Interactive Regret Query Meets High Dimensions

ABSTRACT

The interactive regret query identifies a user's preferred tuple from the database by engaging the user through a series of interactive questions. Existing algorithms for this query maintain a candidate range of the user's preference, and continually refine it based on the user feedback to the questions. When the range is sufficiently small, the user's preference can be approximated, and the preferred tuple can be identified. Unfortunately, existing algorithms struggle when handling high-dimensional tuples. This is because the candidate range involves a convex polytope operation, whose computational cost is exponential to the number of dimensions.

To address this high-dimensional bottleneck, we propose a new algorithm, Approximation Path Construction (APC). Unlike existing algorithms that globally maintain all possible user preferences, APC takes a localized view. It only focuses on a small set of promising candidates of the user's preference. During the interaction, it iteratively explores nearby alternatives to update the candidates based on the user feedback, forming an approximation path towards the user's preference. This shift allows APC to replace expensive global geometric operations with lightweight local explorations. Nevertheless, local exploration presents a new challenge: blindly exploring neighbors may lead to inefficient updates. To overcome it, we introduce a new concept called equally effective, which evaluates neighbors based on whether they yield different potential output tuples. Then, a neighbor is used for update only if its potential output is confirmed to be better than the current one through interaction. In this way, we ensure each update brings an effective approximation to the user's preference. Extensive experiments on both real and synthetic datasets demonstrate that our algorithm APC significantly outperforms existing algorithms, achieving several orders of magnitude speedup in high-dimensional settings.

CCS CONCEPTS

• **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

ACM Reference Format:

. 2018. Interactive Regret Query Meets High Dimensions. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The interactive regret query [34, 38], one of the most representative *multi-criteria decision-making* queries, is widely used in many applications such as car selection, college admissions, or apartment hunting [26, 30–33, 35, 37]. It models each user's preference by a *linear utility function* $f_{\boldsymbol{u}}(\boldsymbol{p}) = \boldsymbol{u} \cdot \boldsymbol{p}$, where $\boldsymbol{p}$ represents the tuple and $\boldsymbol{u}$, called the *utility vector*, quantifies the user's weighting of each dimension. The goal is to identify a tuple from a multi-dimensional database whose function value is *close* to the highest, i.e., satisfies the *regret* requirement. The core part of this query is that the utility function (i.e., the utility vector) is not explicitly provided. Instead, as shown in Figure 1, the system interacts with the user through a series of pairwise comparisons between tuples, to gradually learn the user's utility vector and ultimately return the desired tuple.

Throughout the query process, the challenge lies in effectively learning the user's utility function, i.e., the user's utility vector. In recent years, the SOTA algorithms are UH-Random and UH-Simplex [34]. These two algorithms share a common idea. They map the utility vector $\boldsymbol{u}$ as a point in the geometric space, and maintain a range $\mathcal{R}$, which contains all candidates that could represent the user's utility vector. Once a user feedback is obtained, e.g., a tuple $\boldsymbol{p}_i$ is more preferred than $\boldsymbol{p}_j$, they build a linear constraint $f_{\boldsymbol{u}}(\boldsymbol{p}_i) > f_{\boldsymbol{u}}(\boldsymbol{p}_j)$, i.e., $\boldsymbol{u} \cdot \boldsymbol{p}_i > \boldsymbol{u} \cdot \boldsymbol{p}_j$ to shrink the range $\mathcal{R}$. As the interaction continues, in Figure 2, more constraints are built and the range $\mathcal{R}$ progressively shrinks. When $\mathcal{R}$ becomes sufficiently small, any utility vector within it will serve as a good approximation of the user's utility vector. Consequently, the algorithms return a tuple with a high function value based on any vector sampled from $\mathcal{R}$.

Unfortunately, this idea is restricted to low-dimensional tuples. As demonstrated in [34], the experiments are conducted on the tuples with only 4-5 dimensions. This is because the range $\mathcal{R}$ is defined by a set of linear constraints in the geometric space. Shrinking $\mathcal{R}$ involves a *convex polytope* operation [8], which is computationally expensive. In the worst case, the time complexity is $O(l^{\lfloor d/2 \rfloor})$ [8], where l denotes the number of constraints and d is the number of dimensions. This implies an exponential growth with respect to d. To empirically validate this issue, in Figure 3, we conducted an experiment on an M3 chip. We fixed the number of constraints at 50 and recorded the computation time as the dimension increased. As can be noticed, the time grows dramatically with the number of dimensions, e.g., at $d = 16$, it exceeds 10^4 seconds, which is unacceptable.

Even worse, existing algorithms rely on shrinking the range $\mathcal{R}$ to approximate the user's utility vector. As the number of dimensions increases, the volume of the geometric space, and thus, the initial volume V of the range $\mathcal{R}$, grows exponentially. It necessitates a large number of linear constraints to shrink $\mathcal{R}$. Even under an idealized scenario where each linear constraint halves the remaining volume of $\mathcal{R}$, it would still require approximately $\log_2(V/\alpha)$ constraints to reduce the volume of $\mathcal{R}$ below α. As a result, the range $\mathcal{R}$ needs to be updated multiple times, and these updates face a growing number of constraints l, leading to severe computational bottlenecks.

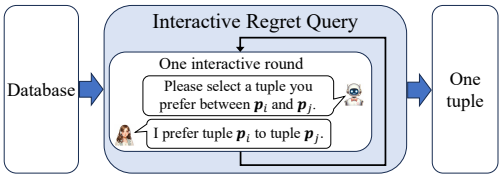

Figure 1: Interactive Regret Query

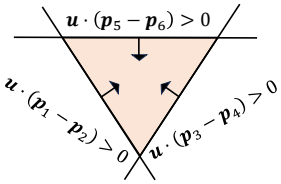

Figure 2: Range $\mathcal{R}$

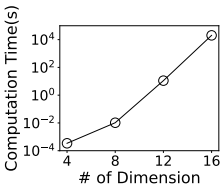

Figure 3: Computation Time

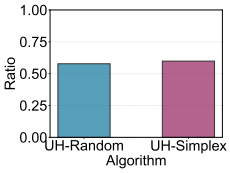

Figure 4: Reduction

In practice, users typically care about a wide range of aspects, which naturally involve multiple dimensions of tuples. For example, when purchasing a car, a user may consider price, horsepower, fuel efficiency, interior space, maximum speed, safety, etc. The number of dimensions easily exceeds 5. Even when users interact with low-dimensional tuples, the systems may sometimes perform computations in a high-dimensional embedding space (e.g., learned representations). These scenarios call for extending the interactive regret queries to be more general, i.e., handling high-dimensional settings.

A naive idea is to first reduce the dimensionality and then apply existing algorithms. In Figure 4, we employed PCA [16] to reduce a 30-dimensional anti-correlated dataset [34] to 4 dimensions and applied the two algorithms in [34]. We measured the ratio between the function value of the returned tuple and that of the desired tuple. As can be noticed, the returned tuple deviates substantially from the expected one, achieving only about 60% of the desired function value. This suggests that simple dimensionality reduction fails to preserve the critical information for reliable interaction.

Motivated by this negative result, we turn to developing algorithms that operate directly in high dimensions rather than relying on dimension reduction. Essentially, existing algorithms stand from a global perspective. The range $\mathcal{R}$ always covers *all* possible candidates of the user's utility vector. However, the computational overhead of $\mathcal{R}$ in high dimensions becomes prohibitive. This drives us to think: *can we approximate the user's utility vector from a local perspective without involving $\mathcal{R}$ to reduce computational overhead?*

Our idea is to consider only a *subset* of promising utility vectors as candidates, serving as representative hypotheses of the user's utility vector. During the interaction, we iteratively explore new utility vectors $\boldsymbol{u}_j$ from the local neighborhood of the current candidates $\boldsymbol{u}_i$, and decide whether to update $\boldsymbol{u}_i$ by $\boldsymbol{u}_j$ based on the user feedback. These updates locally refine the candidates, forming an approximation path toward the user's utility vector. Although reasonable, this idea is nontrivial to apply. The core difficulty lies in identifying which neighboring utility vector $\boldsymbol{u}_j$ to explore. Consider Figure 5 to illustrate. The red point represents the user's utility vector $\boldsymbol{u}^*$, and the black circle denotes the current candidates $\boldsymbol{u}_i$. We may observe multiple neighboring utility vectors of $\boldsymbol{u}_i$, such as $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, either of which may eventually lead to the user's utility vector. However, their corresponding approximation paths differ in the number of updates. One quickly reaches the target, with each update in the path steadily approaching the user's utility vector $\boldsymbol{u}^*$. In contrast, the other takes a detour. Some updates may even move away from $\boldsymbol{u}^*$, resulting in a large number of updates.

