Interactive Search for One of the Top-k

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** → **Data analytics**.

KEYWORDS

top-k query; user interaction; data analytics

ACM Reference Format:

. 2021. Interactive Search for One of the Top-k. In *2021 International Conference on Management of Data (SIGMOD '21), June 20 - June 25, 2021, Xi'an, Shaanxi.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Assisting users to find satisfied tuples in a large dataset is an important task in a variety of application domains, including purchasing used cars, renting apartments and searching dating partners. For example, consider the scenario about 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 [27, 32, 38], 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 which 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 [21, 31, 36] and the skyline query [11]. The first operator is the top-k query [21, 31, 36], 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 [27, 38]. But 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 [11] 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 [27, 28].

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 [32, 38], *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 characteristics of problem IST is that it involves a number of questions 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 [1, 2, 33], 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, 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 is 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 on cherry-picking. She

SIGMOD '21, June 20-25, 2021, Xi'an, Shaanxi

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [19, 27, 32, 38] but they are different from ours. [19] 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 *general position* [25], which could not be applied in many cases. Roughly speaking, all tuples in general position satisfies some geometry properties (e.g., no 3 points are co-linear), which is not realistic in real datasets. [27, 38] 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 [27, 38] could return the top-1 tuple as the answer. However, in our experiment, more than 15 questions are needed 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, [27] involves "fake" tuples (i.e., tuples not in the dataset) in the questions during user interaction. [32] involves user interaction and finds the user's utility function by interacting with the user. Similar to [19], since [32] 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 [19, 27, 32, 38] requires 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 [8, 35] and the regret minimizing query [14, 28, 30]. The similarity query [8, 35] 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 [9]. In practice, a user do 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 [14, 28, 30], another type of preference-based queries, 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 database, 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 [14, 38]. For further details, see [13, 39] and references therein.

To overcome the deficiencies of the preference-based queries, some existing studies [6, 7, 9, 10, 22, 27, 34, 38, 40] involve user interaction. [6, 7, 22] proposed the interactive skyline query, which tries to reduce the number of skyline tuples in the answer by interacting with users. However, it limits on learning the user preference on the values on an attribute (e.g., values *red*, *yellow* and *blue* for attribute color). Even if the preference on all attribute values is obtained, the number of skyline tuples could still be arbitrarily large [28].

[9, 10, 34] 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.

[27] 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 simple 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 database). 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 [38]. To overcome the defect, [38] proposed algorithms

UH-Simplex and *UH-Random*, which utilize *real tuples* (selected from the database) 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 [38] 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 [38, 40] focus on finding the user's (close to) favorite tuple. To some extent, their problems [38, 40] can be seen as a special case of our problem when $k = 1$.

There are alternative approaches [20, 32] which focus on learning the user preference with the help of user interaction. [32] proposed algorithm *Preference-Learning* which approximates the user preference by pairwise comparison. Nevertheless, it focuses on learning the preference rather than returning tuples, and thus it might ask the user unnecessary questions [38]. For example, if Alice prefers car p_1 to both car 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 [32]. In addition, [20] 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* [3, 17, 19, 23, 24], which learns the ranking of tuples by pairwise comparison. However, most of the existing methods [3, 17, 23, 24] only consider the relation 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 a inter-relation showing that \$200 is better than \$500 (since \$200 is cheaper)), and thus, require more feedback from the user [38]. Algorithm *Active-Ranking* [19] considers the inter-relation between tuples to learn the ranking by interacting with users. However, it assumes that all the tuples are in *general position* [25], 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 [32].

Our work focuses on returning one of the top-k tuples by interacting with users 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) Unlike [27] (utilizing fake tuples), we only use real tuples during the interaction. (5) We involve a few questions only since we return one of the top-k tuples. Firstly, existing studies like [3, 17, 23, 24] ask a lot of questions since they require to learn a full ranking. Secondly, [3, 17, 23, 24] do not utilize the inter-relation between tuples and thus, some unnecessary interaction is involved. Thirdly, [40] requires a user to sort tuples and [9, 34] requires a user to assign concrete scores. But, we just require a user to pick a favorite tuple between two candidates for each question, but 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 the tuples are in 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 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 [37, 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 [4, 38], 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 $f : \mathbb{R}_+^d \to \mathbb{R}_+$, 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 which 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 the paper, we use "utility vector" to refer to the user preference and call the domain of u as 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 [27, 38], 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. At each round, the system adaptively 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 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.

Algorithm (d dimension)	Worst Case	Optimal Case	Expected Case
UH-Random [38]	$O(n)$	Unknown	Unknown
UH-Simplex [38]	$O(n)$	Unknown	$O(deg_{max}\sqrt[d]{n})$
Active-Ranking [19]	$O(n^2)$	$O(d\log n)$	$O(cd\log n)$, where$c>1$
Preference-Learning [32]	$O(n^2)$	Unknown	Unknown
RH	$O(n\log n)$	$O(d\log n)$	$O(cd\log n)$, 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: Database($u = (0.4, 0.6)$)

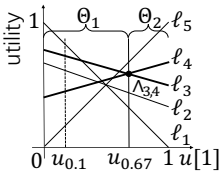


Figure 1: 2D Partitions

Problem 1. ***(Interactive Search for One of the Top-k (IST))*** *Given a point set D, we are to ask the 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 is a dataset of n points such that for any algorithm, it needs to ask the 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 $q \in D/\{p\}$ 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, the utility of p can be viewed as a line segment ℓ when the utility function represented by $u[1]$ varies: $f(p) = (p[1]-p[2])u[1]+p[2]$ 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 $\mathcal{Q}$ 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 $\mathcal{Q}$, 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 $\mathcal{Q}$ 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 $\mathcal{Q}$. 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 $\mathcal{Q}$, and insert into $\mathcal{H}$ two new intersections (i.e., p_i and its new neighbour in $\mathcal{Q}$ and p_j and its new neighbour in $\mathcal{Q}$) if they are valid. If p_i and p_j are the k-th and $(k+1)$-th point in $\mathcal{Q}$ 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 $\mathcal{Q}$ 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 $\mathcal{Q} =< p_1, p_3, p_2, p_4, p_5 >$ 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 $\mathcal{Q}$ 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 into $\mathcal{H}$. Besides, the label of p_1 and p_3 are not updated since they are not the k-th and $(k+1)$-th point in $\mathcal{Q}$ before swapping.

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
Insert into $\mathcal{H}$ the intersections (if valid) 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_j, p_i 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 $\Theta = \{\Theta_1, \Theta_2, ..., \Theta_m\}$, $\mathcal{E} = \{q_1, q_2, ..., q_m\}$

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 and 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 partition obtained by our algorithm. We show that $r_i \geq r'_i$, 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 [4, 12] propose algorithms which achieve similar purpose. However, [4] only presents an approximate algorithm, while our algorithm returns an exact solution. Besides, it is not easy to process the real implementation of [12] due to its complicated data structure with its theoretical result. To the best of our knowledge, there are no real implementation of [12] 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 $u[1] < \wedge_{i,j}[1]$, it 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*.

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. At each round, we prune half

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

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 $\wedge_{3,4}[1] = r_1$. If 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 $q_1 = p_3$ 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 at 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, both of which has performance guarantee. In particular, *HD-PI* enjoys good empirical performance and *RH* asks $O(cd \log_2 n)$ questions in expectation ($c \geq 1$ is a constant). For a fixed d, *RH* is asymptotically optimal w.r.t. the number of questions asked.

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

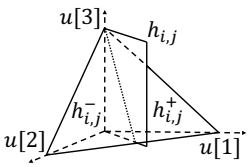


Figure 2: Hyperplane

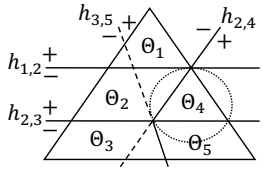


Figure 3: 3D Partitions

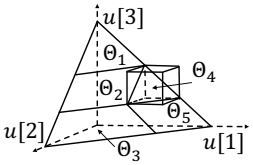


Figure 4: B_Rectangle

$h_{i,j}$	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
$h_{3,4}$	Θ_4, Θ_5	Θ_3	Θ_1, Θ_2	0.8

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

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* [15]. 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 [38].

In geometry, a polyhedron $\mathcal{P}$ is the intersection of a set of halfspaces [15] 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 the user provides a 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 the concepts: *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 first compute the maximum and minimum values of each dimension to be $max_i = \max_{v \in \mathcal{V}} v[i]$ and $min_i = \min_{v \in \mathcal{V}} v[i]$ where $i \in [1, d]$. 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 $h^-_{i,j}$.*

It is easy to see that determining whether $\mathbb{B}(\mathcal{P}) \in h^+_{i,j}$ and $\mathbb{R}(\mathcal{P}) \in h^+_{i,j}$ can be done in $O(1)$ and $O(2^d)$ time by simply checking the distance from $\mathbb{B}_c$ to $h_{i,j}$ and checking each vertex of $\mathbb{R}(\mathcal{P})$, respectively. Though bounding rectangle has a higher time complexity, it provides a "tigher" bounds on $\mathcal{P}$. The performances of these two bounding methods are compared experimentally in Section 6.1.

