Interactive Top-k Points Searching

ABSTRACT

When facing a dataset with a large number of tuples, it is unpractical for a user to scan the whole database to pick his/her satisfied tuples. Top-k queries are able to find the best k tuples based on a given preference from the user. However, in practice, it is difficult for a user to specify his/her preference explicitly. We study how to enhance the top-k queries by interacting with users. We present two tuples at each round and ask the user which one s/he prefers to. Based on his/her feedback, we learn his/her preference implicitly and then return one of the top-k tuples. Different from the existing work which aims to find the ranking of all the tuples or only the best one, we are looking for one of the top-k tuples. In this way, we not only are able to recommend satisfied tuples to user, but also require asking few questions. Specifically, we present an algorithm *2D-PI* which needs an asymptotically optimal number of questions in 2-dimensional space, and two algorithms *HD-PI* and *RH* with provable performance guarantees in d-dimensional space ($d \geq 2$), where they focus on the number of questions asked user and the execution time, respectively. Experiments which were conducted on synthetic and real datasets show that our algorithms outperform existing algorithms by returning satisfied tuples with much fewer questions in less time.

CCS CONCEPTS

• **Information systems** → **Data analytics**.

KEYWORDS

top-k queries; user interaction; data analytics

ACM Reference Format:

. 2021. Interactive Top-k Points Searching. In *2021 International Conference on Management of Data (SIGMOD '21), June 20 - June 25, 2021, Xi'an, Shaanxi.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

For a dataset with a number of tuples described by several attributes, various operators aiming to return representative tuples have been studied. Such operators categorized as *multi-criteria decision-making tools* are utilized in miscellaneous applications, including computer selling, car purchase and tickets booking. For example, computers can be described with the speed of the CPU, the capacity of the hard drive, the price and so on. Alice would like to buy a cheap computer with a large capacity hard drive. In literature [16, 18, 20], we use a monotonic function, called utility function, to represent the her preference. For the traditional top-k queries [19], the utility function is known. The system calculates the *utility* (i.e., function value) of each computer based on the utility function and returns k computers with the highest utilities. The utility shows to which extent a user favors a computer, where a higher utility indicates the computer is more favored by the user. The top-k queries are able to return tuples tailored for the user, since they search tuples based on the his/her utility function particularly. However, it is usually not easy for a user to specify his/her utility function accurately. For example, we cannot tell that the importance of the attributes: price and capacity of hard drive to us are 41.2% and 58.1%, respectively. If the utility function given to the system differs slightly with the user's preference, the output generally varies a lot.

Motivated by this, we consider applying user interaction [11, 16, 20, 21] to avoid asking the user to explicitly specify his/her utility function. Specifically, it tries to learn the user's utility function implicitly by interacting with the user for rounds. At each round, it asks the user to provide some "hints" on what his/her utility function looks like, which are easy for the user to tell. Based on the user's feedback, it estimates the user's utility function and determines the utility of each tuple. Different from requiring the user to tell his/her exact utility function, we only want to ask some easy questions to the user, which naturally appear in our daily life. We stick to the strategy proposed by [11, 18], which requires little user effort: *we display two tuples to the user and ask him/her to pick his/her favorite one (i.e., the tuple with larger utility).* For example, a seller might show Alice two computers and ask her which one she prefers to. A tour guide may present two beautiful spots and ask Alice: where would you like to travel?

Based on the way of choosing tuples shown to user, existing algorithms can be classified into two categories: passive learning [1, 7, 12] and active learning [11, 20]. For the former case, the displayed tuples are designed in advance, which results in that the tuples shown to different users are the same. While for the latter one, the tuples are chosen adaptively, based on the user' s previous answers. Obviously, in order to be applied for all the users, passive learning needs to interact with the user for more rounds than the active learning. Thus, we focus on selecting tuples adaptively.

In this paper, we want to return one of the user's top-k tuples by asking the user's effort as little as possible (i.e., adaptively asking the user questions as few as possible), since no user is patient to answer questions for dozens of hours. The user's effort is measured by the number of questions asked user. Thus, we care about: *how many questions do we need to ask to return one of the top-k tuples.*

Among the existing methods, [11] proposes an algorithm *Active-Learning* which obtains the ranking of all tuples w.r.t. the user' s utility function (and then the top-k tuples are known) by interacting with the user. To be specific, it checks the order of each pair of tuples. If the order is unclear, it asks the user to tell which one s/he prefers to between this pair. *Active-Ranking* has been proved that it is able to find the ranking of tuples by asking a logarithmic

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

number of questions in expectation. However, it assumes that the tuples in the database are in a *general position*, i.e., there do not exist i tuples in the database which lie in a $(i-2)$-dimensional space for $i = 2, 3, ..., d+1$, not for arbitrary database. Besides, it learns the ranking of tuples instead of the top-k points, which lead to requiring much user's effort.

[20] first proposes the problem of finding the (closed to) favorite tuple (i.e., top-1 tuple) by adaptively interacting with the user. It shows algorithms *Median* and *Hull* that ask an asymptotically optimal number of questions in 2-dimensional space and algorithms *UH-Random* and *UH-Simplex* with provable performance guarantees in d-dimensional space ($d \geq 2$). However, [20] is limited on finding the (close to) favorite tuples, which requires to learn much information, i.e., ask the user many questions. In practice, the user is more likely to accept receiving a tuple with which s/he is satisfied (might not the best) by being asked fewer questions. For example, instead of obtaining the favorite car by asking 10 questions, the user are willing to receive the second favorite car by asking only 2 questions. To some extent, [20] can be seen as a particular case of our work when $k = 1$. There are also some other preference learning algorithms in the literature. We postpone their detailed discussion to Section 2.

Contributions: We propose algorithms for returning one of the user's top-k tuples by interacting with the user. Specifically,

- 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 needed for finding one of the user's top-k tuples.
- We propose an algorithm *2D-PI* in 2-dimensional space. It needs $O(\log_2 \lceil \frac{2n}{k+1} \rceil)$ questions to determine one of the user's top-k tuple, which is asymptotically optimal in terms of the number of questions.
- We propose an algorithm *RH*. It guarantees a logarithmic number of questions asked in d-dimensional space in expectation, which is asymptotically optimal with respect to the number of questions asked if d it fixed.
- We design an algorithm *HD-PI* with a provable guarantee in d-dimensional space.
- We conducted experiments to demonstrate the superiority of our algorithms. The results show that they are able to return one of the user's top-k tuple by requiring nearly half of the questions when k is slightly large ($k > 50$).