The fundamental obstacle lies in the fact that the user's utility vector $\boldsymbol{u}^*$ is unknown, and only local information is available. When identifying utility vectors, e.g., $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, there is no reliable way to evaluate which one is closer to $\boldsymbol{u}^*$. As a result, the exploration is blind: the selected neighbors may move away from the target, inadvertently leading to a long and inefficient path. This issue is further exacerbated in high-dimensional settings, where the number of possible neighboring utility vectors grows rapidly, making it even more challenging to avoid poor approximation paths.

To overcome this obstacle, our idea is to utilize tuples for evaluating the neighboring utility vector $\boldsymbol{u}_j$, rather than solely considering utility vectors themselves, since we cannot directly evaluate their proximity to the user's utility vector. This shift stems from the goal of the interactive regret query, which is to return a tuple. Following this insight, we define a new concept for evaluating utility vectors, called *equally effective*. Intuitively, each utility vector is associated with a set of tuples that satisfy the regret requirement w.r.t. it. We say two utility vectors are equally effective if their associated sets share at least one common tuple, i.e., both utility vectors can use the same tuple as the output. This concept gives us a concrete operational rule. When exploring the neighboring utility vector $\boldsymbol{u}_j$ of the current candidates $\boldsymbol{u}_i$, we require $\boldsymbol{u}_j$ to be non-equally effective to $\boldsymbol{u}_i$. In this way, if $\boldsymbol{u}_i$ is updated to be $\boldsymbol{u}_j$, the associated set is fully changed, ensuring the update is effective: (1) the potential output changes, avoiding redundant steps; and (2) the new potential output is better than the previous one, implying that $\boldsymbol{u}_j$ is better than $\boldsymbol{u}_i$ to approximate $\boldsymbol{u}^*$, making tangible approximation progress. The interaction terminates once all neighbors are equally effective (i.e., all non-equally effective neighbors have been checked and ruled out based on user feedback). At this point, all neighboring utility vectors yield the same output tuple, and no further update is necessary.

Following this idea, we propose an algorithm *approximation path construction*, APC for short, that aims to construct a path based on the equally effective evaluation, gradually approximating the user's utility vector. To further enhance the performance of APC, we design several strategies, which can be broadly categorized into two groups. The first group focuses on computational acceleration (e.g., dimension reduction), while the second group aims to reduce the number of interactive rounds (e.g., path shortening).

To the best of our knowledge, we are the first to consider the interactive regret query in the high-dimensional setting. Our contributions are summarized as follows.

- We consider the interactive regret query in the high-dimensional scenario and discuss the issues of existing algorithms.
- We present an approximation path construction algorithm, called APC for short. It scales well in high dimensions with a few interactive rounds in a small execution time.
- Extensive experiments were conducted on both synthetic and real datasets. The results show that our algorithm is able to handle datasets with about two orders of magnitude more dimensions than the SOTA algorithm.

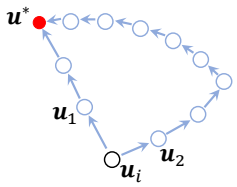

Figure 5: Local Update

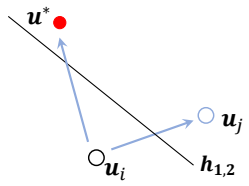

Figure 6: Detour

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 formally defines our problem. Section 4 discusses the motivation and Section 5 presents our algorithm. Section 6 demonstrates our experimental results, and finally, Section 7 concludes the paper with possible future work.

2 RELATED WORK

Non-interactive query. The top-k query and the skyline query utilize utility functions to model users' preferences. Based on the function, each tuple $\boldsymbol{p}$ is assigned a *utility* (i.e., a function value), quantifying its alignment with the user's preference. The top-k query retrieves the k tuples with the highest utilities. Despite its simplicity, this query heavily relies on the assumption that users can explicitly specify their utility functions (i.e., provide parameters of the function), which is rarely feasible in practical [18, 34]. The skyline query circumvents the need for utility function specification. It considers all possible utility functions and returns tuples with the highest utilities w.r.t. at least one utility function. Nevertheless, it is limited by the uncontrollable output size [18, 19]. It is possible that each tuple has the highest utility w.r.t. one utility function, and thus, the entire database is returned.

The regret query [3, 19, 22] seeks a set S of tuples such that, for any utility function, there is at least one tuple in S whose utility is *close* to the optimal tuple in the database. The closeness is measured through a concept called *regret ratio*, which is the relative utility difference between tuples. This query addresses the limitations of the top-k query and the skyline query. It ensures that for any utility function, there is a good tuple in the output S, eliminating the necessity for users to specify their utility functions; it sets the size of S to a fixed integer, ensuring a bounded output size. Nonetheless, since it considers all utility functions, an intrinsic challenge arises from the trade-off between the regret ratio and output size. When the output size is small, the regret ratio is typically large [3, 13, 34].

Interactive query. The interactive regret query enhances the regret query by integrating user interaction. Through iterative interactions, it progressively learns the user's preference, i.e., refines the possible range of utility functions. Consequently, based on this refined range, it returns a tuple set $\mathcal{S}$ with a minimized regret ratio, where $|\mathcal{S}|$ is typically set to be 1. Several interactive algorithms have been developed to address this query [18, 25, 34, 38].

[18] proposes the first algorithm UtilityApprox. In each interactive round, it constructs several artificial tuples and asks the user to indicate which one s/he prefers. By strategically designing these artificial tuples, it efficiently captures user preferences. However, since these tuples are artificially constructed (i.e., tuples not selected from the database), they might be unrealistic (e.g., a car at 10 dollars with 50000 horsepower). The user would be disappointed if some tuples displayed during the interaction do not exist [34].

To address this limitation, [34] proposes algorithms UH-Random and UH-Simplex that utilize *real* tuples (i.e., tuples selected from the database) for interaction. Specifically, in each interactive round, UH-Random randomly selects tuples from the database, while UH-Simplex selects tuples from the database that are likely to be optimal based on some criteria. Nevertheless, both algorithms involve costly operations, e.g., the *convex hull* computation, resulting in prolonged execution times. To mitigate this issue, [38] introduces algorithm SinglePass, which reduces the computation workload by collecting less information per interactive round. But this requires substantially more questions (e.g., hundreds of questions). [25] proposes reinforcement learning (RL)-based algorithms EA and AA for the interactive regret query. However, the RL-based algorithms necessitate offline training in advance, making them unsuitable for online real-time scenarios. Moreover, one common and critical issue with these interactive algorithms is that their computational complexity escalates dramatically with the increasing dimensionality. They generally perform effectively when handling low-dimensional datasets (e.g., 5 dimensions). As the dimensions increase, these algorithms may take several hours or even days to execute.

There are several relevant interactive algorithms designed for other queries. [28, 29] propose algorithms that return tuples whose utilities are among the top-k. Further advances by [26, 30] incorporate different types of dimensions into the query. However, these algorithms primarily emphasize ranking tuples, which are secondary information derived from utilities. In contrast, we directly target the utility difference from the best tuple in the database.

In the field of machine learning, the interactive regret query closely relates to the problem of *preference learning* [24] and *learning to rank* [9, 11, 15, 17]. [24] proposes an algorithm that emphasizes learning user preferences through user interaction. Nevertheless, it prioritizes deriving the user's preference rather than returning desired tuples, potentially leading to redundant questions. For instance, if Alice prefers tuple $\boldsymbol{p}_1$ to both $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$, further interaction involving $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$ is less interesting in our problem, but could be beneficial in [24]. As for learning to rank, existing algorithms [9, 15, 17] often neglect dimension interrelation, resulting in asking many questions. This interrelation refers to meaningful ordering relationships of values in dimensions. For instance, consider two tuples that differ only in price. The one with \$200 is inherently preferable to the one with \$500, due to lower cost. [11] accounts for dimension interrelations to improve tuple ranking, but it still asks redundant questions for the same reason stated for [24].

3 PROBLEM DEFINITION

Input. The input consists of a set $\mathcal{D}$ of n tuples. Each tuple is associated with d dimensions and can be regarded as a d-dimensional point $\boldsymbol{p} = (p[1], p[2], ..., p[d])$, where $p[i]$ ($i \in [1, d]$) denotes the i-th dimension value of $\boldsymbol{p}$. Without loss of generality, we assume that each dimension is normalized to $(0, 1]$ and larger values are preferred [34, 36]. Table 1 shows an example, where $\mathcal{D}$ has five 2-dimensional points. In the following, we use "tuple/point" interchangeably. Frequently used notations are summarized in Appendix A.