As an overview, our *HD-PI* and *RH* algorithm follow the interactive framework from [38]: we interact with the user for rounds until we find a point which is the user's top-k points. At each round,

- (**Points selection**) We select two points from the database and present them to the user.
- Then the user picks the favorite point.
- (**Information Maintenance**) Based on the feedback, we update the maintained information.
- (**Stopping condition**) Finally, 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 for each component above.

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 user.

5.2.1 Information Maintenance. We maintain two data structures: (1) a set C, which contains several 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 relations between hyperplanes and partitions in C.

Before a user provides any information, 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 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, we prove in Theorem 5.2 that dividing the utility space into the least number of partitions is an NP-hard problem.

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.*

Therefore, 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* [15] in D based on some existing algorithms [15], 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 $\Theta_i = \{u \in \mathbb{R}^d | u \in h^+_{i,j}, \forall p_j \in V / \{p_i\}\}$ and add it to C. It can be verified that p_i is the top-1 point w.r.t. any $u \in \Theta_i$. Note that there are $O(n)$ partitions in C and initially $R = \cup_{\Theta \in C} \Theta = \{u \in \mathbb{R}^d_+ | \sum_{i=1}^d u[i] = 1\}$.

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

$h_{i,j}$	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 ($\beta = 0.1$)

the partitions in C (i.e., in $h^+_{i,j}$, in $h^-_{i,j}$ or intersect $h_{i,j}$). In addition, we store an *even score* of each $h_{i,j}$ in Γ for measuring how well $h_{i,j}$ divides the partitions in C. Even scores will be used later for point selection and we postpone its formal definition to the next section. Table 3 is the example of Γ for partitions in Figure 3.

When a user answers more questions, more information about the user's utility vector is known and the data structures are updated accordingly. Assume that the user prefers p_i to p_j in a question. Then, invalid partitions in C will be removed and Γ will be updated based on hyperplane $h_{i,j}$. Specifically, for each Θ in C, there are two cases: (1) if Θ is in $h^-_{i,j}$, it is removed from C since Θ 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 $\Theta \cap h^+_{i,j}$. After partitions in C are updated, Γ is updated accordingly. If there is a hyperplane $h_{i',j'}$ in Γ such that R is in $h^+_{i',j'}$ (resp. in $h^-_{i',j'}$), we remove $h_{i',j'}$ from Γ since no matter what the user's utility vector is (as long as it is in R), $p_{i'}$ must rank higher (resp. lower) than $p_{j'}$ and thus, $h^+_{i',j'}$ (resp. $h^-_{i',j'}$) cannot be used to update C later.

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 the user prefers p_3 to p_2. Then, the user's 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 cannot contain the user's utility vector 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, 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 hyperplanes are renewed. The updated Γ is shown in Table 4.

Note that [4] presents an algorithm which focuses on the similar purpose of dividing the utility space into the least number of partitions. However, [4] only obtains an approximate solution by consuming much time. In detail, it has two steps, where the time complexity of the first step is $O(n^{\lfloor d/2 \rfloor} k^{\lfloor d/2 \rfloor})$ [26] and the second step is an NP-hard problem. But, our method only requires $O(n^{\lfloor d/2 \rfloor})$ to obtain a small number of partitions.

5.2.2 Point Selection. In this section, we present the point selection strategy. Intuitively, we aim to filter as many partitions in C as possible at each round so that the user's utility vector can be quickly located. According to the definition of $h_{i,j}$, if the user prefers p_i to p_j, we remove partitions in $h^-_{i,j}$. Otherwise, we remove partitions in $h^+_{i,j}$. In other words, if a hyperplane evenly divides C into two halves, half of the partitions in C can be removed by asking one question, no matter what answer the user provides. Following this idea, at each round, we try to find points p_i and p_j, whose corresponding hyperplane $h_{i,j}$ in Γ divides the partitions in C the *most evenly*, and display p_i and p_j to the user. To evaluate the "evenness", we define an *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 more evenly. The first term in Definition 5.4 shows that we want more partitions in $h^+_{i,j}$ or $h^-_{i,j}$ (and as even as possible), and the second term gives penalty to those partitions intersecting $h_{i,j}$ since those partitions will not contribute to the reduction on the size of C. At each round, we present the user with points p_i and p_j whose hyperplane $h_{i,j}$ in Γ has the highest even score.

5.2.3 Stop Condition. Stopping conditions 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 immediately if there exists a point in D which is guaranteed to be one of the top-k points w.r.t. any utility vector in R. The following lemma tells whether a given point p_i is a qualified point to be returned.

Lemma 5.5. *Given the 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 $|\{p_j \in D |\ R \nsubseteq h^-_{j,i}\}| < k$.*

Intuitively, given points p_i and p_j, if $R \nsubseteq h^-_{j,i}$, 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 such points exist, a naive idea is to check each $p_i \in D$ using Lemma 5.5. However, it can be time-consuming if D is large. To reduce the burden, we randomly sample a utility vector u in R and check whether each of the top-k points w.r.t. u is a qualified point to be returned 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 performance is summarized as follows.

Theorem 5.6. *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 at 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 at each round. Since there are at most n partitions, there is one partition left after $O(n)$ rounds and stopping condition 1 is satisfied. If the selected hyperplane can divide C into equal halves without any intersecting partitions at each round, we can prune half partitions. After $O(\log n)$ rounds, there exists only one partition and the algorithm terminates. □

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. Different from *HD-PI*, we only maintain a polyhedron R, called utility range, in the utility space, which contains the user's utility vector. Initially, R is the utility space, i.e., $R = \{u \in \mathbb{R}^d_+ | \sum_{i=1}^d u[i] = 1\}$. At each round, based on the user preference on p_i and p_j, we update R to be $R \cap h^+_{i,j}$ (or $h^-_{i,j}$).

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
$C \leftarrow \{\Theta_1, \Theta_2, ..., \Theta_m\}$
Initial 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 of $h_{i,j}$ 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

5.3.2 Stop 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 for *RH*.

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 order between p_i and p_j w.r.t. u_0 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 immediately and arbitrarily return one of the top-k points.

5.3.3 Point Selection. Recall that at 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, at each round, we select two points p_i and p_j whose 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 $R_c = \frac{\sum_{v \in \mathcal{V}_R} v}{|\mathcal{V}_R|}$ where $\mathcal{V}_R$ is the set of vertices of R. We select points p_i and p_j whose hyperplane $h_{i,j}$ is the closest to R_c.

However, to determine the hyperplane with the minimum distance to R_c, we need to check $O(n^2)$ hyperplanes, which could be time consuming if n is large. Thus, the following the strategy is used to reduce the number of hyperplanes to be considered.

We first 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 $H_i = \{h_{i,j} | \forall j, j < i\}$. We start the hyperplane selection from H_2 (since $H_1 = \emptyset$) and move to the next hyperplane set if the current H_i do not contain any hyperplane intersecting R since these hyperplanes cannot be used to make R smaller. At 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 *RH* is shown in Algorithm 4 and 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, we need to ask $O(d \log n)$ questions in expectation and then prove that in general case, the expected number of questions asked is $O(cd \log n)$, where $c > 1$ is a constant. □

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
Initialize the utility range R
Initialize sets H_i, where $i = 2, ..., n$
while $\exists h_{i,j} \in H_i$ *intersecting* R **do**
  Find $h_{i,j} \in H_i$ intersecting R with the min-distance to R_c
  Display points p_i and p_j of $h_{i,j}$ to the user
  Based on the user feedback, update R
  if *the stopping condition is satisfied* **then**
    return a point satisfying Lemma 5.5 or an arbitrary top-k points if the ranking of D is known

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

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, 18, 29, 38]. Specifically, the synthetic datasets are *anti-correlated* [11] 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 were deleted. Six attributes are used to represent the performance of each player. For all of the datasets, each dimension is normalized to $(0, 1]$. Note that in existing studies [12, 38], they preprocessed datasets to contain skyline points only (all possible top-1 points for any utility function) since they look for (close to) top-1 points. Consistent with their setting, we preprocessed all of the datasets to include k-skyband points (which are all possible top-k points for any utility function) [18] 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 are 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* [38], *Hull* [38], *Active-Ranking* [19], *UtilityApprox* [27], *UH-Random* [38], *UH-Simplex* [38] and *Preference-Learning* [32]. 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, namely *Median-Adapt* and *Hull-Adapt*, by modifying their *point deletion condition* to that the point cannot be one of the user's top-k (originally top-1) points according to the learnt information, and their *stopping condition* to that 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* learns the user's utility vector by interacting with the user and terminates when the evaluating tuples criterion regret ratio [27] 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, it must be one of the top-k points.
- Algorithms *UH-Simplex* and *UH-Random* return one point, which either has the largest utility w.r.t. the user's utility vector or has a regret ratio satisfying a given threshold ϵ, by interacting with the user. We set ϵ the same as that in algorithm *UtilityApprox*. Besides, we create another version, namely *UH-Simplex-Adapt* and *UH-Random-Adapt*, by modifying their point deletion condition and their stopping condition the same as algorithms *Median-Adapt* and *Hull-Adapt*.
- Algorithm *Preference-Learning* learns the user's utility vector by interacting with the user. We set the learnt utility vector error threshold ϵ to $1 - 10^{-6}$ and arbitrarily return one of the top-k points w.r.t. the learnt utility vector. Note that although $\epsilon = 0.95$ in [32], the returned point is not always among the user's top-k points if $\epsilon = 0.95$. Thus, we set $\epsilon = 1 - 10^{-6}$, since it is close to the theoretical optimum according to the experimental results in [32].