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

2 RELATED WORK

[16] first proposes the problem of minimizing the *regret ratio*, which is a evaluating parameter, by interacting with the user for rounds. The regret ratio represents how much regretful a user is if s/he sees the returned tuple instead of his/her favorite one in the database. Its proposed algorithm *UtilityApprox* is shown to be useful in reducing the regret ratio in both theoretical and practical aspects. However, the weakness is that it displays *fake tuples* to the user during the interaction, which are intentionally constructed (not from the database). This might produce some unrealistic tuples (e.g. a car with 10 dollars and 50000 power) and the user may be disappointed if the displayed tuples with which s/he is satisfied do not exist.

To over come the disadvantages of "fake tuple", [20] proposes to find the user' s (close to) favorite tuple by interacting with the user on *real tuples* (tuples in the database) for rounds. It presents tuples to the user and asks him/her to pick his/her favorite one among them at each round. It shows algorithms, *Median* and *Hull*, that ask an asymptotically optimal number of questions in 2-dimensional space and algorithms, *UH-Random* and *UH-Simplex*, with provable performance guarantees in d-dimensional space ($d \geq 2$). Furthermore, to reduce the number of rounds interacting with user, [21] proposes an improved version *Sorting-Random* and *Sorting-Simplex* which display tuples to the user and ask him/her to give an order of them at each round. However, [21] does not reduce the user's effort essentially, since giving an order among tuples is equivalent to picking the user's favorite one among tuples several times. Besides, both [20, 21] limit on finding the (close to) favorite tuple of the user, which requires to interact with the user for many rounds. In practice, users are not patient to be involved in too many rounds and usually it is enough to find a tuple which will be satisfied by user, not necessary the best. For example, compared with returning the favorite car by asking 10 questions, the user are willing to receive the second favorite car by asking only 2 questions. which also results in more rounds of interaction.

Thus, our work focuses on returning one of the top-k tuples by interacting with the user on real tuples. This avoids the weakness of displaying fake tuples and reduces the user's effort essentially (reducing the number of interaction rounds but still asking the user to pick his/her favorite tuple), while still returning a tuple that the user is satisfied with. To some extent, the problem studied by [20, 21] can be seen as a particular case of our problem, when $k = 1$.

There are alternative methods [11, 18] which focus on learning the user's preference by pairwise comparison. [18] proposes an algorithm *Preference-Learning* which shows to user two tuples (pairwise) at each round and ask him/her to pick his/her favorite one. However, it does not scale well on the database size and requires a large number of rounds to learn the user's preference exactly. The algorithm *Active-Ranking* proposed by [11] tries to learn the ranking of tuples by interacting with the user, where all the tuples are assumed to be in *general position*. Rather than applying the learned information to find one of the user's top-k tuples, it focuses on deriving the order for any pair of tuples in the database, which needs to interact with the user for a number of rounds. [11] shows an tight bound on the number rounds required in expectation. However, it obtains the theoretical guarantee under the assumption that the tuples are the general position not for general cases.

In machine learning area, our problem can be roughly mapped to the problem of *learning to rank* [13] by pairwise comparison, which apply supervised machine learning (ML) to solve ranking problems. [14] [9] address it by using the strategy of choosing pairs of tuples adaptively (called active learning), i.e., selecting

tuples based on the user's previous feedback. [14] gives favorable guarantees for quicksort on the popular *Bradley-Terry-Luce* model (*BTL* model) under natural assumptions on the parameters. [9] introduces techniques to resolve the top-ranked tuples with a few pairwise comparisons. However, they stay on the relation between tuples and does not bring in the concept of attributes of tuples and preference of users, which requires more interaction rounds with the user. In comparison, we consider the inter-relation between tuples so that even if some tuples have not been involved in the pairwise comparison, we may be able to accurately decide whether it is interested by user based on the learned information.

In the following, we show a comparison between our algorithms and some existing algorithms which can be adapted to our problem. We adapt each of them to return one of the user's top-k tuple by interacting with the user. The details of the adaptation are shown in Section 6. We list the number of rounds required by each algorithm in Table 1. It is obvious that our algorithm *HD-PI* is able to ignore the impact of dimensionality in the optimal case and *RH* achieves asymptotically optimal in terms of the number of questions asked in expectation if d is fixed. Note that although algorithm *Action-ranking* also has an expected logarithmic bound, it is under the condition that all the tuples are in general position, while the proof of *RH* works for any cases.

Compared with the existing methods, we have the following advantages. First, we require less user effort. Specifically, since we cares about one of the top-k tuples we do not need to learn much information, while some existing works[16, 20] focus on finding the (close to) favorite tuple. Instead of asking the user to give an order for the displayed tuples [21], we only need the user to pick his/her favorite one. We also utilize the concept of attributes of tuples, while some existing work [1, 7, 12] ignore the inner-connection between tuples. Second, we ensure that the tuples presented to the user are real, while some existing algorithms[16] rely on fake tuples. Third, we exploit the inter-relation between tuples and deal with the geometry concept of tuples more carefully and deeply. As a result, we are able to return one of the user' s top-k tuples effectively and efficiently.

3 PROBLEM DEFINITION

The input of our problem is a set D containing n tuples, each of which is specified by d attributes. In the rest of the paper, we regard each tuple as a point in a d-dimensional space and use word "tuple" and "point" interchangeably. For each point p, we denote its i-th dimensional value as $p[i]$, where $i \in [1, d]$. Without loss of generality, 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. The dataset contains 5 points in a 2-dimensional space, where each dimension is normalized.

Following [2, 20], the user's preference is modeled by a linear function $f(p) = \sum_{i=1}^{d} u[i]p[i]$, called utility function, denoted as $f(p) = u \cdot p$ 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 this attribute is more important to the user. In the rest of the paper, we use "utility vector" to refer to the user's preference and call the domain of u as the *utility space*. Given a utility vector u, we compute the utility of each point and rank them based on their utilities. The k points with the largest utilities are defined as the *top-k points* w.r.t u. Since the ranking of points remains unchanged with different scaled utility vectors [16, 20], 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 p_2 in Table 2. Its utility w.r.t. f is $f(p_2) = 0.4\times0.3+0.6\times0.7 = 0.54$. The utilities of other points are computed similarly. Assume $k = 2$. Points p_1 and p_3 with the highest utilities are the top-k points.*

Given a set D, our goal is to return a point, which is one of the user's top-k points, by interacting with users for rounds. At each round, the system adaptively chooses a pair of points as a question shown to the user. Then, the user chooses the point s/he prefers to. Based on his/her feedback, we learn the user's preference implicitly. Finally, when sufficient information has been collected, the interaction stops and the system returns the desired point to the user. Formally, we are interested in the following problem.

Problem 1. ***(Interactive Top-k Searching (ITS))*** *Given a 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.*

We first present a lower bound on ITS. Due to the lack of space, some complete proofs can be found in the technical report [3].