Table 1: Database ($\boldsymbol{u} = (0.4, 0.6)$)

$\boldsymbol{p}$	$p[1]$	$p[2]$	$f_{\boldsymbol{u}}(\boldsymbol{p})$
$\boldsymbol{p}_1$	0.9	0.2	0.48
$\boldsymbol{p}_2$	0.5	0.7	0.62
$\boldsymbol{p}_3$	0.4	0.8	0.64
$\boldsymbol{p}_4$	0.3	1.0	0.72
$\boldsymbol{p}_5$	0.8	0.3	0.50

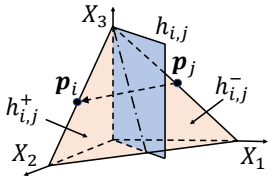

Figure 7: Utility Space and Hyper-plane

Utility Function. Following [1, 25, 29], the user preference is modeled through a linear function $f_{\boldsymbol{u}}$, called the *utility function*.

$$f_{\boldsymbol{u}}(\boldsymbol{p}) = \boldsymbol{u} \cdot \boldsymbol{p} = \sum_{i=1}^{d} u[i]p[i]$$

- $\boldsymbol{u} = (u[1], ..., u[d])$ is called *utility vector*, representing the user's weighting of each dimension. Each $u[i]$ is non-negative and a larger $u[i]$ implies higher importance of the i-th dimension to the user, where $i \in [1, d]$. Without loss of generality, following [18, 25, 27, 34], we assume that $\sum_{i=1}^{d} u[i] = 1$.
- Function value $f_{\boldsymbol{u}}(\boldsymbol{p})$ is called the *utility* of $\boldsymbol{p}$ w.r.t. $\boldsymbol{u}$. It quantifies how desirable the point $\boldsymbol{p}$ is to the user. The point with the highest utility is regarded as the user's favorite point, i.e., top-1 point.

Example 3.1. Consider the points in Table 1 and assume that the utility function is $f_{\boldsymbol{u}}(\boldsymbol{p}) = 0.4p[1] + 0.6p[2]$, i.e., utility vector $\boldsymbol{u} = (0.4, 0.6)$. The utility of $\boldsymbol{p}_4$ w.r.t. $\boldsymbol{u}$ is $f_{\boldsymbol{u}}(\boldsymbol{p}_4) = 0.3 \times 0.4 + 1.0 \times 0.6 = 0.72$. The utilities of other points are computed similarly. Point $\boldsymbol{p}_4$ with the highest utility is the user's favorite point.

Regret Ratio [27, 34]. Given a dataset $\mathcal{D}$ and a utility function $f_{\boldsymbol{u}}$, the regret ratio of a point $\boldsymbol{q} \in \mathcal{D}$ over $\mathcal{D}$ w.r.t. $f_{\boldsymbol{u}}$ is defined as:

$$regratio(\boldsymbol{q}, \boldsymbol{u}) = \frac{max_{\boldsymbol{p} \in \mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p}) - f_{\boldsymbol{u}}(\boldsymbol{q})}{max_{\boldsymbol{p} \in \mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p})}$$

Intuitively, the regret ratio of point $\boldsymbol{q}$ measures how far point $\boldsymbol{q}$ is from being the user's favorite point. It is the ratio of the difference between the utility of point $\boldsymbol{q}$ and the highest utility in the dataset $\mathcal{D}$. If $regratio(\boldsymbol{q}, \boldsymbol{u})$ is below a small threshold, we regard point $\boldsymbol{q}$ as performing nearly as well as the user's favorite point.

Example 3.2. Continue Example 3.1, where $\boldsymbol{u} = (0.4, 0.6)$. The regret ratio of $\boldsymbol{p}_3$ is $regratio(\boldsymbol{p}_3, \boldsymbol{u}) = \frac{0.72 - 0.64}{0.72} = 0.11$.

The regret ratio is a measurement to evaluate the points. It plays a role similar to the ranking position of tuples, but is more informative. As discussed in [25, 27], the regret ratio directly reflects the utility difference between tuples. In contrast, ranking can be coarse: two tuples may have very different utilities yet appear adjacent in the ranking, making it less sensitive for measuring tuples.

Interactive Regret Query with High Dimension (IRHD). We now present the interactive regret query [34, 38] in *high-dimensional scenarios*. Given a high-dimensional dataset $\mathcal{D}$ and a threshold $\epsilon > 0$, the query aims to interact with the user to progressively learn the user's utility vector $\boldsymbol{u}$, and ultimately return a point from $\mathcal{D}$ whose regret ratio w.r.t. $\boldsymbol{u}$ is guaranteed to be below ϵ. Here, we emphasize that ϵ is a problem parameter rather than a method parameter. It cannot be tuned to optimize the performance of algorithms. Instead, it is specified by the user to indicate the desired output. The interaction engages the user through a sequence of *rounds*, each comprising the following three components.

- *Question Selection.* According to previous interactive rounds, two points $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ are selected from $\mathcal{D}$ to formulate a question. The user is asked to tell his/her preferred one between these two.
- *Information Maintenance.* Upon receiving the user's feedback, the maintained information for approximating the user's utility vector is updated. We use $\boldsymbol{p}_i \succ \boldsymbol{p}_j$ to denote the user prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$. Note that the user always interacts with tuples $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ in the original space. But the system may conduct computations in either the original space or the embedding space of much higher dimensionality (e.g., learned representations). In the following, for ease of illustration, we assume that the user and the system share the same tuple dimensionality.
- *Stopping Condition.* If sufficient information is collected, the desired point can be identified and returned. Otherwise, the next interactive round proceeds.

It is worth emphasizing that a fundamental difference of our focus from existing ones [18, 25, 34, 38] lies in that we consider the interactive regret query in *high-dimensional settings*, whereas existing ones mainly focus on low dimensions (typically 2–5 dimensions). In high-dimensional settings, calculating and acquiring information are significantly challenging, as mentioned in Section 1. This motivates us to design an efficient interactive algorithm that can, in high-dimensional scenarios, (1) strategically select points from $\mathcal{D}$ for interaction; (2) effectively collect information for approximating the user's utility vector; and (3) appropriately design a stopping condition based on the approximated utility vector.

4 MOTIVATION

The existing SOTA interactive algorithms [25, 29, 34] share a common idea to approximate the user's utility vector through interaction. In this section, we first revisit this idea and identify its issues. Then, we introduce the motivation of our algorithm.

Existing algorithms view the interaction from a geometric perspective. There are two concepts: *utility space* and *half-space*.

Utility Space. They regard the utility vector $\boldsymbol{u}$ as a point in the d-dimensional geometric space $\mathbb{R}^d$. The domain of $\boldsymbol{u}$ is called the *utility space* and denoted by $\mathcal{U}$. As introduced in Section 3, each $u[i]$, where $i \in [1, d]$, is non-negative and collectively, they satisfy $\sum_{i=1}^{d} u[i] = 1$. These constraints restrict the utility space $\mathcal{U}$ to be a *polyhedron* [8] in space $\mathbb{R}^d$. For example, when $d = 3$, as shown in Figure 7, the utility space $\mathcal{U}$ is a triangle (the red part) in space $\mathbb{R}^3$.

Half-space. In space $\mathbb{R}^d$, as depicted in Figure 7 (the blue part), each pair of points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$ forms a *hyper-plane* [8]

$$h_{i,j} = \{\boldsymbol{r} \in \mathbb{R}^d \mid \boldsymbol{r} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) = 0\},$$

which passes through the origin with its unit normal aligns with $\boldsymbol{p}_i - \boldsymbol{p}_j$. The hyper-plane divides the utility space $\mathcal{U}$ into two halves, called *half-spaces* [8]. The half-space above (resp. below) $h_{i,j}$, denoted by $h^+_{i,j}$ (resp. $h^-_{i,j}$), contains all utility vectors $\boldsymbol{u} \in \mathbb{R}^d$ satisfying $\boldsymbol{u} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) > 0$ (resp. $\boldsymbol{u} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) < 0$). This said, if the user's utility vector $\boldsymbol{u}^*$ lies in $h^+_{i,j}$ (resp. $h^-_{i,j}$), point $\boldsymbol{p}_i$ yields a higher (resp. lower) utility than point $\boldsymbol{p}_j$ w.r.t. $\boldsymbol{u}^*$ [34]. Note that the unit norm of the hyper-plane is towards the positive half-space.

Combining these two concepts, the following lemma lays the foundation for existing algorithms to learn the user's utility vector.