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 be 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 user. Each algorithm were conducted 10 times with different generated user's utility vectors and the average performance was reported.

In the following, the setting of our algorithms is studied in Section 6.1. The performance of algorithms on 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 performances, 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 relationships between hyperplanes and partitions, which 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 ratio 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 showed 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 [12, 14], 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 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 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 are also involved by setting $d = 2$ (Since the performance of d-dimensional algorithms are 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 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 condition is easier to achieve, the adaptation on 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* with the existing d-dimensional algorithms on 2-dimensional and 4-dimensional datasets, respectively. Algorithm *Active-Ranking* requires the largest number of questions with a long execution time, e.g., 1000 questions and 100 seconds on the 4-dimensional dataset, since it learns the ranking of all points. It even runs gradually slower and asks more questions when k increases,

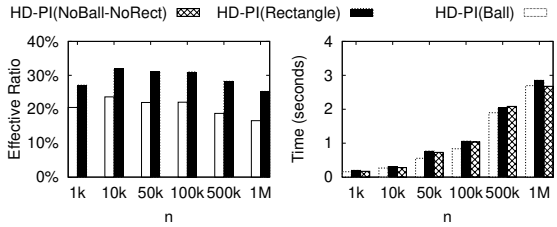


Figure 5: Bounding Strategies

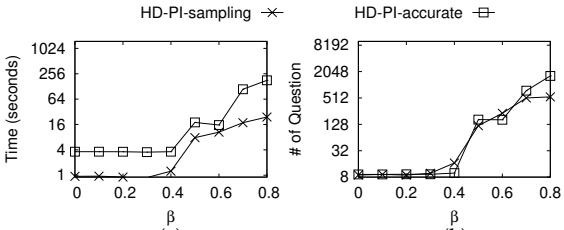


Figure 6: Performances on different β

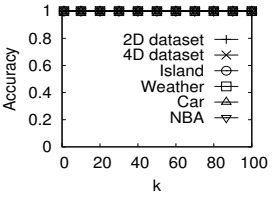


Figure 7: Accuracy

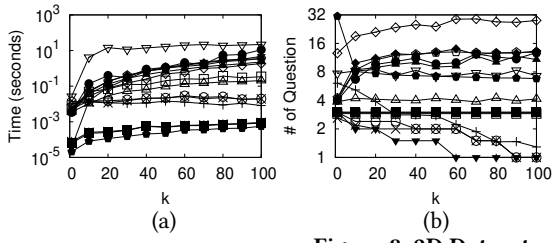
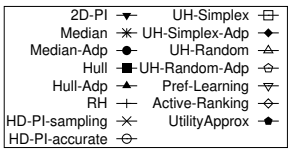


Figure 8: 2D Dataset



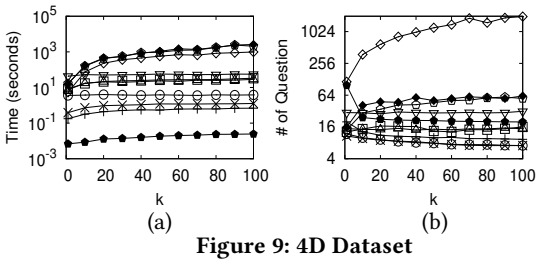


Figure 9: 4D Dataset

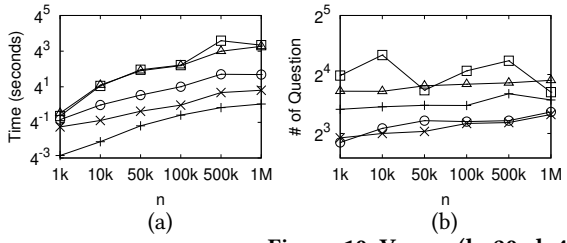
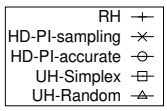


Figure 10: Vary n (k=20, d=4)



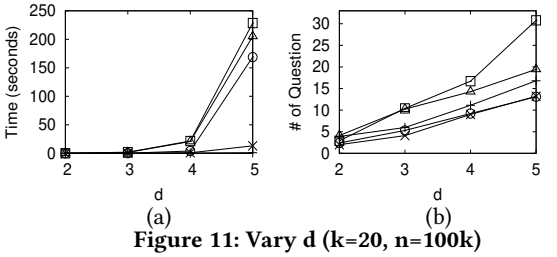


Figure 11: Vary d (k=20, n=100k)

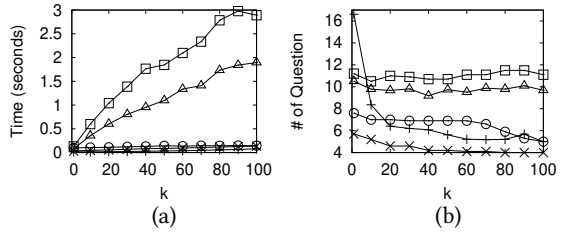
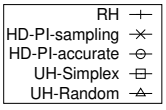


Figure 12: Weather



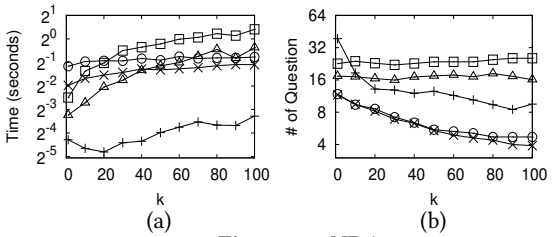


Figure 13: NBA

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 than that of our algorithms. Specifically, on the 2-dimensional dataset, *UH-Random* and *UH-Simplex* take 2-10 times as much time as our algorithms, and the execution time required by *Preference-Learning* is about 2-3 orders of magnitude more than that required by 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, *UH-Random* and *UH-Simplex* performs well when $k = 1$, since they are designed specially for returning the top-1 point. However, the number of questions asked almost remain unchanged when k increases. Algorithm *Preference-Learning* also asks nearly the same number of questions with different values of k. 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 large execution time and more 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 *UtilityApprox*, our algorithms *RH* and *HD-PI* take the least time to obtain the desired point. 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 algorithms. Moreover, except *UtilityApprox*, there is no existing algorithm whose number of questions asked decreases significantly when k increases, while our algorithms *HD-PI* and *RH* could 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 algorithlms also ask at least 2 fewer questions than others for arbitrary n. In Figure 11, we evaluated the scalability on the dimensionality d. Compared with 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 with 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 datasets *Weather* (with the largest data size) and *NBA* (with the largest dimensionality) in Figures 12 and 13, respectively. The results on datasets *Island* and *Car* can be found in the technical report [5]. 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 datasets *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 dataset *Weather* when $k \geq 50$, while the others take more than 1 seconds.

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 feedbacks. Following the setting in [32, 38], 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). For algorithm *Preference-Learning*, it 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 [32] 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*, since the user's utility vector is unknown, the threshold ϵ for regret ratio cannot be set in the way described before. Thus, we simply set $\epsilon = 0$ and this guarantees that one of the top-20 cars is returned.

Each algorithm is 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 the participants. Specifically, since participants sometimes gave the same score for the 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 there are 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 p of partition Θ is added to V_d. Then, partition Θ is divided into several smaller partitions Θ_i added to C such that $\Theta_i = \{u \in \Theta | u \in h^+_{i,j}, \forall p_j \in V / \{p_i\}\}$, where V contains all 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

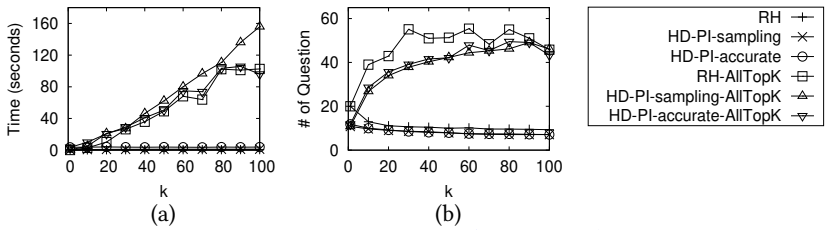


Figure 14: 4D Dataset (Motivation)

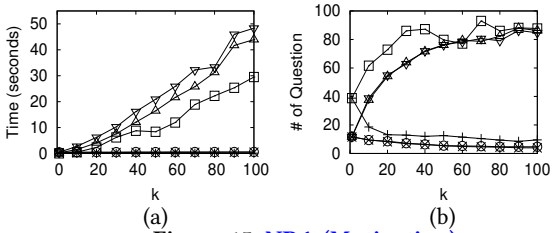


Figure 15: NBA (Motivation)