Theorem 3.1. *There is a dataset of n points such that for any algorithm, it needs to ask $\Omega(\log_2 \frac{n}{k})$ questions to the user 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 such dataset. □

4 2D ALGORITHM

In this section, we focus on 2-dimensional ITS. 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, 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 each point into a line segment, 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

Algorithm (d dimension)	Worst Case	Optimal Case	Expected Case
UH-Random [20]	$O(n)$	unknown	unknown
UH-Simplex [20]	$O(n)$	unknown	$O(deg_{max}\sqrt[d]{n})$
Active-Ranking [11]	$O(n^2)$	$O(d\log n)$	$O(cd\log n)$, where$c>1$
Preference-Learning [18]	$O(n^2)$	None	None
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($u = (0.4, 0.6)$)

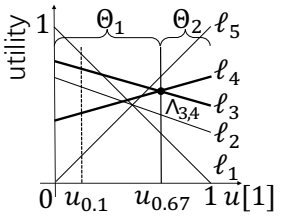

Figure 1: 2D Partitions

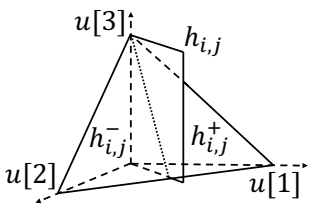

Figure 2: Hyperplane

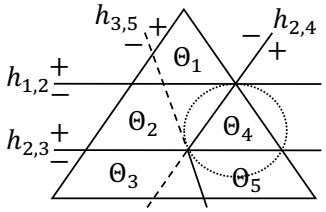

Figure 3: 3D Partitions

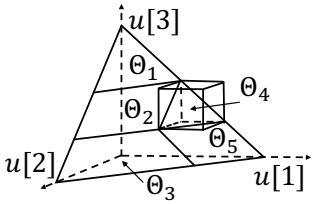

Figure 4: B_Rectangle

through $(u[1], 0)$ in the geometric space. It is easy to verify that 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. In Figure 1, the utility line of $u_{0.1} = (0.1, 0.9)$ intersects ℓ_1, ℓ_2, ℓ_3, ℓ_4 and ℓ_5, respectively. Based on the order of intersections, the ranking w.r.t. $u_{0.1}$ is p_1, p_3, p_2, p_4, p_5.

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 his/her 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 desired 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 the following two components: (1) a queue $\mathcal{Q}$ of size n, which stores all the points in D based on the order (from top to bottom) of the intersections between t and their transformed lines; (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 we now construct partition Θ_x. 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, we start the construction of the next partition $\Theta_{x+1}$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 $l_1 = 0$ and set $\mathcal{Q} = \{p_1, p_3, p_2, p_4, p_5\}$ with p_1, p_2 labeled by T_1. The intersections of neighboring pairs in $\mathcal{Q}$ 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, 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 $\mathcal{H}$. The label of p_1, p_3 are not updated for they are not the k-th and $(k+1)$-th point in $\mathcal{Q}$ before swapping.*

Theorem 4.1. *Algorithm 1 runs in $O(n^2 \log n)$ time.*

Proof. In the worst case, we need to process the intersections of lines w.r.t. all pairs of points in D and there are $O(n^2)$ intersections.

Algorithm 1 Utility Space Partition

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

Algorithm 2 User Interaction

Input: $\Theta = \{\Theta_1, \Theta_2, ..., \Theta_m\}$, $\mathcal{E} = \{q_1, q_2, ..., q_m\}$
Output: q_x, which is one of the user's top-k points
1: $C \leftarrow < \Theta_1, \Theta_2, ..., \Theta_m >$, $left \leftarrow 1$, $right \leftarrow m$
2: **while** $|C| > 1$ **do**
3: $x \leftarrow left - 1 + \lfloor \frac{right - start + 1}{2} \rfloor$
4: $p_i, p_j \leftarrow$ points has the k-th and $(k+1)$-th largest utility w.r.t. the utility vector $(r_x, 1 - r_x)$
5: Display p_i, p_j to user
6: **if** The user prefers p_i to p_j **then**
7: $right \leftarrow x$
8: **else**
9: $left \leftarrow x + 1$
10: **end if**
11: $C \leftarrow < \Theta_{left}, ..., \Theta_{right} >$
12: **end while**
13: **return** q_x, the associated point of the only partition left in C

At each intersection, we update $\mathcal{H}$ and $\mathcal{Q}$ 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 4.1. *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 the number of partitions obtained by our algorithm will not be more than that of the optimal case. □

Note that [2, 6] propose algorithms which achieve similar purpose. However, [2] only presents an approximate algorithm, while our algorithm is able to return the exact solution. Besides, it is not easy to process the real implementation of [6] due to its complicated data structure with its theoretical result. To the best of our knowledge, there are no real implementation of [6] in the literature.

4.3 User Interaction

After the utility space is partitioned, we interact with the user to locate the partition containing his/her utility vector. Note that the ranking of two points changes at most once by sweeping t from left to right, i.e. varying its $u[1]$-value from 0 to 1. Given two lines ℓ_i and ℓ_j in the geometric space, whose intersection is $\wedge_{i,j}$, if the corresponding p_i ranks higher than 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 u such that $u[1] > \wedge_{i,j}[1]$. Thus, 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 is 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 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 show two points p_i, p_j to the user, where p_i, p_j are the points with the k-th and $(k+1)$-th largest utility w.r.t. the utility vector $(r_x, 1 - r_x)$, respectively. 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 4.2. *Continue Example 4.1. C is initialized as $< \Theta_1, \Theta_2 >$. Since Θ_1 is the median partition, we present to the user the points p_3, p_4 which has the k-th and $(k+1)$-th largest utility w.r.t. the utility vector $(r_1, 1 - r_1)$. 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 4.2. *Algorithm 2D-PI solves 2-dimensional ITS 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 4.3. *Algorithm 2D-PI is asymptotically optimal in terms of the number of questions asked for 2-dimensional ITS.*

5 HIGH DIMENSIONAL ALGORITHM

In this section, we consider high dimensional ITS. We first show some preliminary 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 on 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 Preliminary

Recall that in d-dimensional space $\mathbb{R}^d$, the utility space is a $(d-1)$-dimensional polyhedron. For example in Figure 2, the utility space

is a triangular region when $d = 3$. For each pair of points $p_i, p_j \in D$, where $i, j \in [1, n]$, we could 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* [4]. 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. <), i.e., p_i ranks higher (resp. lower) than p_j w.r.t u [20].

In geometry, a polyhedron $\mathcal{P}$ is the intersection of a set of halfspaces [4] 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, we maintain polyhedrons which possibly contain the user's utility vector. Every time the user provides a feedback, we build a halfspace 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 $h_{i,j}$, e.g., in Figure 3, the 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 Figure 3 and Figure 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 $\mathcal{P}$. We will compare the performance of these two bounding methods experimentally in Section 6.1.