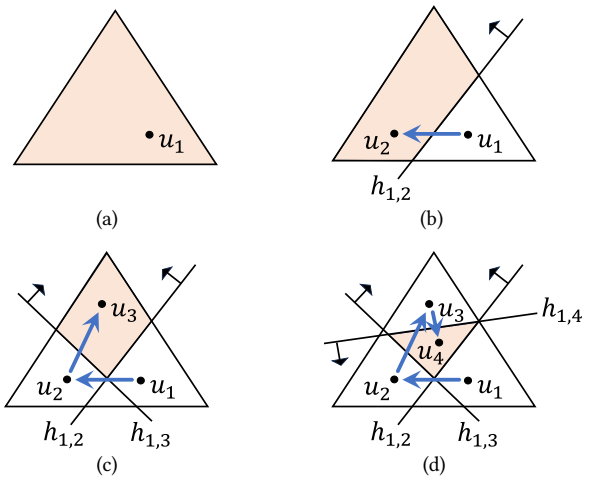

Figure 8: Utility Space Partitioning

Lemma 4.1 ([34, 38]). *Given $\mathcal{U}$ and two points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$ that are presented to a user in an interactive round, the user's utility vector is in $h^+_{i,j} \cap \mathcal{U}$ if and only if the user prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$.*

Building upon Lemma 4.1, existing algorithms leverage half-spaces to shrink the utility space $\mathcal{U}$, refining the location of the user's utility vector. They maintain a polyhedron, termed *utility range* and denoted by $\mathcal{R}$, which is guaranteed to contain the user's utility vector throughout the interaction. Initially, $\mathcal{R}$ is set to $\mathcal{U}$. In each interactive round, two points $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ are selected from the database for interaction. Based on these two points, a hyper-plane $h_{i,j}$ is built. If the user prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, i.e., $\boldsymbol{p}_i \succ \boldsymbol{p}_j$ (resp. the user prefers $\boldsymbol{p}_j$ to $\boldsymbol{p}_i$, i.e., $\boldsymbol{p}_j \succ \boldsymbol{p}_i$), the utility space is updated with the corresponding half-space $h^+_{i,j} \cap \mathcal{R}$ (resp. $h^+_{j,i} \cap \mathcal{R}$). As the interaction proceeds, more half-spaces are involved and $\mathcal{R}$ becomes gradually smaller. Once $\mathcal{R}$ shrinks to be sufficiently small, the user's utility vector is considered to be accurately approximated. Then, the desired point can be found based on the approximated utility vector.

Example 4.2. Take the scenario where $d = 3$ to illustrate. Initially, as shown in Figure 8(a), $\mathcal{R}$ is set to be the whole utility space. In the first interactive round, suppose that points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ are presented as a question, and the user prefers $\boldsymbol{p}_1$ to $\boldsymbol{p}_2$. As depicted in Figure 8(b), a hyper-plane $h_{1,2}$ is built, where the unit norm $\boldsymbol{p}_1 - \boldsymbol{p}_2$ is towards the positive half-space. The utility range is updated to be $\mathcal{R} = h^+_{1,2} \cap \mathcal{U}$ (the red part). Subsequent rounds are similar. Each user feedback introduces a new hyper-plane (e.g., $h_{1,3}$ and $h_{1,4}$), incrementally shrinking the utility range $\mathcal{R}$, as shown in Figures 8(c) and (d). Once $\mathcal{R}$ becomes sufficiently small, the user's utility vector can be approximated. The interaction terminates, and a point is selected from $\mathcal{D}$ based on the approximated utility vector from $\mathcal{R}$.

While effective in low-dimensional settings, existing algorithms struggle when facing high-dimensional scenarios. This is because of the utility range $\mathcal{R}$. During the interaction, $\mathcal{R}$ is used to (1) guide interactive question selection (a question is useful only if its corresponding hyper-plane intersects $\mathcal{R}$; otherwise, it does not help to shrink $\mathcal{R}$) and (2) to determine whether to terminate the interaction process. In each interactive round, existing algorithms update the utility range $\mathcal{R}$ based on the user feedback. As mentioned in Section 1, this operation involves the convex polytope computation, which is known to be exponential to the number of dimensions d [8]. As dimensionality increases, the computational cost of maintaining $\mathcal{R}$ becomes prohibitive, severely limiting the scalability and efficiency of these algorithms. Although [34] proposes strategies, trying to bypass $\mathcal{R}$ when selecting questions, these strategies rely on some necessary conditions, and thus, $\mathcal{R}$ is still required in many cases. Moreover, $\mathcal{R}$ is also needed to judge the termination.

This drives us to pursue a different strategy, bypassing the computation of $\mathcal{R}$ altogether. Essentially, existing algorithms adopt a *global* strategy to approximate the user's utility vector. The utility range $\mathcal{R}$ covers all possible candidates of the user's utility vector based on the collected user feedback. This observation motivates us to ask a fundamental question: if maintaining all candidate utility vectors is expensive, can we instead focus on a small subset of candidates (e.g., one) that *locally* approximate the user's utility vector?

Following this idea, we shift our strategy to maintain only one promising candidate, serving as a representative hypothesis of the user's utility vector. During the interaction, we iteratively explore the neighborhood of the current candidate $\boldsymbol{u}_i$, and evaluate whether the candidates can be updated by any neighboring utility vector $\boldsymbol{u}_j$ based on the user feedback. If so, the candidates are refined. This local refinement is continually repeated, implicitly constructing an approximation path toward the user's utility vector.

Example 4.3. Consider the case where $d = 3$. As shown in Figure 8(a), we start by randomly selecting a utility vector $\boldsymbol{u}_1$ in the utility space as the initial candidate. In the first round, we explore the neighborhood of $\boldsymbol{u}_1$. Suppose we find a neighboring utility vector $\boldsymbol{u}_2$, and learn the user prefers $\boldsymbol{p}_1$ to $\boldsymbol{p}_2$. Since $f_{\boldsymbol{u}_1}(\boldsymbol{p}_1) > f_{\boldsymbol{u}_1}(\boldsymbol{p}_2)$ and $f_{\boldsymbol{u}_2}(\boldsymbol{p}_1) < f_{\boldsymbol{u}_2}(\boldsymbol{p}_2)$, i.e., $\boldsymbol{u}_2$ aligns with the user's preference but $\boldsymbol{u}_1$ does not, we refine $\boldsymbol{u}_1$ to be $\boldsymbol{u}_2$ in Figure 8(b). This process continues with further updates, as demonstrated in Figure 8(c) and (d), forming an approximation path $\boldsymbol{u}_1 \rightarrow \boldsymbol{u}_2 \rightarrow \boldsymbol{u}_3 \rightarrow \boldsymbol{u}_4$ towards the user's utility vector.

As can be noticed, this local strategy eliminates the need to globally compute $\mathcal{R}$. It restricts each update of the candidates to the localized neighborhood, significantly reducing computational overhead.

5 ALGORITHM

In this section, we first present our core algorithm for constructing the approximation path in Section 5.1. Then, in Section 5.2, we introduce several strategies to enhance it, including the way to reduce the computational cost and shorten the path length. Finally, we summarize the algorithm along with these strategies, and provide its theoretical analysis in Section 5.3.

5.1 Approximation Path Construction

The approximation path construction involves two key operations. The first operation is to identify which neighboring utility vectors of the current candidates to explore. The second operation is to decide whether to update the current candidates by the neighboring utility vectors based on the user feedback. The construction process iterates these two operations, gradually forming an approximation path toward the user's utility vector $\boldsymbol{u}^*$.

Operation One. It is challenging to identify which neighboring utility vectors to explore. Consider the example shown in Figure 6.

The user's utility vector $\boldsymbol{u}^*$ lies at the red point, and the black circle denotes the current candidate $\boldsymbol{u}_i$. A neighboring utility vector $\boldsymbol{u}_j$ appears to be plausible and is a valid option to explore. In this case, two points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$, where $\boldsymbol{p}_1$ has a higher utility w.r.t. $\boldsymbol{u}_i$ and $\boldsymbol{p}_2$ has a higher utility w.r.t. $\boldsymbol{u}_j$, are used for interaction. If the user prefers $\boldsymbol{p}_2$ to $\boldsymbol{p}_1$, the candidate is updated from $\boldsymbol{u}_i$ to $\boldsymbol{u}_j$. As can be noticed, this update moves away from the user's utility vector $\boldsymbol{u}^*$, forming a detour towards $\boldsymbol{u}^*$.

The fundamental obstacle to identifying neighboring utility vectors lies in the fact that the user's utility vector $\boldsymbol{u}^*$ is unknown, and only the local neighborhood is observable. We cannot directly evaluate the proximity of the utility vectors to $\boldsymbol{u}^*$. As a result, when selecting neighboring utility vectors, it is unclear whether they move closer to $\boldsymbol{u}^*$ or away from it, making the exploration inherently blind. This obstacle is further exacerbated in high-dimensional settings, where the number of neighbors increases exponentially, making it increasingly difficult to avoid poor approximation paths.