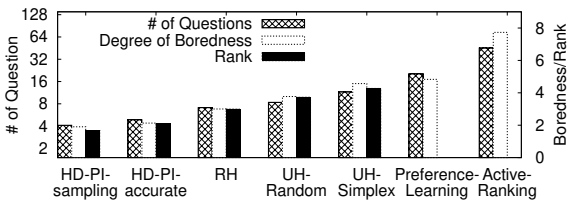


Figure 16: User Study

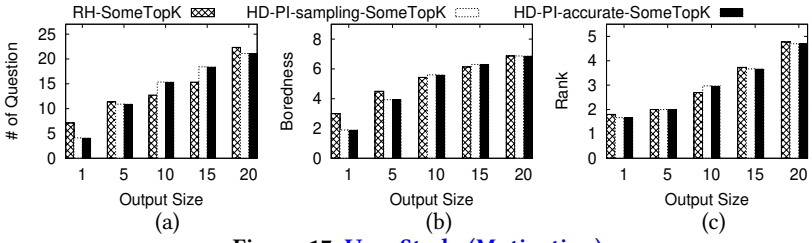


Figure 17: User Study (Motivation)

synthetic dataset and the *NBA* dataset (with the largest dimensionality). The results on the other datasets can be found in the technical report [5]. It can be seen that returning all the top-k points costs a lot. It runs $1-2$ orders of magnitude longer and requires 4-10 times more questions than returning one of the top-k points when $k \geq 20$. For example, *RH* asks about 9 questions within 0.08 seconds to return one of the top-k points, while its modified version needs around 86 questions within 22 seconds on dataset *NBA* 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 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 rank of the 5 cases given by participants. 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% relationships between hyperplanes and partitions. In summary, *2D-PI* asks the least number of questions in a 2-dimensional space with 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 seconds, while *UH-Random* and *UH-Simplex* needs 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 recommending 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 asks an asymptotically number of questions. 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, respectively. In particular, for *HD-PI*, we propose two versions: sampling and accurate, which targets 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.

REFERENCES

[1] [n.d.]. https://www.alchemer.com/resources/blog/how-many-survey-questions/
[2] [n.d.]. https://www.questionpro.com/blog/optimal-number-of-survey-questions/
[3] Pankaj K. Agarwal, Sariel Har-Peled, and Kasturi R. Varadarajan. 2004. Approximating extent measures of points. *J. ACM* 51, 4 (2004), 606–635. https://doi.org/10.1145/1008731.1008736
[4] Abolfazl Asudeh, Azade Nazi, Nan Zhang, Gautam Das, and H. V. Jagadish. 2019. RRR: Rank-Regret Representative. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 263–280.
[5] Anonymous Author(s). 2020. Interactive Search for One of the Top-k (technical report). In *Anonymous Link*.
[6] Wolf Tilo Balke, Ulrich Güntzer, and Christoph Lofi. 2007. Eliciting matters - Controlling skyline sizes by incremental integration of user preferences. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4443 LNCS (2007), 551–562.
[7] Wolf-Tilo Balke, Christoph Lofi, and Ulrich Güntzer. 2007. User Interaction Support for Incremental Refinement of Preference-Based Queries. *1st International IEEE Conference on Research Challenges in Information Science (RCIS)* January (2007).
[8] 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.
[9] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. 2014. Domination in the Probabilistic World: Computing Skylines for Arbitrary Correlations and Ranking Semantics. *ACM Trans. Database Syst.* 39, 2 (May 2014).
[10] Ilaria Bartolini, Paolo Ciaccia, and Florian Waas. 2001. FeedbackBypass: A new approach to interactive similarity query processing. In *VLDB*. 201–210.
[11] Stephan Borzsonyil and Konrad Stocker. 2001. The Skyline Operator. *Proceedings of the International Conference on Data Engineering* (2001), 421–430.
[12] 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. *Leibniz International Proceedings in Informatics, LIPIcs* 68, 23 (2017), 1–23.
[13] Barbara Catania and Lakhmi C. Jain. 2014. *Advanced Query Processing: Volume 1 Issues and Trends*. Springer Publishing Company, Incorporated.
[14] Sean Chester, Alex Thomo, S. Venkatesh, and Sue Whitesides. 2014. Computing k-regret minimizing sets. *Proceedings of the VLDB Endowment* 7, 5 (2014), 389–400.
[15] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. 2008. *Computational geometry: Algorithms and applications*. Springer Berlin Heidelberg. 1–386 pages. https://doi.org/10.1007/978-3-540-77974-2
[16] H. Edelsbrunner and E. Welzl. 1986. Constructing Belts in Two-Dimensional Arrangements With Applications. *SIAM J. Comput.* 15, 1 (1986), 271–284. https://doi.org/10.1137/0215019
[17] Brian Eriksson. 2013. Learning to top-K search using pairwise comparisons. *Journal of Machine Learning Research* 31 (2013), 265–273.
[18] 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. https://doi.org/10.1016/j.ins.2014.08.052
[19] Kevin G. Jamieson and Robert D. Nowak. 2011. Active ranking using pairwise comparisons. *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011, NIPS 2011* 1 (2011), 1–17. arXiv:arXiv:1109.3701v2
[20] Bin Jiang, Jian Pei, Xuemin Lin, David W. Cheung, and Jiawei Han. 2008. Mining preferences from superior and inferior examples. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2008), 390–398. https://doi.org/10.1145/1401890.1401940
[21] Jongwuk Lee, Gae won You, and Seung won Hwang. 2009. Personalized top-k skyline queries in high-dimensional space. *Information Systems* 34, 1 (2009), 45–61. https://doi.org/10.1016/j.is.2008.04.004
[22] Jongwuk Lee, Gae Won You, Seung Won Hwang, Joachim Selke, and Wolf Tilo Balke. 2012. Interactive skyline queries. *Information Sciences* 211 (2012), 18–35. https://doi.org/10.1016/j.ins.2012.04.007
[23] 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* (Geneva, Switzerland) *(SIGIR '10)*. Association for Computing Machinery, New York, NY, USA, 904.
[24] Lucas Maystre and Matthias Grossglauser. 2017. Just sort it! A simple and effective approach to active preference learning. *34th International Conference on Machine Learning, ICML 2017* 5 (2017), 3640–3656. arXiv:1502.05556
[25] metode penelitian Nursalam, 2016 and A.G Fallis. 2013. *Hand book of computational geometry*. Vol. 53. 1689–1699 pages. arXiv:arXiv:1011.1669v3
[26] Kyriakos Mouratidis and Bo Tang. 2018. Exact processing of uncertain topk queries in multicriteria settings. *Proceedings of the VLDB Endowment* 11, 8 (2018), 866–879. https://doi.org/10.14778/3204028.3204031
[27] Danupon Nanongkai, Ashwin Lall, Atish Das Sarma, and Kazuhisa Makino. 2012. Interactive regret minimization. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2012), 109–120.
[28] Danupon Nanongkai, Atish Das Sarma, Ashwin Lall, Richard J. Lipton, and Jun Xu. 2010. Regret-minimizing representative databases. *Proceedings of the VLDB Endowment* 3, 1 (2010), 1114–1124. https://doi.org/10.14778/1920841.1920980
[29] Dimitris Papadias, Greg Fu, and Bernhard Seeger. 2005. *Progressive Skyline Computation in Database Systems*. Vol. 30. 41–82 pages. https://doi.org/10.1145/1061318.1061320
[30] Peng Peng and Raymond Chi Wing Wong. 2014. Geometry approach for k-regret query. *Proceedings - International Conference on Data Engineering* (2014), 772–783. https://doi.org/10.1109/ICDE.2014.6816699
[31] Peng Peng and Raymond Chi Wing Wong. 2015. K-hit query: Top-k query with probabilistic utility function. *Proceedings of the ACM SIGMOD International Conference on Management of Data* 2015-May (2015), 577–592. https://doi.org/10.1145/2723372.2723735
[32] Li Qian, Jinyang Gao, and H. V. Jagadish. 2015. Learning user preferences by adaptive pairwise comparison. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1322–1333. https://doi.org/10.14778/2809974.2809992
[33] Melanie Revilla and Carlos Ochoa. 2017. Ideal and Maximum Length for a Web Survey. *International Journal of Market Research* 59, 5 (2017), 557–565. https://doi.org/10.2501/IJMR-2017-039
[34] Gerard Salton. 1989. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley Longman Publishing Co., Inc., USA.
[35] Thomas Seidl and Hans Peter Kriegel. 1997. Efficient user-adaptable similarity search in large multimedia databases. *Proceedings of the 23rd International Conference on Very Large Databases, VLDB 1997* (1997), 506–515.
[36] Mohamed A. Soliman and Ihab F. Ilyas. 2009. Ranking with uncertain scores. *Proceedings - International Conference on Data Engineering* 2 (2009), 317–328. https://doi.org/10.1109/ICDE.2009.102
[37] Min Xie and Raymond Chi-wing Wong. 2018. Efficient k-Regret ery Algorithm with Restriction-free Bound for any Dimensionality. (2018).
[38] Min Xie, Raymond Chi Wing Wong, and Ashwin Lall. 2019. Strongly truthful interactive regret minimization. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2019), 281–298. https://doi.org/10.1145/3299869.3300068
[39] 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. https://doi.org/10.1007/s00778-019-00570-z
[40] Jiping Zheng and Chen Chen. 2020. Sorting-Based Interactive Regret Minimization. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12318 LNCS (2020), 473–490. https://doi.org/10.1007/978-3-030-60290-1_36 arXiv:2006.10949