As an overview, our *HD-PI* and *RH* algorithm follow the interactive framework from [20]: we interact with the user for rounds until we find points which are the user's top-k points. At each iteration,

- (**Points selection**) We select two points from the database and present them to the user.
- Then the user picks his/her 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 results.

In the following, we present the algorithm for high dimensional ITS 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* (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 consists of 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 the 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 5.1 that dividing the utility space into the least number of partitions is an NP-hard problem.

Theorem 5.1. *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 point 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* [4] in D based on some existing algorithms (e.g. [4]), which are the top-1 points w.r.t at least a utility vector in the utility space. In practice, we can use a sampling strategy for approximating V. Then, for each $p_i \in V$, we create a partition $\Theta_i = \{u \in \mathbb{R}^d | u \in h^+_{i,j}, \forall p_j \in V / p_i\}$ in C. It can be verified that p_i is the top-1 point w.r.t any $u \in \Theta$. Note that there are $O(n)$ partitions in C and initially $R = \cup_{\Theta \in C} \Theta = \{u \in \mathbb{R}^d_+ | \sum_{i=1}^d u[i] = 1\}$.

Each row of the list Γ corresponds to a hyperplane $h_{i,j}$, constructed by points $p_i, p_j \in V$. It records the relationship between $h_{i,j}$ and the partitions in C (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 Γ for partitions in Figure 3.

When the user answers more questions, we know more information about the user's utility vector. Then, invalid partitions in C will be removed and Γ will be updated accordingly.

Assume that the user prefers p_i to p_j in a question. Then, the data structures are 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 $\Theta \cap h^+_{i,j}$. Since partitions in C are updated, Γ is updated accordingly. In particular, if there is a hyperplane $h_{i',j'}$ in Γ such that (1) R is in $h^+_{i',j'}$ or (2) R is in $h^-_{i',j'}$, we remove $h_{i',j'}$ from Γ since no matter

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

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

what the user's utility vector is (as long as it is in R), $p_{i'}$ must rank higher than $p_{j'}$ and thus, $h^+_{i',j'}$ cannot be used to update C later.

Example 5.1. *Consider 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 prefer p_3 to p_2. Then his/her utility vector is in $h^+_{2,3} (= h^-_{3,2})$. We first remove Θ_1, Θ_2, Θ_4 in C since they cannot contain the user's utility vector. Moreover, since none of the partitions intersect $h_{2,3}$, other partitions in C do not change. Then, based on the updated C we modify Γ by removing Θ_1, Θ_2, Θ_4 in each row. After the removal, we find that $h_{1,2}$ and $h_{2,3}$ can be deleted from Γ since R is in $h^-_{1,2}$ and $h^-_{2,3}$. Finally, the even scores of the remaining rows are renewed. The updated Γ is shown in Table 4.*

Note that [2] presents an algorithm which focuses on the similar purpose of dividing the utility space into the least number of partitions. However, [2] only obtains an approximate solution by consuming much time. In detail, it has two steps, where the time complexity of the first step is $O(n^{\lfloor d/2 \rfloor} k^{\lfloor d/2 \rfloor})$ [15] and the second step is an NP-hard problem. While our method only require $O(n^{\lfloor d/2 \rfloor})$ 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 we can quickly location the user's utility vector. 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 $h_{i,j}$ 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 5.1. *Given a hyperplane $h_{i,j}$ of points p_i and p_j, the* even score *of $h_{i,j}$ 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 $h^+_{i,j}$, in $h^-_{i,j}$ and intersect $h_{i,j}$, respectively.*

Intuitively, a higher even score means that the corresponding hyperplane divides the partitions more evenly. The first term in Definition 5.1 shows that we want more partitions in $h^+_{i,j}$ or $h^-_{i,j}$ (and as even as possible), while 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 points p_i and p_j whose hyperplane $h_{i,j}$ in Γ has the highest even score.

Algorithm 3 The *HD-PI* Algorithm

Input: A set of points D
Output: A point p, which is one of the user's top-k points
1: Divide the utility space into several partitions Θ_x
2: $C \leftarrow \{\Theta_1, \Theta_2, ..., \Theta_m\}$
3: Initial the utility range R and the list Γ
4: **while** $|C| > 1$ & $\nexists p \in D$ satisfying Lemma 5.2 **do**
5: Select the hyperplane $h_{i,j}$ in Γ with the highest even score
6: Display points p_i and p_j corresponding to $h_{i,j}$ to the user
7: Update R, C and Γ based on the user's feedback
8: **end while**
9: **return** A point p, which is one of the user's top-k points

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 can stop immediately if there exists a point in D which is guaranteed to be among the top-k points w.r.t any $u \in R$. The following lemma tells whether a given point p_i is a qualified point to be returned.

Lemma 5.2. *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 among the top-k points w.r.t any $u \in R$.

To determine whether such points exist, a naive idea is check each p_i in D using Lemma 5.2. However, the naive checking 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 5.2. 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 our full *HD-PI* algorithm is presented in Algorithm 3. Its performance is summarized as follows.

Theorem 5.2. *HD-PI solves ITS 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), ITS 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 ITS 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 interaction, based on the user preference on p_i, p_j, we update R to be $R \cap h^+_{i,j}$ (or $h^-_{i,j}$).

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 the 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 complete ranking of 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 iteration, the 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 the 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 the 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 those 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 the theoretical analysis is presented in Theorem 5.3.

Theorem 5.3. *Algorithm RH solves ITS by interacting with the user for $O(cd \log n)$ rounds in expectation, where $c > 1$ is a constant.*

Proof Sketch. The ranking of points w.r.t. different utility vectors in the utility space might be different. 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 set of points D
Output: A point p, which is one of the user's top-k points

```
Initialize a random order of points in D
for i = 2, ..., n do
    Initialize the set H_i
    while ∃h_{i,j} ∈ H_i intersecting R do
        Find h_{i,j} ∈ H_i intersecting R with the min-distance to R_c
        Display points p_i and p_j corresponding to h_{i,j} to the user
        Based on the user feedback, update R
        if the stopping condition is satisfied then
            return the qualified point of Lemma 5.2 or an arbitrary
            top-k points if the complete ranking of D is known
        end if
    end while
end for
```

Corollary 5.4. *Algorithm RH is asymptotically optimal in terms of the number of questions asked in expectation for ITS.*

6 EXPERIMENT

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 both synthetic and real datasets. Specifically, the synthetic datasets are *anti-correlated* which were generated by the generator developed for *skyline* operators [5, 17]. The real datasets are *Island*, *Weather*, *Car* and *NBA* which are commonly used in existing studies [18, 20]. *Island* is 2-dimensional, which contains 63,383 geographic positions and *Weather* includes 178,080 tuples described by four attributes. *Car* is 4-dimensional, which consists of 68,010 second-hand cars after it is filtered by only keeping the cars whose attributes' value are in normal range. *NBA* involves 16,916 players after we delete the records with missing values. Six attributes are used to represent the performance of each player. For all the datasets, we normalized each dimension to $[0, 1]$ and preprocessed them to contain *k-skyband points* only, where for each point, there are fewer than k points in the dataset which are better than it in all attributes [10].