To overcome this obstacle, we propose to shift our evaluation focus. The goal of the interactive regret query is to return a tuple. Thus, instead of solely considering the utility vectors (i.e., the proximity of the utility vector to $\boldsymbol{u}^*$), we involve tuples to evaluate the neighboring utility vectors. In this view, what matters is whether the potential output of the current candidate is close to that of the user's utility vector $\boldsymbol{u}^*$. Following this insight, we define a new concept, called *equally effective*.

Definition 5.1 (Equally Effective). Given two utility vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ in $\mathcal{U}$, they are said to be equally effective if there exists a point $\boldsymbol{p} \in \mathcal{D}$ whose regret ratios w.r.t. $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are smaller than ϵ, i.e., $regratio(\boldsymbol{p}, \boldsymbol{u}_1) < \epsilon$ and $regratio(\boldsymbol{p}, \boldsymbol{u}_2) < \epsilon$.

Intuitively, if two utility vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are equally effective, there exists a common point $\boldsymbol{p}$ that satisfies the regret ratio requirement w.r.t. both. As a result, regardless of whether the user's utility vector is approximated by $\boldsymbol{u}_1$ or $\boldsymbol{u}_2$, point $\boldsymbol{p}$ would remain a valid output. This implies that moving from $\boldsymbol{u}_1$ to $\boldsymbol{u}_2$ (or from $\boldsymbol{u}_2$ to $\boldsymbol{u}_1$) does not introduce actual progress in the potential output (since point $\boldsymbol{p}$ can always be returned). This concept gives us a concrete rule when exploring neighboring utility vectors. We require the utility vectors to be non-equally effective to the current candidates. This ensures that if the candidates $\boldsymbol{u}_i$ are updated to be $\boldsymbol{u}_j$, the potential output is altered, thereby avoiding redundant updates that yield the same potential output. Moreover, as will be discussed in the next operation, the update is conducted only if the new potential output is better than the previous one. This implies that $\boldsymbol{u}_j$ is better than $\boldsymbol{u}_i$ to approximate $\boldsymbol{u}^*$, making tangible approximation progress.

The next problem is how to efficiently identify utility vectors that are equally effective, since it requires considering the regret ratios across the whole dataset. To address this, we take a relaxed approach: we focus on the utility vectors that are equally effective w.r.t. a specific point $\boldsymbol{p}$. Formally, given a point $\boldsymbol{p} \in \mathcal{D}$, we say that the utility vectors are *equally effective regarding* $\boldsymbol{p}$ if the regret ratio of $\boldsymbol{p}$ is smaller than ϵ w.r.t. each of them. To obtain these utility vectors, for each point $\boldsymbol{q} \in \mathcal{D} \setminus \{\boldsymbol{p}\}$, we build a constraint $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0$, denoted by $c_{\boldsymbol{p},\boldsymbol{q}}$ for simplicity. Let $\mathcal{S}_{\boldsymbol{p}}$ be the set of utility vectors that satisfy all these constraints. The following lemma shows that the regret ratio of $\boldsymbol{p}$ is smaller than ϵ w.r.t. any utility vector $\boldsymbol{u}' \in \mathcal{S}_{\boldsymbol{p}}$. Note that due to a lack of space, the proofs of some lemmas/theorems are shown in Appendix D.

Lemma 5.2. *Given a point $\boldsymbol{p} \in \mathcal{D}$, for any utility vectors $\boldsymbol{u}'$ in set $\mathcal{S}_{\boldsymbol{p}} = \{\boldsymbol{u} \mid \boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0, \forall \boldsymbol{q} \in \mathcal{D} \setminus \{\boldsymbol{p}\}\}$, the regret ratio of $\boldsymbol{p}$ w.r.t. $\boldsymbol{u}'$ is smaller than ϵ.*

Note that the constraints for set $\mathcal{S}_{\boldsymbol{p}}$ may contain *redundant* ones. Here, a constraint is said to be *redundant* if any utility vector $\boldsymbol{u}$ satisfying the other constraints will necessarily satisfy this one as well. We use $C_{\boldsymbol{p}}$ to denote the set of *non-redundant* constraints, which can be found via linear programming (LP). Due to the lack of space, the details of LP are shown in Appendix B. The time complexity of LP solvers, such as the interior-point method [20, 23], grows polynomially with the number of dimensions d. In the next section, we will show a strategy that reduces this computational cost linearly with the number of dimensions.

Since the equally effective utility vectors are not explored, we treat them as a whole. In this way, each set $\mathcal{S}_{\boldsymbol{p}}$ becomes a unit to be considered. The approximation path turns into a sequence of sets.

Now, suppose that the current candidate is set $\mathcal{S}_{\boldsymbol{p}}$ and the potential output is $\boldsymbol{p}$. The subsequent problem is which neighboring sets of $\mathcal{S}_{\boldsymbol{p}}$ should be explored. To address this, let us visit Lemma 5.3. Consider the corresponding set $C_{\boldsymbol{p}}$. If, for all constraints $c_{\boldsymbol{p},\boldsymbol{q}} \in C_{\boldsymbol{p}}$, $\boldsymbol{p}$ is more preferred by the user than $\boldsymbol{q}$, we have the following lemma.

Lemma 5.3. *If $\boldsymbol{p}$ is more preferred by the user than $\boldsymbol{q}$ for all constraints $c_{\boldsymbol{p},\boldsymbol{q}} \in C_{\boldsymbol{p}}$, the regret ratio of $\boldsymbol{p}$ is smaller than ϵ w.r.t. the user's utility vector $\boldsymbol{u}^*$, i.e., $regratio(\boldsymbol{p}, \boldsymbol{u}^*) < \epsilon$.*

Intuitively, Lemma 5.3 reveals a key insight: if a point $\boldsymbol{p}$ is preferred over points $\boldsymbol{q}$ for all constraints $c_{\boldsymbol{p},\boldsymbol{q}} \in C_{\boldsymbol{p}}$, we can safely return $\boldsymbol{p}$ as the output. This motivates us in exploring the neighboring sets $\mathcal{S}_{\boldsymbol{q}}$ of $\mathcal{S}_{\boldsymbol{p}}$ such that $c_{\boldsymbol{p},\boldsymbol{q}} \in C_{\boldsymbol{p}}$.

Operation Two. Suppose the current candidate set is $\mathcal{S}_{\boldsymbol{p}}$, and we are exploring a neighboring set $\mathcal{S}_{\boldsymbol{q}}$. We present the two points $\boldsymbol{p}$ and $\boldsymbol{q}$ to the user for interaction. If the user prefers $\boldsymbol{p}$ to $\boldsymbol{q}$, the candidate remains. Otherwise, the candidate is updated from $\mathcal{S}_{\boldsymbol{p}}$ to $\mathcal{S}_{\boldsymbol{q}}$. This ensures each update in the path leads to a strict improvement of candidate utility vectors, since the potential output (i.e., point $\boldsymbol{q}$) of the new candidates is more preferred by the user.

Algorithm. Our algorithm works as follows. The pseudocode is shown in Algorithm 1. Initially, we randomly select a utility vector $\boldsymbol{u} \in \mathcal{U}$ (line 1) and find the top-1 point $\boldsymbol{p}$ w.r.t. $\boldsymbol{u}$ (line 2). Based on $\boldsymbol{p}$, we build the sets $C_{\boldsymbol{p}}$ and $\mathcal{S}_{\boldsymbol{p}}$. Set $\mathcal{S}_{\boldsymbol{p}}$ is added to the approximation path as the candidate (line 4). Note that the set $\mathcal{S}_{\boldsymbol{p}}$ does not need to be exactly calculated. It is a label showing the progress of the path construction. The neighboring sets of $\mathcal{S}_{\boldsymbol{p}}$ are found based on set $C_{\boldsymbol{p}}$. For each constraint $c_{\boldsymbol{p},\boldsymbol{q}} \in C_{\boldsymbol{p}}$ (lines 5-6), we explore a neighboring set $\mathcal{S}_{\boldsymbol{q}}$. We use the points $\boldsymbol{p}$ and $\boldsymbol{q}$ to interact with the user. If the user prefers $\boldsymbol{p}$ to $\boldsymbol{q}$, the candidate utility vectors do not need to be updated. We directly move to the next constraint. Otherwise, we set $\boldsymbol{p} \leftarrow \boldsymbol{q}$ (line 8), and iterate the procedure: i) build the new sets $C_{\boldsymbol{p}}$ and $\mathcal{S}_{\boldsymbol{p}}$ and ii) involve set $\mathcal{S}_{\boldsymbol{p}}$ into the approximation path. The process stops when we encounter a constraint set $C_{\boldsymbol{p}'}$ such that $\boldsymbol{p}'$ is more preferred by the user than $\boldsymbol{q}'$ for all $c_{\boldsymbol{p}',\boldsymbol{q}'} \in C_{\boldsymbol{p}'}$. The correctness of our algorithm is guaranteed by Lemma 5.3.