Appendix A EXPERIMENTS

A.1 Performance on Real Datasets

The results on dataset *Island* (2D) is shown in Figure 18. All the algorithms run efficiently. They are finished within 4 seconds. For the number of questions asked, our algorithm *2D-PI* performs the best w.r.t. arbitrary parameter k. Specifically, it only needs about 1 question as long as k is larger than 10. Algorithm *HD-PI* requires nearly half of the questions asked by the other existing algorithms and *RH* asks fewer questions than the other existing algorithms when $k \geq 20$. Figure 19 shows the performance on dataset *Car*. Both *HD-PI* and *RH* only ask nearly half the number of questions required by *UH-Random* and *UH-Simplex* when $k \geq 50$. As for the execution time, *RH* and *HD-PI* take at most 0.22 seconds, while existing algorithms need at least 2 seconds when $k \geq 60$.

A.2 User Effort Comparison

We compared the user's effort required by returning one of the top-k tuples and returning all the top-k tuples. The performance on the 2-dimensional synthetic dataset, the dataset *Island, Car* and *Weather* are shown in Figure 20, 21, 22 and 23, respectively. It can be seen that returning all the top-k points takes user's effort largely. It runs several orders of magnitude longer and requires a few times

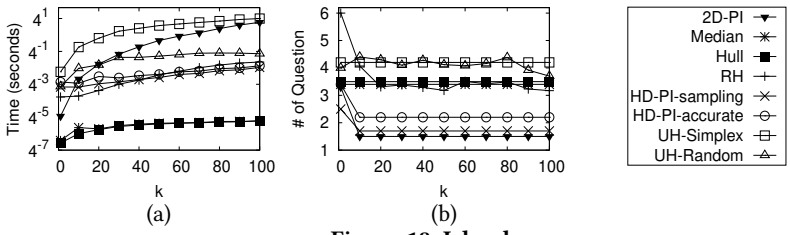


Figure 18: Island

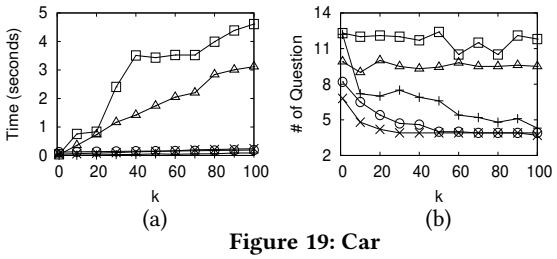


Figure 19: Car

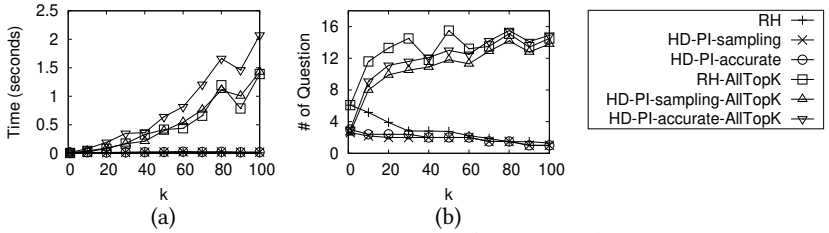


Figure 20: 2D Dataset (Motivation)

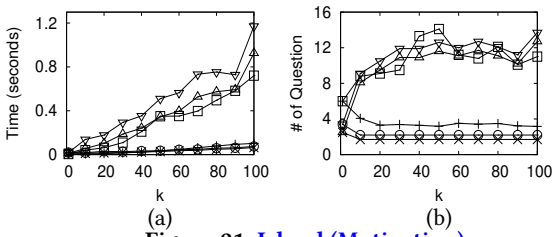


Figure 21: Island (Motivation)

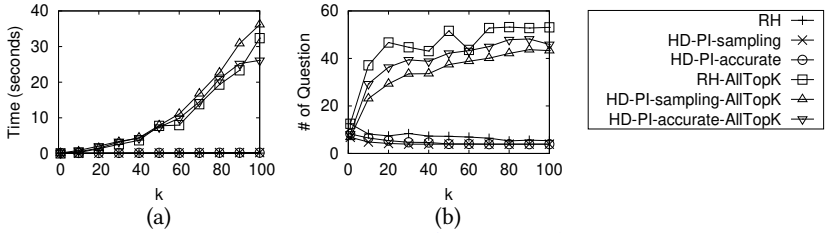


Figure 22: Car (Motivation)

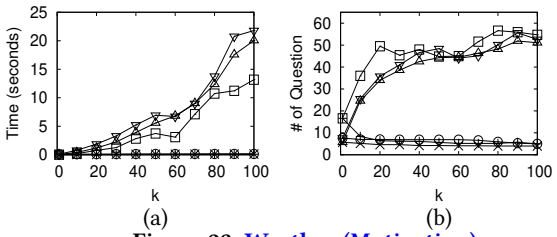


Figure 23: Weather (Motivation)

Notation	Meaning
D	The database
d	Number of attributes/dimensions
n	Number of points
u	The utility vector
Θ_i	The i-th partition
ℓ_i	The corresponding line segment of p_i
l_i	The leftmost utility vector of Θ
r_i	The rightmost utility vector of Θ
q_i	A point which is among the top-k points w.r.t any utility vector in Θ_i
$\mathcal{Q}$	A data structure queue used to store points
$\mathcal{H}$	A data structure min-heap used to store the intersection ℓ_i, ℓ_j of the neighboring pair $p_i$and p_j in $\mathcal{Q}$
C	The set containing all the partitions
$\mathcal{E}$	The set containing all the points q_i
R	The utility range
V	The set containing all the convex points
$h_{i,j}$	A hyperplane constructed by points p_i, p_j

Table 5: Summary of Frequent Used Notations

more questions than returning one of the top-k points when $k \geq 20$. For example, on the 2-dimensional dataset, *RH* and *HD − PI* ask about 1 question within 0.02 seconds to return one of the top-100 points, while their modified version need around 14 questions in at least 1.4 seconds. On the real datasets *Weather*, *RH* and *HD − PI* ask about 4 questions within 0.24 seconds to return one of the top-100 points, while they need more than 50 questions in at least 13 seconds to return all the top-100 points.

Appendix B PROOFS

Proof of Theorem 3.2. Consider a dataset D, where $\forall p \in D$, there are $k-1$ points $q \in D/p$ such that $\forall i \in [1,d], p[i] = q[i]$ and p has the maximum utility w.r.t some utility vectors in the utility space. Any algorithm that aims to identify all the points with pairwise comparison must be in the form of a binary tree. Each leaf node contains k same points and each internal node corresponds to a question asked user. Since there are $\frac{n}{k}$ leaves, the height of the tree is $\Omega(\log_2 \frac{n}{k})$. That is any algorithm requires to ask $\Omega(\log_2 \frac{n}{k})$ questions to determine one of the top-k points in the worst case. □

Proof of Lemma 4.3. 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 partition obtained by our algorithm. In our algorithm, we continue constructing the first partition Θ_1 as long as there is a point which is among the top-k points w.r.t. any utility vector in $[0, u_t[1]]$, where $u_t[1]$ is the $u[1]$-value of the utility line t. Since there exist a point q'_1 which is among the top-k points w.r.t. any utility vector in $\Theta'_1 = [0, r'_1]$ (optimal case), Θ_1 (our algorithm) will not stop at a utility vector

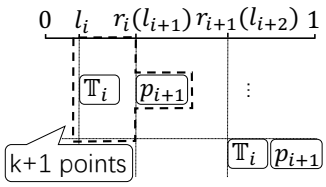


Figure 24: Partition Θ_i and Θ_{i+1}

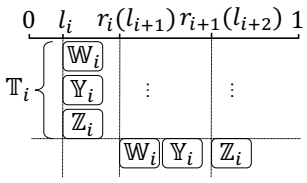


Figure 25: Partition Θ_i and Θ_{i+1}

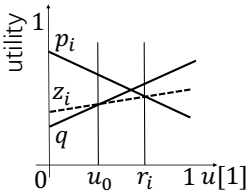


Figure 26: Z_i Case

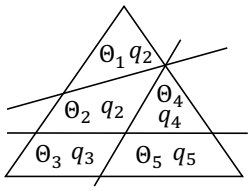


Figure 27: Partition

u such that $u[1] < r_1'$, i.e., $r_1 \geq r_1'$. We say that that our algorithm does not end the first partition "earlier" than the optimal case.