Algorithms. We evaluated our 2-dimensional algorithm: *2D-PI* and d-dimensional algorithms: *HD-PI* and *RH*. Mentioned in Section 5.2.1, the sampling strategy is utilized to accelerate finding convex points for *HD-PI*. Specifically, we uniformly sample the utility space and find the top-1 point w.r.t. each sampled utility vector. Based on the different strategies for searching convex points, we distinguish algorithm *HD-PI* into two versions: accurate and sampling. The competitor algorithms are: *Median* [20], *Hull* [20], *Active-Ranking* [11], *UtilityApprox* [16], *Preference-Learning* [18], *UH-Random* [20] and *UH-Simplex* [20]. Since none of them can solve our problem directly, we adapted them as follows:

- Algorithms *Median* and *Hull* (only work in 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 remaining points (instead of 1 remaining point).

- Algorithm *Active-Ranking* concentrates on learning the ranking of the 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 parameter *regret ratio* [16] satisfies a given threshold ϵ. We set $\epsilon = 1 - \frac{f(p_k)}{f(p_1)}$, 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, where either its utility is the largest w.r.t the user's utility vector or its regret ratio satisfies a given threshold ϵ, by interacting with the user. We set ϵ the same as that in algorithm *UtilityApprox*. Besides, we create another version, namely *UH-Simplex-Adapt* and *UH-Random-Adapt*, by modifying their point deletion condition and the 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 and terminates if the dot product of the user's utility vector and the estimated utility vector meets a given threshold ϵ. We set $\epsilon = 1 - 10^{-6}$ and arbitrarily return one of the top-k points w.r.t the estimated utility vector. Note that $\epsilon = 0.95$ in [18]. However, the returned point is always not 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 [18].

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 as 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, where each algorithm were conducted 10 times with different generated user's utility vectors and the average performance was reported: (1) *execution time*. The time needed to find the desired point; (2) *the number of questions asked*. The number of rounds interacting with user, which quantifies the user's effort.

In the following, the parameter setting of our proposed algorithms is first shown in Section 6.1. Then, the performance of algorithms on synthetic and real datasets is presented in Section 6.2 and Section 6.3, respectively. Next, in Section 6.4, the result of a user study on dataset *Car* is demonstrated. Finally, the experiments are summarized in Section 6.5.

6.1 Performance Study of Our Proposed Algorithms

In this experiment, we compared different strategies applied to our proposed algorithm *HD-PI*. For this comparison, we also included two measurements for evaluating different bounding strategies: (1) *effective ratio*. The ratio N_B/N_I, where N_I is the number of checkings 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 by bounding ball/rectangle strategy. (2) *execution time*. The execution time of algorithm *HD-PI* with different bounding strategies. When we study 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 algorithm *HD-PI* using the bounding ball (rectangle) as a bounding strategy. When we study the second measurement, we additionally included one variant called *HD-PI(NoBall-NoRectangle)* which is algorithm *HD-PI* without using any bounding strategy.

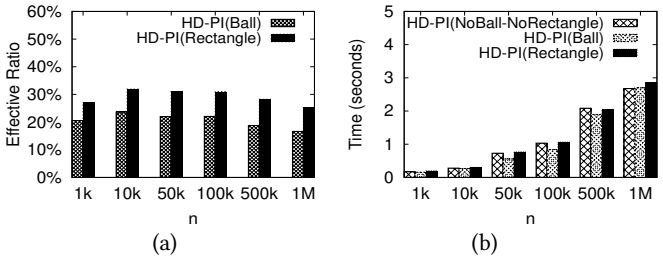

Figure 5: Bounding Strategy Comparison

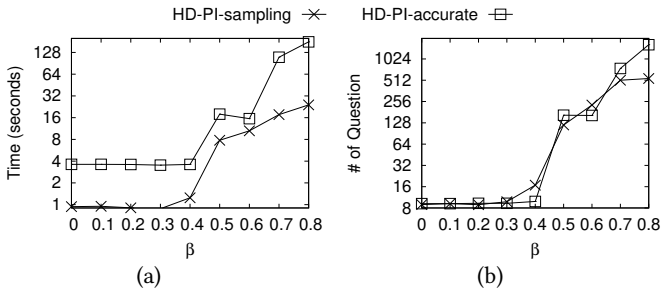

Figure 6: Performances on different β

Figure 5 shows the experimental results. The bounding rectangle strategy identifies more relationships than the bounding ball strategy, where their effective ratio are around 30% and 20%, respectively. However, since the bounding ball strategy only needs $O(1)$ checking time, its time cost is the smallest. Thus, we stick to the bounding ball strategy in the rest of the experiments.

We studied the effect of the parameter β, which is a balancing parameter in the even score, on algorithm *HD-PI* in Figure 6, through evaluating the execution time and the number of questions asked. The execution time and the number of questions increase when β increases. Thus, we set $\beta = 0.01$ in the rest of the experiments.

In Figure 7, we studied the quality of the result of algorithm *HD-PI* with sampling strategy for finding convex points. Following [16, 20], we define the *accuracy* of the returned point p as $\frac{f(p)}{f(p_k)}$ 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. We vary parameter k to see the accuracy of the returned point on different

