018. Efficient K-Regret Query Algorithm with Restriction-Free Bound for Any Dimensionality. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 959–974.

[39] Min Xie, Raymond Chi-Wing Wong, Peng Peng, and Vassilis J. Tsotras. 2020. Being Happy with the Least: Achieving α-happiness with Minimum Number of Tuples. In *Proceedings of the International Conference on Data Engineering*. 1009–1020.

[40] Jiping Zheng and Chen Chen. 2020. Sorting-Based Interactive Regret Minimization. In *Web and Big Data-4th International Joint Conference, APWeb-WAIM*. Springer, 473–490.