Consider the second partition. Since $l_2' = r_1'$ and $l_2 = r_1$, $l_2 \geq l_2'$. We say that our algorithm does not start the second partition "earlier" than the optimal case. For Θ_2', there exists a point q_2' which is among the top-k points w.r.t. any utility vector in $[l_2', r_2']$. Since $l_2 \geq l_2'$, q_2 must be among the top-k points w.r.t. any utility vector in $[l_2, r_2']$. Because Θ_2 will not be ended as long as there exist a point which is among the top-k points w.r.t any utility vector in $[l_2, u_t[1]]$, where u_t is the $u[1]$-value of the utility line t, Θ_2 will not stop at a utility vector u such that $u[1] < r_2'$, i.e., $r_2 \geq r_2'$. Similarly, our algorithm does not end the second partition "earlier" than the optimal case.

Same as the analysis on the second partition, for any partition Θ_i, where $i \geq 2$, our algorithm does not start and end it "earlier" than the optimal case, i.e., $l_i \geq l_i'$ and $r_i \geq r_i'$, where $i \geq 2$. This means the number of partitions obtained by our algorithm will not be more than that of the optimal case. □

Proof of Lemma 4.5. We first show that the number of partitions obtained by *utility space partitioning* in the worst case is $\lceil \frac{2n}{k+1} \rceil$, and then discuss the number of questions asked is $O(\log_2 \lceil \frac{2n}{k+1} \rceil)$.

For each partition $\Theta_i = [l_i, r_i]$, there is a set $\mathbb{Y}_i$ of points which are among the top-k points w.r.t. any utility vector in Θ_i. Denote by p_i the point in $\mathbb{Y}_i$ which ranks the highest (has the largest utility) w.r.t. utility vector $(l_i, 1 - l_i)$, and represent by l_i' the value closely following l_i.

Consider any pair of partitions Θ_i and Θ_{i+1}, where $j = 1, 3, 5, \ldots$. We claim that (1) point p_i will not be among the top-k points w.r.t. any utility vector in $[l_{i+1}', 1]$; (2) a set $\mathbb{T}_i$ of the top-k points w.r.t. utility vector $(l_i, 1-l_i)$ will not be among the top-k points w.r.t. any utility vector in $[l_{i+2}', 1]$. This means after each pair of partitions Θ_i and Θ_{i+1}, there are at least $k+1$ points (i.e., points in $\mathbb{T}_i$ and p_{i+1} shown in Figure 24). which will not be among the top-k points w.r.t. any utility vector in $[l_{i+2}', 1]$. Since each partition corresponds to a set $\mathbb{T}_i$ and there are n points in the dataset, there are at most $\lceil \frac{n}{k+1} \rceil$ pairs of partitions. Therefore, there will be at most $\lceil \frac{2n}{k+1} \rceil$ partitions in the worst case.

The points in $\mathbb{T}_i$ (set of top-k points w.r.t. utility vector $(l_i, 1-l_i)$) are divided into three kinds of sets, shown in Figure 25, denoted by $\mathbb{W}_i$, $\mathbb{Y}_i$ and $\mathbb{Z}_i$ respectively: (1) points in $\mathbb{W}_i$ rank higher (has larger utility) than p_i w.r.t. utility vector $(l_i, 1 - l_i)$; (2) points in $\mathbb{Y}_i$ are among the top-k points w.r.t. any utility vector in Θ_i; (3) $\mathbb{Z}_i = \mathbb{T}_i - (\mathbb{W}_i \cup \mathbb{Y}_i)$.

In the following, we are to prove three claims. Based on Claim 1, since $p_i \in \mathbb{Y}_i$, point p_i will not be among the top-k points w.r.t. any utility vector in $[l_{i+1}', 1]$. According to Claim 1, 2 and 3, since $\mathbb{T}_i = \mathbb{Y}_i \cup \mathbb{W}_i \cup \mathbb{Z}_i$, points in $\mathbb{T}_i$ will not be among the top-k points w.r.t. any utility vector in $[l_{i+2}', 1]$.

Claim 1. *Points in $\mathbb{Y}_i$ will not be among the top-k points w.r.t. any utility vector in $[l_{i+1}', 1]$.*

Proof. Each point y_i in $\mathbb{Y}_i$ is not one of the top-k points w.r.t. utility vector $(l_{i+1}', 1 - l_{i+1}')$. Otherwise $l_{i+1}' \in \Theta_i$. There must be a set $\mathbb{K}$ of k points which rank higher than y_i w.r.t. the utility vector $(l_{i+1}', 1 - l_{i+1}')$. Those k points should rank lower than y_i w.r.t utility vector $(l_i, 1 - l_i)$. If there is a point in $\mathbb{K}$ which ranks higher than y_i w.r.t. utility vector $(l_i, 1 - l_i)$, it must be among the top-k points w.r.t. any utility vector in $[l_i, l_{i+1}']$, i.e., $l_{i+1}' \in \Theta_i$, which is inconsistent with the definition of Θ_i.

Since points in $\mathbb{K}$ rank lower than y_i w.r.t utility vector $(l_i, 1 - l_i)$ and rank higher than y_i w.r.t. utility vector $(l_{i+1}', 1 - l_{i+1}')$, their ranking must change with that of y_i at some utility vectors in Θ_i. Note that for any pair of points, their ranking changes at most once when the $u[i]$-value of the utility line t varies from 0 to 1. Points in $\mathbb{K}$ will always rank higher than y_i for any utility vector in $[l_{i+1}', 1]$. Thus, point y_i will not be one of the top-k points w.r.t. any utility vector in $[l_{i+1}', 1]$. □

Claim 2. *Points in $\mathbb{W}_i$ will not be among the top-k points w.r.t. any utility vector in $[l_{i+1}', 1]$*

Proof. For each $w_i \in \mathbb{W}_i$, it should rank lower than p_i w.r.t. utility vectors $(l_{i+1}, 1 - l_{i+1})$. Otherwise, w_i ranks higher than p_i w.r.t any utility vector in Θ_i, which is inconsistent with the definition of p_i. Since w_i ranks higher than p_i w.r.t. utility vector $(l_i, 1 - l_i)$ and ranks lower than p_i w.r.t. utility vectors $(l_{i+1}, 1 - l_{i+1})$, its ranking must change with that of p_i at a utility vector in Θ_i. Note that the ranking of each pair of points changes at most once. Point w_i must rank lower than p_i w.r.t. any utility vector in $[l_{i+1}, 1]$. Based on Claim 1, since $p_i \in \mathbb{Y}_i$ will not be one of the top-k points w.r.t. any utility vector in $[l_{i+1}', 1]$, point $w_i \in \mathbb{W}_i$ will not be among the top-k points w.r.t. any utility vector in $[l_{i+1}', 1]$ either. □

Claim 3. *Points in $\mathbb{Z}_i$ will not be among the top-k points w.r.t. any utility vector in $[l_{i+2}', 1]$*

Proof. Each point $z_i \in \mathbb{Z}_i$ ranks lower than p_i w.r.t. utility vector $(l_i, 1 - l_i)$ and is not among the top-k points w.r.t. some utility vectors in Θ_i. Here is a case shown in Figure 26. The dotted line represents z_i whose ranking is k w.r.t. utility vector $(l_i, 1 - l_i)$. Point z_i is not among the top-k points w.r.t. any utility vector u such that $u_0 < u[1] < r_i$ (where $l_i < u_0 < r_i$), and then becomes one of the top-k points again at the utility vector $(r_i, 1 - r_i)$.

Since z_i is not among the top-k points w.r.t. some utility vectors in Θ_i, there must exist a set E_i of points which are not among the top-k points w.r.t. utility vector $(l_i, 1-l_i)$ and become among the top-k points w.r.t. some utility vectors in Θ_i. This means the ranking of points in E_i change with that of z_i w.r.t. some utility vectors in Θ_i. Since the ranking of any pair of points changes at most once, points in E_i must rank higher than z_i w.r.t. any utility vector in $[r_i, 1]$. Denote by e_i the point in E_i which ranks the highest (has the largest utility) w.r.t. utility vector $(l_i, 1-l_i)$.

Consider the point p_{i+1} which ranks the highest w.r.t. utility vector $(l_{i+1}, 1-l_{i+1})$ among $\mathbb{Y}_{i+1}$. There are at least k points which rank higher than p_{i+1} w.r.t. any utility vector in $[l'_{i+2}, 1]$ (l'_{i+2} closely follows l_{i+2}). There are two kinds of relationship between e_i and p_i.