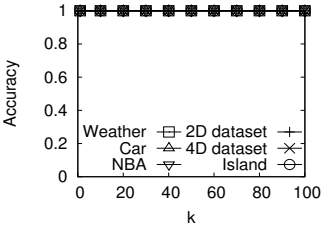

Figure 7: Accuracy

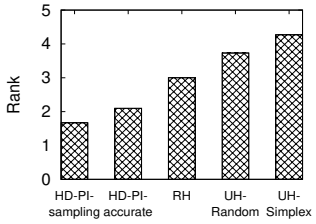

Figure 8: Rank

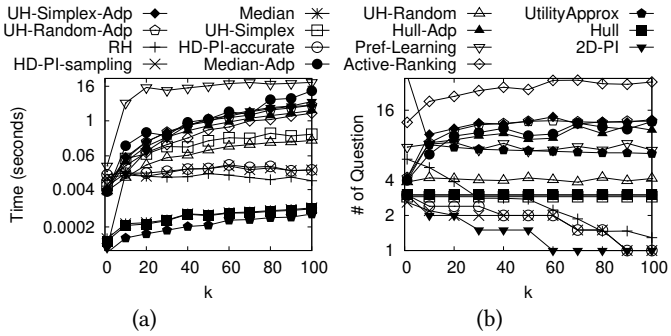

Figure 9: 2-dimensional Synthetic Datasets

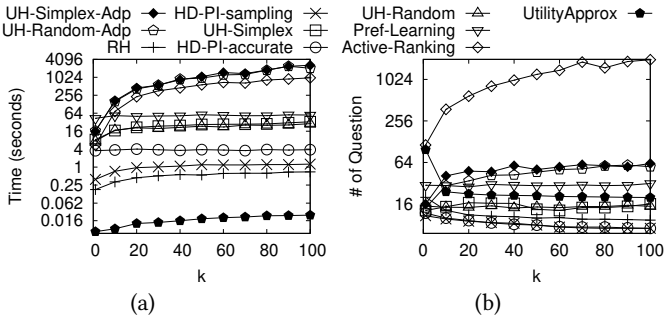

Figure 10: 4-dimensional Synthetic Datasets

datasets. It can be seen that the sampling strategy affects little to the final result.

6.2 Performance on Synthetic Datasets

We compared our 2-dimensional algorithm *2D-PI*, against the existing 2-dimensional algorithms on a 2-dimensional dataset in Figure 9 by varying the parameter k. For completeness, our d-dimensional algorithms *HD-PI* and *RH* and existing d-dimensional algorithms are also compared by setting $d = 2$ (Since their performance is almost the same as that on the 4-dimensional dataset, we analyze them later). Figure 9(a) and (b) show the execution time and the number of questions asked of each algorithm. All the algorithms execute within a few seconds. Note that algorithm *Median* and *Hull* are slightly faster than *2D-PI*. However, they ask many questions. When $k \geq 60$, they ask three times as many questions as *2D-PI*. Although algorithm *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. In particular, it only asks 1 question when $k \geq 60$. For algorithms *Median-Adapt* and *Hull-Adapt*, although the modified stopping condition is easier to achieve, the adaptation on the point deletion condition reduces the effectiveness of deleting points, which results in a large execution time and number of questions (even increase when k increases).

Figures 9 and 10 demonstrate the performance (execution time and number of questions) of our algorithms: *HD-PI* and *RH* with the existing d-dimensional algorithms on 2-dimensional and 4-dimensional datasets. Algorithm *Active-Ranking* requires the largest number of questions with much execution time, e.g. 1000 questions and 100 seconds on a 4-dimensional space, since it learns the ranking of all the points. It even runs gradually slower and asks more questions when k increases, because it is sensitive to the input size (increases with k due to the k-skyband preprocessing). The execution time of algorithms *UH-Random*, *UH-Simplex* and *Preference-Learning* are larger than that of our algorithms. Specifically, *UH-Random* and *UH-Simplex* take 2-10 times as much time as *RH* and *HD-PI*, and the execution time required by *Preference-Learning* is about 2-3 orders of magnitude more than that required by our algorithms on the 2-dimensional dataset. On the 4-dimensional dataset, *UH-Random* and *UH-Simplex* take more than four times and *Preference-Learning* takes more than ten times as much execution time as *RH* and *HD-PI*. 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, when k increases, the number of questions almost remain unchanged. Algorithm *Preference-Learning* also asks comparably number of questions with different k. For algorithms *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 larger execution time and number of 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 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 do not rely on the dataset. However, as we argue in Section 2, displaying fake points is not suitable for real systems. Although *HD-PI* and *RH* take slightly more time than *UtilityApprox*, their execution time are small and reasonable since they use real points and require the least number of questions. When $k = 100$ on the 4-dimensional dataset, the least number of questions required by the existing algorithms is about 15, while our algorithms require only half of the questions. In addition, except *UtilityApprox*, there are no existing algorithms whose number of questions asked decreases significantly when k varies from 1 to 100, while our algorithms *HD-PI* and *RH* could have at least 32% reduction on the number of questions.

According to the experimental results, in the following, we focus on the state-of-art existing d-dimensional algorithms *UH-Random*

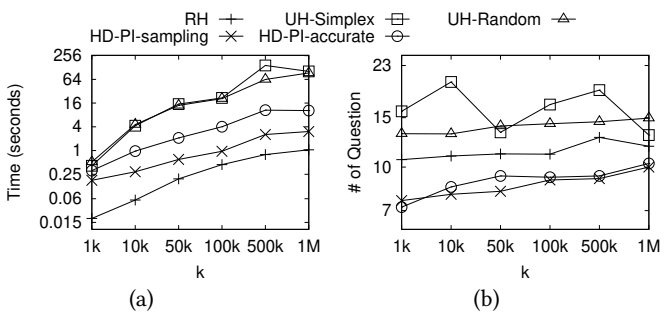

Figure 11: Vary n (k=20, d=4)

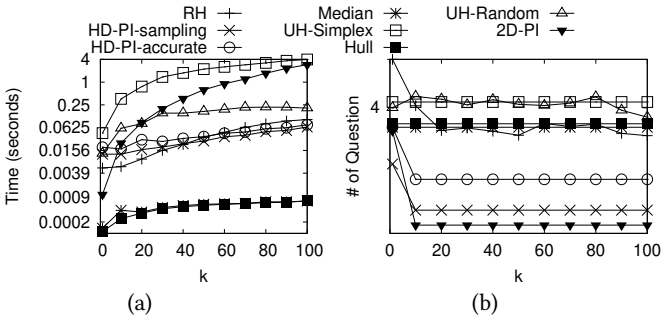

Figure 13: Island

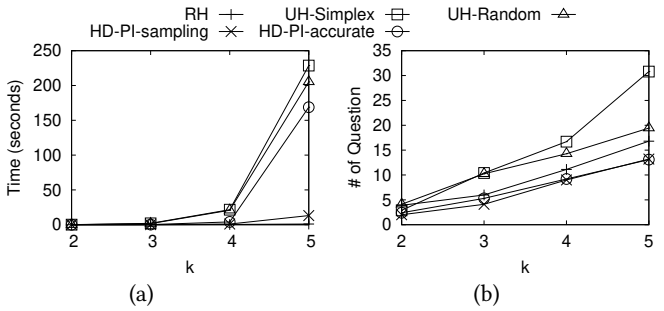

Figure 12: Vary d (k=20, n=100k)

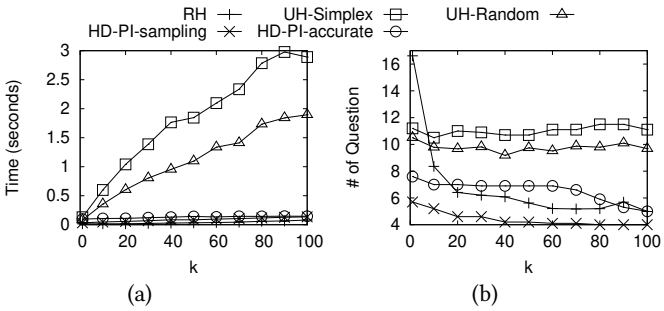

Figure 14: Weather

and *UH-Simplex* for comparison, and include algorithms *Median* and *Hull* for the 2-dimensional case specially.