Algorithm 1: Approximation Path Construction (Basic)

Input: Dataset $\mathcal{D}$, threshold ϵ
Output: A point $\boldsymbol{p}$

1 Randomly select a utility vector $\boldsymbol{u}$;
2 Find the top-1 point $\boldsymbol{p}$ w.r.t. $\boldsymbol{u}$ over $\mathcal{D}$;
3 **while** *point* $\boldsymbol{p}$ *is reset* **do**
4 Find C_p and add $\mathcal{S}_p$ to the approximation path;
5 **foreach** *constraint* $c_{\boldsymbol{p},\boldsymbol{q}} \in C_p$ **do**
6 Interact with the user with $\boldsymbol{p}$ and $\boldsymbol{q}$;
7 **if** *the user prefers* $\boldsymbol{q}$ *to* $\boldsymbol{p}$ **then**
8 $\boldsymbol{p} \leftarrow \boldsymbol{q}$;
9 **break**;
10 **return** point $\boldsymbol{p}$;

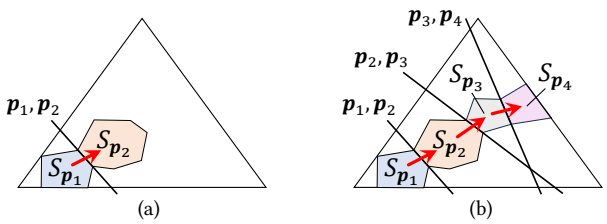

Figure 9: Approximation Path Construction

Example 5.4. Consider Figure 9 as an example. We set $\epsilon = 0$ for ease of illustration. Initially, we randomly select a utility vector $\boldsymbol{u}_1$ in the utility space and identify its corresponding top-1 point, e.g., $\boldsymbol{p}_1$. We construct the sets $C_{\boldsymbol{p}_1}$ and $\mathcal{S}_{\boldsymbol{p}_1}$ based on the points in $\mathcal{D}$. Suppose that the blue part in Figure 9(a) denotes $\mathcal{S}_{\boldsymbol{p}_1}$. It is involved as the start of the approximation path. From set $C_{\boldsymbol{p}_1}$, we randomly pick a constraint, e.g., $c_{\boldsymbol{p}_1,\boldsymbol{p}_2}$, and use points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ to interact with the user. Assume that the user prefers $\boldsymbol{p}_2$ to $\boldsymbol{p}_1$. We build sets $C_{\boldsymbol{p}_2}$ and $\mathcal{S}_{\boldsymbol{p}_2}$, and extend the path by adding $\mathcal{S}_{\boldsymbol{p}_2}$ (the red part in Figure 9(a)) to the approximation path. We continue this process by utilizing the constraints from $C_{\boldsymbol{p}_2}$, and so on. As shown in Figure 9(b), the path consists of a sequence of sets $\mathcal{S}_{\boldsymbol{p}}$. The procedure terminates when we reach the set $C_{\boldsymbol{p}_4}$ such that for any constraint $c_{\boldsymbol{p}_4,\boldsymbol{p}} \in C_{\boldsymbol{p}_4}$, point $\boldsymbol{p}_4$ is more preferred by the user than $\boldsymbol{p}$. The approximation path is: $\mathcal{S}_{\boldsymbol{p}_1} \rightarrow \mathcal{S}_{\boldsymbol{p}_2} \rightarrow \mathcal{S}_{\boldsymbol{p}_3} \rightarrow \mathcal{S}_{\boldsymbol{p}_4}$, and point $\boldsymbol{p}_4$ is returned as the output.

5.2 Enhancement

We develop several strategies to enhance the approximation path construction. These strategies can be broadly categorized into two groups. The first group focuses on acceleration, while the second group aims to reduce the number of interactive rounds.

5.2.1 Ray shooting - Group 1. The path construction process relies on obtaining the non-redundant constraint set $C_{\boldsymbol{p}}$ first, and then posing interactive questions to the user based on the constraints in $C_{\boldsymbol{p}}$. Although we can utilize the LP to obtain $C_{\boldsymbol{p}}$, the computation is costly since the LP involves multiple constraints in high dimensions. This limitation drives us to design a faster strategy.

Consider Example 5.4. Once the user indicates $\boldsymbol{p}_2$ is more preferred than $\boldsymbol{p}_1$, the algorithm directly transits from set $C_{\boldsymbol{p}_1}$ to $C_{\boldsymbol{p}_2}$ and the approximation path extends from $\mathcal{S}_{\boldsymbol{p}_1}$ to $\mathcal{S}_{\boldsymbol{p}_2}$. The remaining constraints in set $C_{\boldsymbol{p}_1}$ are never used. This observation indicates that computing the entire non-redundant constraint set upfront may be unnecessary. Motivated by this, our idea is to perform in a lazy manner: rather than obtaining the full set $C_{\boldsymbol{p}}$, as long as one non-redundant constraint is found (i.e., one neighboring set is obtained), the interaction can be driven forward.

To achieve this, we develop a ray shooting strategy that locates one non-redundant constraint each time. Suppose that we are considering the constraints $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0$, denoted by $c_{\boldsymbol{p},\boldsymbol{q}}$ for simplicity, where $\boldsymbol{q} \in \mathcal{D} \setminus \{\boldsymbol{p}\}$. Let $\boldsymbol{u}_{\boldsymbol{p}}$ be a utility vector inside $\mathcal{S}_{\boldsymbol{p}}$ and $\boldsymbol{v}$ be any unit vector. We can build a ray $\boldsymbol{u}(\lambda) = \boldsymbol{u}_{\boldsymbol{p}} + \lambda\boldsymbol{v}$, where λ is a positive real number. This ray starts at $\boldsymbol{u}_{\boldsymbol{p}}$, and extends along $\boldsymbol{v}$ as λ increases. For each constraint $c_{\boldsymbol{p},\boldsymbol{q}} : \boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0$, if $\boldsymbol{u}$ is substituted with $\boldsymbol{u}(\lambda)$, there may be an upper bound for λ.

$$(\boldsymbol{u}_{\boldsymbol{p}} + \lambda\boldsymbol{v}) \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0 \quad \Rightarrow$$
$$\lambda < -\frac{\boldsymbol{u}_{\boldsymbol{p}} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q})}{\boldsymbol{v} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q})}, \text{ if } \boldsymbol{v} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) < 0$$

Since there are $|\mathcal{D} \setminus \{\boldsymbol{p}\}| = n - 1$ constraints, there are at most $n - 1$ upper bounds for λ. Then, we have the following lemma.

Lemma 5.5. *The constraint that leads to the smallest upper bound for λ is non-redundant.*

Based on Lemma 5.5, we can efficiently identify a non-redundant constraint by generating a ray $\boldsymbol{u}(\lambda) = \boldsymbol{u}_{\boldsymbol{p}} + \lambda\boldsymbol{v}$ and identifying the constraint that yields the smallest upper bound for λ. We refer to this constraint as the one that *bounds* the ray. To generate the ray, the unit vector $\boldsymbol{v}$ is chosen uniformly at random, while the utility vector $\boldsymbol{u}_{\boldsymbol{p}}$ depends on the current stage of path construction. If the current $C_{\boldsymbol{p}}$ is the first one to be considered, we use the randomly selected utility vector (Algorithm 1 Line 1) as $\boldsymbol{u}_{\boldsymbol{p}}$. Subsequently, suppose that a non-redundant constraint is found via ray shooting. Let $\boldsymbol{u}'$ denote the intersection between the ray and this constraint. If the user's feedback causes a transition from $C_{\boldsymbol{p}}$ to a new $C_{\boldsymbol{p}'}$, we use this intersection $\boldsymbol{u}'$ as the new start $\boldsymbol{u}_{\boldsymbol{p}'}$ for the ray.

For each constraint, to check whether there is an upper bound for λ, it requires computing two inner products: $\boldsymbol{u}_{\boldsymbol{p}} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q})$ and $\boldsymbol{v} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q})$. Each takes $O(d)$ time since the vectors are d-dimensional. Thus, processing one constraint needs $O(d)$ time. Since there are $O(n)$ constraints, the total time complexity of the ray shooting strategy is $O(dn)$. This is a significant improvement over linear programming, whose runtime is polynomial in d.

The ray shooting strategy allows us to identify one non-redundant constraint at a time. To find multiple ones, we can generate multiple rays and perform ray shooting for each. This naturally leads to a question: how many rays are needed to ensure that all non-redundant constraints are discovered? Let M denote the number of rays. Lemma 5.6 provides a theoretical analysis if we miss a non-redundant constraint. Specifically, consider Figure 10. The set of all possible rays forms the surface of a d-dimensional unit sphere. Each non-redundant constraint corresponds to a region on this surface such that if a ray passes through that region, it will be bounded by the constraint. For instance, the red constraint corresponds to the blue region. Any ray that passes through the blue region is bounded by the red constraint. Let $V(T)$ denote the volume of such a region.