- Point e_i is p_{i+1}. Then, there are at least k points which rank higher than e_i w.r.t. any utility vector in $[l'_{i+2}, 1]$.
- Point e_i is not p_{i+1}. (1) If p_{i+1} ranks lower than e_i w.r.t utility vector $(l_{i+1}, 1-l_{i+1})$, the ranking of p_{i+1} must change with that of e_i at a utility vector in $[l_{i+1}, r_{i+1}]$. Otherwise e_i should be in $\mathbb{Y}_i$ and ranks higher than p_{i+1} w.r.t. utility vector $(l_{i+1}, 1-l_{i+1})$, which violates the definition of p_{i+1}. Since the ranking of each pair of points changes at most once, p_{i+1} must rank higher than e_i w.r.t. any utility vector in $[l'_{i+2}, 1]$. (2) If p_{i+1} ranks higher than e_i w.r.t. $(l_{i+1}, 1-l_{i+1})$, based on the definition of e_i (e_i ranks the highest w.r.t. utility vector $(l_i, 1-l_i)$ among E_i), the ranking of e_i and p_{i+1} must change at a utility vector in $[l_i, r_i]$. Since the ranking of each pair of points changes at most once, p_{i+1} must rank higher than e_i w.r.t. any utility vector in $[l'_{i+2}, 1]$.

Based on Claim 1, points $p_{i+1} \in \mathbb{Y}_{i+1}$ will not be among the top-k points w.r.t. any utility vector in $[l'_{2+1}, 1]$. Since p_{i+1} must rank higher than e_i w.r.t. any utility vector in $[l'_{i+2}, 1]$, e_i will not be among the top-k points w.r.t. any utility vector in $[l'_{2+1}, 1]$. Due to the previous illustration that points in E_i must rank higher than z_i w.r.t. any utility vector in $[r_i, 1]$, z_i will not be one of the top-k points w.r.t. any utility vector in $[l'_{i+2}, 1]$. □

Now we have shown that there will be at most $\lceil \frac{2n}{k+1} \rceil$ partitions. Since the number of partitions can be removed by half at each interactive round, we can locate the user's utility vector in a single partition after $O(\log_2 \lceil \frac{2n}{k+1} \rceil)$ rounds. □

Proof of Lemma 5.1. Give a polyhedron $\mathcal{P}$, based on the definition of bounding ball and bounding rectangle, $\mathcal{P} \subseteq \mathbb{B}(\mathcal{P})$ and $\mathcal{P} \subseteq \mathbb{R}(\mathcal{P})$. If $\mathbb{B}(\mathcal{P}) \subseteq h^+_{i,j}$ or $\mathbb{R}(\mathcal{P}) \subseteq h^+_{i,j}$, then $\mathcal{P} \subseteq h^+_{i,j}$. □

Proof of Theorem 5.2. We first show in Lemma B.1 the relationship between each partition and its associated point (one of the top-k points w.r.t. any utility vector in the partition) and then prove that the problem utility space partitioning is NP-hard by utilizing the problem *rank-regret representative (RRR)* [4].

Lemma B.1. *If the utility space is divided into the least number of partitions, different partitions are associated with different points.*

Proof. Suppose the utility space is divided into m partitions. Each partition Θ_i, where $i \in [1, m']$, is associated with the same point $q_{m'}$ and each partition Θ_i, where $i \in [m'+1, m]$, is associated with a point different q_i. For example, shown in Figure 27, partitions Θ_1 and Θ_2 are associated with point q_2, and the other partitions Θ_3, Θ_4 and Θ_5 are associated with q_3, q_4 and q_5 ,respectively.

Denote the set $\mathbb{P} = \{q_{m'}, q_{m'+1}, ..., q_m\}$. For each point $q_x \in \mathbb{P}$, we create a partition $\Theta_x = \{u \in \mathbb{R}^d_+ | u \in h^+_{x,j}, q_j \in \mathbb{P}/q_x\}$ in the utility space, such that $\forall u \in \Theta_x$, point q_x ranks the highest (has the largest utility) among $\mathbb{P}$ w.r.t. u. Note that $\forall u \in \Theta_x$, u must belong to a partition Θ_i ($i \in [1, m]$) associated with a point in $\mathbb{P}$ which is one of the top-k points w.r.t. u. Since q_x ranks the highest among $\mathbb{P}$ w.r.t. u, q_x must be one of the top-k points w.r.t. u. In this way, the utility space is divided into $m - m'$ partitions. Each partition Θ_x is associated with a point q_x in $\mathbb{P}$ which is among the top-k points w.r.t. any utility vector in Θ_x. This contradicts to the assumption that the utility space is divided into the least number of partitions (i.e., m partitions). □

The problem *RRR* is that given a database D and a set of utility vectors $\mathcal{F}$, it finds the smallest set $S \subseteq D$ such that for any utility vector $u \in \mathcal{F}$, there is a point in S which is one of the top-k points w.r.t. u. [4] proves that the problem *RRR* is NP-hard. In the following, we show that if $\mathcal{F}$ is the utility space, an instance of *RRR* has a set S of size s if and only if the corresponding instance of utility space partitioning divides the utility space into s partitions.

IF: Suppose the utility space is divided into s (least number) partitions, each of which Θ_i is associated with a point q_i ranking among top-k points w.r.t any utility vector in Θ_i. Then *RRR* has a set $S = \{q_1, q_2, ..., q_s\}$. For each utility vector u in the utility space, it must belong to a partition Θ_i and the associated points $q_i \in S$ is one of the top-k points w.r.t. it.

ONLY IF: Suppose *RRR* has a set S of size s such that for any utility vector u in the utility space, there is a point in S which is one of the top-k points w.r.t. u. For each point $q_i \in S$, we create a partition $\Theta_i = \{u \in \mathbb{R}^d_+ | u \in h^+_{i,j}, q_j \in \mathbb{P}/q_i\}$ in the utility space such that point q_i ranks the highest among $\mathbb{P}$ w.r.t any utility vector in Θ_i. Note that $\forall u \in \Theta_i$, there is a point in S which is among the top-k points w.r.t. u. Since q_i ranks the highest among $\mathbb{P}$ w.r.t. u, it must be one of the top-k points w.r.t. any utility vector in Θ_i. In this way, the utility space is divided into s partitions, each of which is associated with a point which is one of the top-k points w.r.t. any utility vector in the partition. □

Proof of Lemma 5.5. For each point $p_j \in D/p_i$, if the utility range $R \subseteq h^-_{j,i}$, we have $\forall u \in R, u \cdot (p_j - p_i) < 0$. The utility of p_j must be smaller than that of p_i w.r.t. any utility vector $u \in R$. Suppose$|\{p_j \in D | R \nsubseteq h^-_{j,i}\}| < k$. There must be more than $n-k$ points $p_j \in D/p_i$ such that $\forall u \in R, u \cdot (p_j - p_i) < 0$, i.e., there are more than $n-k$ points whose utility is smaller than that of p_i w.r.t. any utility vector $u \in R$. Thus, p_i must be one of the top-k points w.r.t. any utility vector $u \in R$. □

Proof of Theorem 5.7. We consider the algorithm terminates under the Stopping Condition 3 shown in Section 5.3.2, since the Stopping Condition 2 in Section 5.2.3 must be satisfied if we meet the Stopping Condition 3.

Recall that the utility range R is initialized to the utility space $\{u \in \mathbb{R}^d_+ | \sum_{i=1}^d u[i] = 1\}$ which is a $(d-1)$-dimensional polyhedron. Each pair of points p_i and p_j in D could construct a $(d-1)$-dimensional hyperplane $h_{i,j} = \{u \in \mathbb{R}^d | u \cdot (p - q) = 0\}$. If it does

not intersect with R (i.e., $R \in h^+_{i,j}$ or $R \in h^-_{i,j}$), since the user's utility vector u_0 is in R, the order between p_i and p_j w.r.t. u_0 is known.

Before the user provides any information, some hyperplane $h_{i,j}$, constructed by points $p_i, p_j \in D$, intersects with the utility space, i.e., the order between p_i and p_j is unknown. Denote as $h'_{i,j}$ the $(d-2)$-dimensional hyperplane in which the intersection of $h_{i,j}$ and the utility space lie. Note that there are $\eta = \frac{n(n-1)}{2}$ hyperplanes $h_{i,j}$ consisting of $p_i, p_j \in D$ ($n = |D|$). The utility space is divided by at most η hyperplanes $h'_{i,j}$ into a number of disjoint regions, called cells, denoted by $\mathbb{C}$. For each $\mathbb{C}$, it does not intersect with any hyperplane $h_{i,j}$ constructed by $p_i, p_j \in D$, i.e., the order between each pair of points is known. Thus, each cell corresponds to a different ranking of points in D.

Consider the worst case that we need to identify the ranking of points in D w.r.t. the user' s preference when the number of rankings (i.e., number of cells) is the largest. To achieve this, (1) all the hyperplanes $h_{i,j}$, where $p_i, p_j \in D$, intersect with the utility space; (2) the corresponding $(d-2)$-dimensional hyperplanes $h'_{i,j}$, where $p_i, p_j \in D$, are in general position (i.e., the intersection of every η' hyperplanes is $(d-\eta'-1)$-dimensional, where $\eta' = 2, 3, ..., d-1$); and (3) the intersection of each pair $h'_{i,j}$ and $h'_{i',j'}$, where $p_i, p_j, p_{i'}, p_{j'} \in D$, is in the utility space. Then the utility space is divided into $N_{d-1}(\eta) = C^0_\eta + C^1_\eta + ... + C^{d-1}_\eta$ cells [16].