We evaluated the scalability of each algorithm in Figures 11 and 12. In Figure 11, we concentrated on the scalability on the dimensionality. Compared with *UH-Random* and *UH-Simplex*, *RH* and *HD-PI* consistently requires fewer number of questions and less execution time for any dimension. In particular, when $d = 4$, the number of questions required by *UH-Random* and *UH-Simplex* is around 15 and 17, while both our algorithms need at most 11 questions. On 5-dimensional dataset, *UH-Simplex* and *UH-Random* needs around 230*s* and 205*s*, respectively, while *RH* finishes within only 1 seconds. In Figure 12, we studied the scalability on the dataset size. *HD-PI* and *RH* scale the best with respect to the execution time and the number of questions. In particular, their execution times are less than 10 seconds when the dataset increases, while *UH-Random* and *UH-Simplex* run up to around 100 seconds. Besides, our algorithms ask at least 2 fewer questions than others in all cases.

6.3 Performance on Real Datasets

We studied the performance of our algorithms with the existing algorithms *Median* (for 2D), *Hull* (for 2D), *UH-Random* and *UH-Simplex* on 4 real datasets. The results on dataset *Island* (2D) is shown in Figure 13, where parameter k is varied. 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 $k \geq 20$.

The performance of *RH* and *HD-PI*, on high-dimensional datasets *Weather* (4D), *Car* (4D) and *NBA* (6D), are presented in Figures 14, 15 and 16, respectively. Except that *HD-PI* takes similarly time on the dataset *NBA* with *UH-Random* and *Hull*, our algorithms perform much better than the other algorithms both on the execution time and the number of questions asked for any dataset. In particular, *RH* runs the fastest and *HD-PI* requires the fewest questions. Both *HD-PI* and *RH* need less than 8 questions for all datasets when $k \geq 20$, and if $k \geq 50$, they need nearly half of the number of questions required by *UH-Random* and *UH-Simplex*. As for execution time, *RH* and *HD-PI* take at most 0.08, 0.1, 0.1 seconds and 0.15, 0.22, 0.6 seconds on datasets *Weather*, *Car* and *NBA*, respectively.

6.4 User Study

We also conducted a user study on the dataset *Car* to see the impact of the mistakes on the final results. Following the setting in [18, 20], we randomly selected 1000 cars from the dataset described by 4 attributes, namely price, year of purchase, power and used kilometers. We recruited 30 participants and asked them questions to find a car which is one of the user's top-20 cars ($k = 20$).

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*. Since the user's utility vector is unknown, we adapted each existing algorithm as follows, instead of the way described in Section 6.

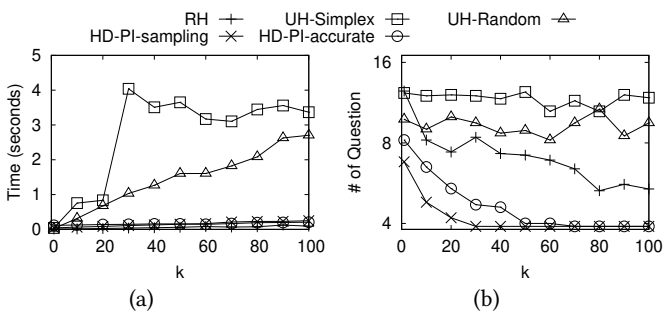

Figure 15: Car

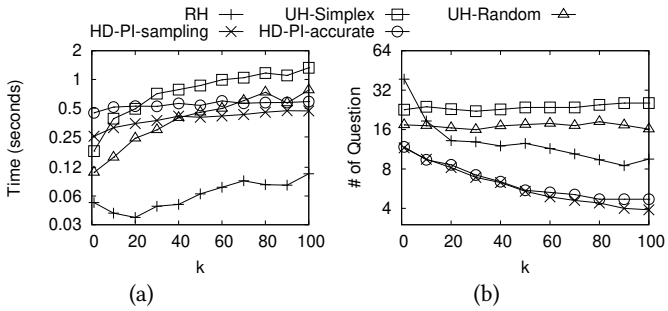

Figure 16: NBA

- *Preference-Learning* maintains an estimated utility vector u_e while interacting with the user. We compared the user's answer of some randomly selected questions with the prediction w.r.t u_e. If 75% comparisons can be correctly identified, we stop and return one of the top-k cars w.r.t u_e.
- For *UH-Random* and *UH-Simplex*, during the experiments shown in Section 6, we find that they never terminate under the stopping condition related to regret ratio, unless the given threshold for regret ratio is very large. Since the threshold will be very small if we follow the setting in Section 6 when $k = 20$, we simply set the threshold of regret ratio as 0.
- Algorithm *Active-Ranking* follows the adaptation in Section 6. It stops when the ranking is obtained and arbitrarily returns one of the top-20 cars.

We measured the performance of each algorithm by the number of questions asked and the degree of boredness evaluated by the user. Specifically, the user gives a score from 1 to 10 to indicate how bored s/he feel when s/he sees the returned car after being asked several questions (10 means the most bored and 1 denotes the least bored). The number of questions and the degree of boredness are shown in Figure 17. Our algorithms require the least number of questions and are the most satisfied by the user. In particular, the number of questions and the degree of boredness of *HD-PI-sampling* (resp., *HD-PI-accurate*) are 4.1, 4.8 (resp., 1.9, 2.13) and for *RH*, they are 7.1 and 3. While the number of questions and the degree of boredness of other existing algorithms *Preference-Learning*, *UH-Simplex*, *UH-Random* are larger than 8.4 and 3.75.

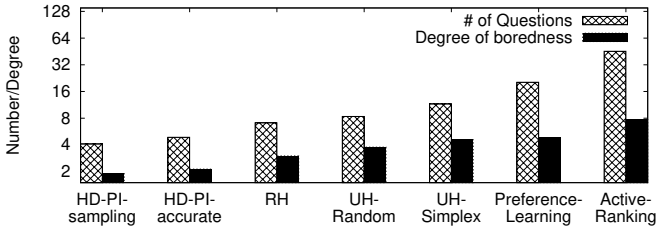

Figure 17: User study

Besides, for the 5 algorithms with the lowest boredness degree: *RH*, *HD-PI* in sampling and accurate versions, *UH-Random* and *UH-Simplex*, the participants sometimes gave the same score for different algorithms. To distinguish them more clearly, we asked the participants to give an order of them based on the number of questions asked and the returned car. The average ranking of each algorithm is shown in Figure 8. Our algorithms rank the best, where the sampling version of *HD-PI* is the most favored by user.