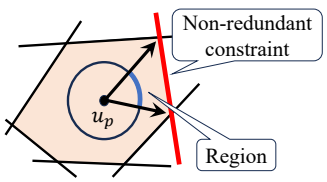

Figure 10: Ray Shooting

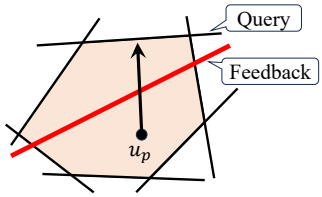

Figure 11: Constraint Pruning

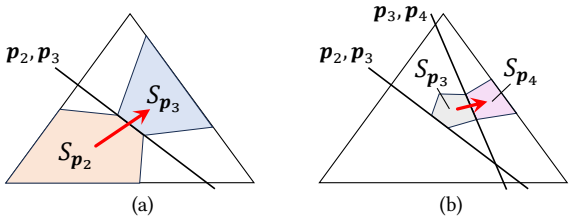

Figure 12: Path Shortening

If the rays are randomly sampled, the probability of missing this non-redundant constraint depends on $V(T)$.

Lemma 5.6. *Given M rays, the probability that the non-redundant constraint is missing is* $\mathbb{P} = \left(1 - \frac{V(T)\cdot\Gamma(d/2)}{2\pi^{d/2}}\right)^M$.

5.2.2 Constraint Pruning - Group 2. During the algorithm processing, when a non-redundant constraint $c_{p,q}$ is identified, we need to explore a neighboring set, and thus, a question that consists of p and q is asked to the user. In the worst case, each non-redundant constraint in the set C_p triggers a question, resulting in up to $|C_p|$ questions. This motivates us to think whether some questions can be pruned by leveraging user feedback obtained from the previous interactive rounds. Specifically, for each user feedback $p_i \succ p_j$, we build a constraint $u \cdot (p_i - p_j) > 0$. We refer to these constraints as *feedback constraints*. Instead, the non-redundant constraints in set C_p are called *query constraints*. During ray shooting, rather than considering the query constraints only, we involve both query and feedback constraints. If the smallest upper bound for λ is derived by a feedback constraint, no question needs to be asked. The following lemma verifies the correctness of this strategy. Consider a query constraint $c_{p,q} : u \cdot (p_i - (1-\epsilon)p_j) > 0$.

Lemma 5.7. *If any ray that would originally be bounded by the query constraint $c_{p,q}$ is now instead bounded by some feedback constraints, the question with points p and q can be safely skipped.*

Example 5.8. Consider Figure 11 as an example. The black lines denote query constraints, and the red line represents a feedback constraint derived from prior user feedback. As can be seen, originally, the ray is bounded by a query constraint, which would trigger a question. After including the feedback constraint, the ray is instead bounded by it, eliminating the need for asking a question.

5.2.3 Path Shortening - Group 2. During the path construction, we continually move from one set to its *neighboring* set. For instant, in Example 5.4, we start from set $\mathcal{S}_{p_1}$ and move to set $\mathcal{S}_{p_2}$, $\mathcal{S}_{p_3}$, $\mathcal{S}_{p_4}$, etc. sequentially. Each set is located next to its predecessor. If the sets in the path are small, each step can only drive a small pace towards the user's utility vector. Observing this, a problem arises: can we expand the pace to shorten the path length?

To expand the pace, our idea is to enlarge each set $\mathcal{S}_p$ by reducing the data points. We maintain a set Q that contains the user feedback to each interactive question, e.g., $Q = \{p_1 \succ p_2, p_1 \succ p_3, ...\}$. Initially, $Q = \emptyset$. We uniformly sample a few utility vectors u satisfying Q, i.e., if $p_i \succ p_j \in Q$, then $u \cdot (p_i - p_j) > 0$. We find the top-1 point w.r.t. each sampled utility vector and store all these points in a set $\mathcal{P}$. Then, we replace dataset $\mathcal{D}$ by $\mathcal{P}$ to conduct the approximation path construction. In this way, when building set C_p for each point $p \in \mathcal{P}$, we only need to build constraint $u \cdot (p - (1-\epsilon)q) > 0$ for all $q \in \mathcal{P} \setminus \{p\}$ (instead of $q \in \mathcal{D} \setminus \{p\}$). This will lead to more utility vectors satisfy C_p, and thus, $\mathcal{S}_p$ becomes larger. When the construction process terminates, i.e., stops at a set $\mathcal{S}_p$, since not all points in $\mathcal{D}$ are considered, point p may no longer be a valid output as before. Thus, we again randomly sample a few new utility vectors satisfying the set Q to repeat the process.

As progress continues, the set Q gradually collects more user feedback. Consequently, fewer utility vectors would satisfy Q, and thus, the sampled utility vectors become closer to each other. The closeness often causes many sampled utility vectors to share the same top-1 point over $\mathcal{D}$, resulting in a small number of points in the set $\mathcal{P}$. In the extreme case where $|\mathcal{P}| = 1$, we turn back to the full dataset $\mathcal{D}$ to construct the approximation path.

It is easy to see that our strategy follows a "large pace to small pace" manner. In early times, there is little user feedback in the set Q. There are a large number of utility vectors satisfying Q. Since the number of sampled utility vectors is limited, for some utility vectors u satisfying Q, we may fail to include in $\mathcal{P}$ the points q that have high utilities w.r.t. them. As a result, for these utility vectors, the point $p \in \mathcal{P}$ is likely to achieve the regret ratio requirement over $\mathcal{P}$ w.r.t. them since their high-utility points q are absent. This makes the set $\mathcal{S}_p$ large, and consequently, the pace expands. As the interaction progresses, the number of utility vectors u satisfying Q decreases. The sampled utility vectors are closer to each other. It becomes easier to find the points that have high utilities w.r.t. all utility vectors satisfying Q. Consequently, each point $p \in \mathcal{P}$ is less likely to meet the regret ratio requirement for many utility vectors unless it truly deserves to. This causes the set $\mathcal{S}_p$ to become smaller, and the pace becomes finer accordingly. Finally, since we are back to the full dataset $\mathcal{D}$, we can ensure the exact result.

Interestingly, there is a byproduct when we replace the set $\mathcal{D}$ with $\mathcal{P}$. When conducting the ray shooting, we only need to build constraint $u \cdot (p - (1-\epsilon)q) > 0$ for all $q \in \mathcal{P} \setminus \{p\}$ (instead of $q \in \mathcal{D} \setminus \{p\}$). The number of constraints is reduced, which enables the execution time to be further reduced.

Example 5.9. Consider Figure 12 to illustrate. Initially, since $Q = \emptyset$, we sample a few utility vectors in the utility space $\mathcal{U}$ and obtain the top-1 set $\mathcal{P}$. Suppose that a point $p_2 \in \mathcal{P}$ is considered. We find one constraint based on the ray shooting, e.g., $(p_2 - (1-\epsilon)p_3) \cdot u > 0$ and ask a question. If the user prefers p_3 to p_2, as shown in Figure 12(a), we move from the set $\mathcal{S}_{p_2}$ to $\mathcal{S}_{p_3}$. If we stop at the set $\mathcal{S}_{p_3}$ after asking a few questions, we then iterate the path construction process as shown in Figure 12(b). As can be noticed, the approximation path includes two sub-paths: one is $\mathcal{S}_{p_2} \rightarrow \mathcal{S}_{p_3}$ (in Figure 12(a)) and the other is $\mathcal{S}_{p_3} \rightarrow \mathcal{S}_{p_4}$ (in

Figure 12(b)). Compared to the approximation path in Example 5.4, the path length has been shortened.

Remark. Sections 5.2.2 and 5.2.3 reduce the interactive rounds, i.e., the number of questions, from two perspectively. The former reduces the number of questions asked when considering each set $\mathcal{S}_p$, while the latter reduces the number of sets $\mathcal{S}_p$ encountered.