In the following, we first assume that all cells divided by these hyperplanes $h'_{i,j}$ ($p_i, p_j \in D$) are in equal size (i.e., the probability of each ranking is equal) and then consider the general case.

Suppose all $N_{d-1}(\eta)$ cells into which the utility space is divided into are in equal size. Note that our algorithm initializes an random order for points to build hyperplane set. For those $\frac{(v-1)(v-2)}{2}$ hyperplanes $h_{i,j}$ constructed by the first $v-1$ points ($i, j \in [1, v]$, $i > j$, $v \in [2, n]$), [19] shows in Lemma 3 that the cells divided by the corresponding hyperplanes $h'_{i,j}$ (the intersection of the utility space and $h_{i,j}$ lies in) are in equal size, i.e., the probability of each ranking related to the first $v-1$ points is equal.

Assume we have learned the ranking of the first $v-1$ points, i.e., none of the hyperplanes in hyperplane sets $H_2, ..., H_{v-1}$ intersects with the utility range R. The first μ hyperplanes $h'_{i,j}$ ($i, j \in [1, v]$, $i > j$) divide the utility space into $N_{d-1}(\mu) = C^0_\mu + C^1_\mu + ... + C^{d-1}_\mu$ cells, where $\mu = \frac{(v-1)(v-2)}{2}$. Since the ranking of the first $v-1$ points is known, the user's preference can be located in a single cells partitioned by the first μ hyperplanes.

For each hyperplane $h_{v,j} \in H_v$, if it intersects with the utility range R, we need to ask the user a question and update R with $R \cap h^+_{v,j}$ (or $R \cap h^-_{v,j}$). Consider the relationship between $h_{v,j}$ and the $N_{d-1}(\mu)$ cells divided by the first μ hyperplanes. Since the first μ hyperplanes and $h_{v,j}$ are in general position and their intersections are in the utility space, the $(d-2)$-dimensional hyperplane $h_{v,j}$ can be seen as being divided by the previous μ hyperplanes into $N_{d-2}(\mu)$ disjoint regions. This means there are $N_{d-2}(\mu)$ cells [16] (divided by the previous μ hyperplanes) which intersect with $h_{v,j}$.

Note that the $N_{d-1}(\mu)$ cells divided by the first μ hyperplanes are in equal size and R must be one of them before considering H_v. For each $h_{v,j} \in H_v$, since it intersects with $N_{d-2}(\mu)$ cells divided by the previous μ hyperplanes, the probability P_v that it intersects with R (i.e., the probability of asking a question) is as follows.

$$\begin{aligned} P_v &= \frac{N_{d-2}(\mu)}{N_{d-1}(\mu)} \\ &= \frac{C^0_\mu + C^1_\mu + ... + C^{d-2}_\mu}{C^0_\mu + C^1_\mu + ... + C^{d-1}_\mu} \\ &\le \frac{C^0_\mu + C^1_\mu + ... + C^{d-2}_\mu}{C^{d-2}_\mu + C^{d-1}_\mu} \\ &= \frac{\sum_{\varphi=0}^{d-2} C^\varphi_\mu}{C^{d-1}_{\mu+1}} \end{aligned}$$

If $2(d-2) \le \mu$, then $C^\varphi_\mu \le C^{d-2}_\mu$, where $\varphi = 0, 1, ..., d-3$. Define $1 \le \alpha \le d-1$ such that $\alpha C^{d-2}_\mu \ge \sum_{\varphi=0}^{d-2} C^\varphi_\mu$. Then we have

$$P_v \le \frac{\alpha C^{d-2}_\mu}{C^{d-1}_{\mu+1}} = \frac{\alpha(d-1)}{\mu+1} \le \frac{2\alpha(d-1)}{(v-2)^2}$$

If $2(d-2) > \mu$, the relation between C^φ_μ and C^{d-2}_μ varies, where $\varphi = 0, 1, ..., d-3$. For ease of calculation, we define $P_v = 1$.

Since $\mu = \frac{(v-1)(v-2)}{2}$, we define

$$P_v = \begin{cases} 1 & v \le \lceil 2\sqrt{d} \rceil + 2 \\ \frac{2\alpha(d-1)}{(v-2)^2} & v > \lceil 2\sqrt{d} \rceil + 2 \end{cases}$$

Because the probability that each hyperplane in H_v needs to be checked by asking a question is P_v, the expected number of hyperplanes asked user is $\sum_{v=2}^{n} \sum_{j=1}^{v-1} P_v$.

$$\begin{aligned} \sum_{v=2}^{n} \sum_{j=1}^{v-1} P_v &= \sum_{v=2}^{\lceil 2\sqrt{d} \rceil + 2} \sum_{j=1}^{v-1} P_v + \sum_{v=\lceil 2\sqrt{d} \rceil + 3}^{n} \sum_{j=1}^{v-1} P_v \\ &\le \frac{(\lceil 2\sqrt{d} \rceil + 2)(\lceil 2\sqrt{d} \rceil + 1)}{2} + \sum_{v=\lceil 2\sqrt{d} \rceil + 3}^{n} \sum_{j=1}^{v-1} \frac{2\alpha(d-1)}{(v-2)^2} \\ &\le \frac{(2\sqrt{d}+3)(2\sqrt{d}+2)}{2} + \sum_{v=\lceil 2\sqrt{d} \rceil + 3}^{n} \frac{2\alpha(d-1)}{(v-2)^2} + \frac{2\alpha(d-1)}{(v-2)^2-1} \\ &+ \frac{2\alpha(d-1)}{(v-2)^2-2} + ... + \frac{2\alpha(d-1)}{(v-2)(v-3)} \\ &\le \frac{4d + 10\sqrt{d} + 6}{2} + \sum_{i=(\lceil 2\sqrt{d} \rceil)^2}^{(n-2)^2} \frac{2\alpha(d-1)}{i} \\ &\le (2d + 5\sqrt{d} + 3) \log_2((\lceil 2\sqrt{d} \rceil)^2 - 1) \\ &+ 2\alpha(d-1) \log_2(\frac{(n-2)^2}{(\lceil 2\sqrt{d} \rceil)^2 - 1}) \\ &\le 10\alpha d \log_2((\lceil 2\sqrt{d} \rceil)^2 - 1) + 2\alpha d \log_2(\frac{(n-2)^2}{(\lceil 2\sqrt{d} \rceil)^2 - 1}) \\ &\le 10\alpha d \log_2 (n-2)^2 \end{aligned}$$

Thus, if the cells divided by the η hyperplanes $h'_{i,j}$ are in equal size, i.e., the probabilities of different rankings of the n points in D are the same, the expected number of questions asked is $\mathbb{E}_u = O(d \log_2 n)$.

Now consider the general case that all cells are in random size, i.e., the probabilities of different rankings of the n points in D are various. Let $\mathbf{P}_i$ denote the probability of the i-th ranking, i.e., the size of the i-th cell. Use $\mathcal{N}_i$ and $\mathbb{E}[\mathcal{N}_i]$ to represent the number of questions and expected number of questions asked to locate the cell containing the user's utility vector u_0 if u_0 is in the i-th cell (ranking). Denote by $\mathbb{E}_g$ the expected number of questions asked in the general case. We have

$$\mathbb{E}_g = \sum_{i=1}^{N_{d-1}(\eta)} \mathbb{E}[\mathcal{N}_i]$$

, where $\eta = \frac{n(n-1)}{2}$. Assume the probability of any ranking is bounded by $\mathbf{P}_i \leq \frac{c}{N_{d-1}(\eta)}$, for some constant $c > 1$. In order to obtain the largest $\mathbb{E}_g$, we distribute the probability $\frac{c}{N_{d-1}(\eta)}$ to $j = \frac{N_{d-1}(\eta)}{c}$ cells whose $\mathbb{E}[\mathcal{N}_i]$ is the largest. Without loss of generality, set $\mathbb{E}[\mathcal{N}_i] = \rho$ for these particular cells and then the largest $\mathbb{E}_g$ should be ρ. For the remaining $N_{d-1}(\eta) - j$ cells, each cell should be bounded by at least d hyperplanes. Since one question (show points p_i and $p_j$to user) is needed for each bounded hyperplane $h_{i,j}$, the number of questions asked for these cells must be larger than d. Therefore, we have

$$\mathbb{E}_u = \frac{1}{N_{d-1}(\eta)} \sum_{i=1}^{N_{d-1}(\eta)} \mathbb{E}[\mathcal{N}_i] \geq \frac{\rho}{N_{d-1}(\eta)} j + d\frac{N_{d-1}(\eta) - j}{N_{d-1}(\eta)}$$

Then we obtain the largest $\mathbb{E}_g$

$$\mathbb{E}_g = \rho \leq c(\mathbb{E}_u - d\frac{N_{d-1}(\eta) - j}{N_{d-1}(\eta)}) \leq c\mathbb{E}_u$$

Therefore, in general case, the expected number of questions asked is $\mathbb{E}_g = c\mathbb{E}_u = O(cd \log_2 n)$, where $c > 1$ is a constant. □