6.5 Summary

The experiments show the superiority of our algorithms over the existing algorithms: (1) Our algorithms are effective and efficient. They are able to ask fewer questions within less time than the existing algorithms (e.g., half of questions on a 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. *UH-Random* and *UH-Simplex* ask around 15 and 17 questions when $n = 1,000,000$, while our algorithms need at most 11 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 2-dimensional space with small time cost (e.g., one third of questions compared with *Median* and *Hull* when $k \geq 60$). In high dimensional space, *RH* requires the least execution time (e.g. it runs within 1 seconds, while *UH-Random* and *UH-Simplex* needs more than 200 seconds) and *HD-PI* needs the fewest number of questions (e.g., half of questions on a 4-dimensional dataset compared with *UH-Random* and *UH-Simplex* when $k \geq 50$).

7 CONCLUSION

We present interactive algorithms for recommending one of the user's top-k points in this paper, pursing less execution time and fewer number of questions asked. In 2-dimensional space, we propose algorithm *2D-PI*, which asks asymptotically number of questions. In d-dimensional space, two algorithms *RH* and *HD-PI* are presented, which focus on execution time and number of questions asked, respectively. For *HD-PI*, depending on the demand on speed and accuracy, we exhibit two versions sampling and accurate. Extensive experiments showed our algorithms are very useful in finding one of the user's top-k points by asking very few questions within little time. As for future work, we consider the situation when the user answers questions mistakenly.

REFERENCES

[1] Arpit Agarwal, Prathamesh Patil, and Shivani Agarwal. 2018. Accelerated Spectral Ranking. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, Stockholmsmässan, Stockholm Sweden, 70–79.

[2] 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 (SIGMOD '19)*. ACM, New York, NY, USA, 263–280.

[3] Anonymous Author(s). 2020. Interactive Top-k Points Searching (technical report). In *Anonymous Link*.

[4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry: Algorithms and Applications* (3rd ed. ed.). Springer-Verlag TELOS, Santa Clara, CA, USA.

[5] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering*. IEEE Computer Society, USA, 421–430.

[6] Wei Cao, Jian Li, Haitao Wang, Kangning Wang, Ruosong Wang, Raymond Chi-Wing Wong, and Wei Zhan. 2017. k-Regret Minimizing Set: Efficient Algorithms and Hardness. In *20th International Conference on Database Theory (ICDT 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Michael Benedikt and Giorgio Orsi (Eds.), Vol. 68. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 11:1–11:19.

[7] Yuxin Chen and Changho Suh. 2015. Spectral MLE: Top-K Rank Aggregation from Pairwise Comparisons. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 371–380.

[8] Herbert Edelsbrunner. 1987. *Algorithms in Combinatorial Geometry*. Springer-Verlag, Berlin, Heidelberg.

[9] Brian Eriksson. 2013. Learning to Top-K Search using Pairwise Comparisons. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Carlos M. Carvalho and Pradeep Ravikumar (Eds.), Vol. 31. PMLR, Scottsdale, Arizona, USA, 265–273.

[10] 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.

[11] Kevin G. Jamieson and Robert D. Nowak. 2011. Active Ranking Using Pairwise Comparisons. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*. Curran Associates Inc., Red Hook, NY, USA, 2240–2248.

[12] Minje Jang, Sunghyun Kim, Changho Suh, and Sewoong Oh. 2016. Top-K Ranking from Pairwise Comparisons: When Spectral Ranking is Optimal. *ArXiv* abs/1603.04153 (2016).

[13] 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 (SIGIR '10)*. Association for Computing Machinery, New York, NY, USA, 904.

[14] Lucas Maystre and Matthias Grossglauser. 2017. Just Sort It! A Simple and Effective Approach to Active Preference Learning. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 2344–2353.

[15] Kyriakos Mouratidis and Bo Tang. 2018. Exact Processing of Uncertain Top-k Queries in Multi-Criteria Settings. In *Proceedings of the VLDB Endowment*.

[16] Danupon Nanongkai, Ashwin Lall, Atish Das Sarma, and Kazuhisa Makino. 2012. Interactive Regret Minimization. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 109–120.

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

[18] Li Qian, Jinyang Gao, and H. V. Jagadish. 2015. Learning User Preferences by Adaptive Pairwise Comparison. *Proc. VLDB Endow.* 8, 11 (2015), 1322–1333.

[19] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. 2007. Top-k Query Processing in Uncertain Databases. In *2007 IEEE 23rd International Conference on Data Engineering*. 896–905.

[20] Min Xie, Raymond Chi-Wing Wong, and Ashwin Lall. 2019. Strongly Truthful Interactive Regret Minimization. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 281–298.

[21] Jiping Zheng and Chen Chen. 2020. Sorting-based Interactive Regret Minimization. arXiv:cs.DB/2006.10949

Notation	Explanation
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
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 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: Notation

Appendix A PROOFS

Proof of Theorem 3.1. aliascntConsider 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 4.1. 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 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.

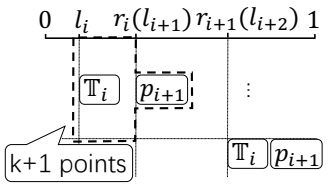

Figure 18: Partition Θ_i and Θ_{i+1}

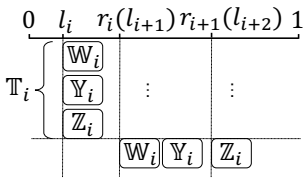

Figure 19: Partition Θ_i and Θ_{i+1}

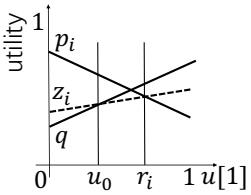

Figure 20: Z_i Case

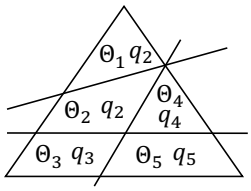

Figure 21: Partition

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 4.2. 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 $j = 1, 3, 5, ...$ 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 18). 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 19, 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 20. 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 5.1. We first show in Lemma A.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)* [2].

Lemma A.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 21, 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. [2] 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 5.2. 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 5.3. 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 $N_{d-1}(\eta) = C_\eta^0 + C_\eta^1 + ... + C_\eta^{d-1}$cells [8].

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 ($i, j \in [1, v]$, $i > j, v \in [2, n]$), [11] 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 $N_{d-1}(\mu) = C_\mu^0 + C_\mu^1 + ... + C_\mu^{d-1}$ 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 [8] (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.

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

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

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

If $2(d-2) > \mu$, the relation between C_μ^φ and C_μ^{d-2} 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. □