5.2.4 Dimension Reduction - Group 1. In high-dimensional settings (e.g., hundreds of dimensions), it becomes computationally expensive to conduct any operations. To mitigate this, our idea is to reduce the number of dimensions d grounded in the *Johnson–Lindenstrauss (JL) lemma* [12, 14]. Specifically, let d' be the number of dimensions after reduction. We build a random matrix $\boldsymbol{A} \in \mathbb{R}^{d' \times d}$, where each element is drawn i.i.d. from $\mathcal{N}(0, 1/d')$. Then, each point $\boldsymbol{p} \in \mathcal{D}$ is projected as $\overline{\boldsymbol{p}} = \boldsymbol{A}\boldsymbol{p}$. In this way, although the user's utility vector $\boldsymbol{u}^*$ is unknown, we can also consider it is projected as $\overline{\boldsymbol{u}^*} = \boldsymbol{A}\boldsymbol{u}^*$. The JL lemma ensures that the projected points $\overline{\boldsymbol{p}}$ and utility vector $\overline{\boldsymbol{u}^*}$ preserve their relevant structure. For ease of illustration, we use $\bar{\cdot}$ to denote a point/vector after dimension reduction.

Lemma 5.10 (Johnson–Lindenstrauss [12, 14]). *For any $\varepsilon \in (0, 1)$, if $d' > O(\log n/\varepsilon^2)$, then with probability at least $\frac{n-2}{n}$, for all $\boldsymbol{p} \in \mathcal{D}$, we have $|\overline{\boldsymbol{p}} \cdot \overline{\boldsymbol{u}^*} - \boldsymbol{p} \cdot \boldsymbol{u}^*| \leq \frac{\varepsilon}{2}(\|\boldsymbol{p}\|^2 + \|\boldsymbol{u}^*\|^2)$.*

Let us consider the case where we still use the points $\boldsymbol{p}$ from the dataset $\mathcal{D}$ to interact with the user, but replace them by $\overline{\boldsymbol{p}}$ for the relevant calculation. Lemma 5.10 indicates that after the dimension reduction, the utility of each point $\boldsymbol{p} \in \mathcal{D}$ w.r.t. the user's utility vector $\boldsymbol{u}$ only varies slightly. Unfortunately, this small distortion still affects our algorithm, deteriorating the accuracy of the final returned point. Specifically, consider any two points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$.

Lemma 5.11. *For any $\varepsilon \in (0, 1)$, if $d' > O(\log n/\varepsilon^2)$, we have $|\overline{\boldsymbol{u}^*} \cdot (\overline{\boldsymbol{p}_i} - \overline{\boldsymbol{p}_j}) - \boldsymbol{u}^* \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j)| \leq \frac{\varepsilon}{2}(\|\boldsymbol{p}_i\|^2 + \|\boldsymbol{p}_j\|^2) + \varepsilon\|\boldsymbol{u}^*\|^2$ with probability at least $\frac{n-2}{n}$.*

Suppose that two points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$ are presented to the user as a question, and w.l.o.g that the user prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$. We can conclude that $\boldsymbol{u}^* \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) > 0$. Nevertheless, based on Lemma 5.11, after the dimension reduction, $\overline{\boldsymbol{u}^*} \cdot (\overline{\boldsymbol{p}_i} - \overline{\boldsymbol{p}_j})$ may be smaller than 0. If we interpret each user feedback $\boldsymbol{p}_i \succ \boldsymbol{p}_j$ in $\mathcal{Q}$ by $\overline{\boldsymbol{u}} \cdot (\overline{\boldsymbol{p}_i} - \overline{\boldsymbol{p}_j}) > 0$, the user's utility vector after dimension reduction $\overline{\boldsymbol{u}^*}$ may not satisfy $\mathcal{Q}$. This motivates us to adjust the constraints, ensuring that $\overline{\boldsymbol{u}^*}$ can always satisfy $\mathcal{Q}$. Let us revisit the inequality in Lemma 5.11. By switching some terms on both sides, we have

$$\begin{aligned} \boldsymbol{u}^* \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) &\leq \overline{\boldsymbol{u}^*} \cdot (\overline{\boldsymbol{p}_i} - \overline{\boldsymbol{p}_j}) + \frac{\varepsilon}{2}(\|\boldsymbol{p}_i\|^2 + \|\boldsymbol{p}_j\|^2) + \varepsilon\|\boldsymbol{u}^*\|^2 \\ &\leq \overline{\boldsymbol{u}^*} \cdot (\overline{\boldsymbol{p}_i} - \overline{\boldsymbol{p}_j}) + \frac{\varepsilon}{2}(\|\boldsymbol{p}_i\|^2 + \|\boldsymbol{p}_j\|^2) + \varepsilon \end{aligned}$$

The second line is derived based on $\sum_{i=1}^{d} u[i] = 1$. The maximum $\|\boldsymbol{u}^*\|^2$ happens when there exist a dimension j such that $u[j] = 1$ and $u[i \neq j] = 0$. Following this derivation, for any user feedback $\boldsymbol{p}_i \succ \boldsymbol{p}_j$ in $\mathcal{Q}$, we use $\overline{\boldsymbol{u}} \cdot (\overline{\boldsymbol{p}_i} - \overline{\boldsymbol{p}_j}) + \frac{\varepsilon}{2}(\|\boldsymbol{p}_i\|^2 + \|\boldsymbol{p}_j\|^2) + \varepsilon > 0$ instead of $\overline{\boldsymbol{u}} \cdot (\overline{\boldsymbol{p}_i} - \overline{\boldsymbol{p}_j}) > 0$ as the constraint to interpret it.

Lemma 5.12. *The user's utility vector after dimension reduction $\overline{\boldsymbol{u}^*}$ satisfies $\mathcal{Q}$, if for any user feedback $\boldsymbol{p}_i \succ \boldsymbol{p}_j$ in $\mathcal{Q}$, we use the constraint $\overline{\boldsymbol{u}^*}$ satisfies $\overline{\boldsymbol{u}} \cdot (\overline{\boldsymbol{p}_i} - \overline{\boldsymbol{p}_j}) + \frac{\varepsilon}{2}(\|\boldsymbol{p}_i\|^2 + \|\boldsymbol{p}_j\|^2) + \varepsilon > 0$*

Algorithm 2: Approximation Path Construction

Input: Dataset $\mathcal{D}$, threshold ϵ
Output: A point $\boldsymbol{p}$

```
1  Initialize matrix A and transform each p ∈ D into p̄ ∈ D̄;
2  Initialize user feedback set Q ← ∅, p̄* ← NULL;
3  while True do
4      Set P = ∅ and sample N utility vectors satisfying Q.
5      foreach sampled utility vector ū do
6          Find the top-1 point p̄ w.r.t. ū over D̄;
7          Add point p̄ to P;
8      if |P̄| > 1 then
9          if p̄* is NULL then p̄* ← any point in P ;
10         p̄*, Q = Path (P, Q, p̄*);
11     else
12         p*, _ = Path (D, Q, p*);
13         return point p*;
   Path (Dataset P, set Q, point p̄*)
14 Set C = ∅ and p̄ = p̄*;
15 while point p̄ is reset do
16     foreach q̄ ∈ D̄ \ {p̄} do
17         Add constraint ū · (p̄ − (1 − ε)q̄) > 0 into set C
18     foreach p ≻ q ∈ Q do
19         Add the corresponding constraint into set C
20     for M iterations do
21         Conduct a ray shooting to find a non-redundant
             query constraint c_{p̄,q̄} from C;
22         if p and q have not been asked then
23             Interact with the user with p and q;
24             if the user prefers p to q then
25                 Add p ≻ q into set Q;
26             else
27                 Add q ≻ p into set Q;
28                 p̄ ← q̄;
29                 break;
30 return p̄, Q;
```

This constraint adjustment guarantees that, even after dimensionality reduction, the user's utility vector still satisfies $\mathcal{Q}$. As a result, when sampling utility vectors to build the set $\mathcal{P}$, the user's utility vector is inherently considered. Note that when $|\mathcal{P}| = 1$, we not only fall back to the original dataset $\mathcal{D}$, but also revert to the original dimensional space to continue path construction.

5.3 Summary and Analysis

We summarize our algorithm for approximation path construction with the proposed strategies. The main idea is to traverse a sequence of utility vector sets, gradually approximating the user's utility vector. The pseudocode is shown in Algorithm 2.

We first apply the dimension reduction strategy to transform the points $\boldsymbol{p} \in \mathcal{D}$ into d'-dimensional points $\overline{\boldsymbol{p}}$ (lines 1-2). Denote the

transformed dataset by $\overline{\mathcal{D}}$. We initialize the feedback set $\mathcal{Q} \leftarrow \emptyset$ and the potential output $\overline{\boldsymbol{p}}* \leftarrow$ NULL. The algorithm operates in iterations. In each iteration, we follow the path-shortening strategy. We randomly sample a set of N utility vectors satisfying $\mathcal{Q}$ (line 4). For each sampled utility vector, we find its corresponding top-1 point in $\overline{\mathcal{D}}$ and add it to a set $\mathcal{P}$ (lines 5-7).

Consider the case where $|\mathcal{P}| > 1$ (line 8). We apply the algorithm **Path** to conduct the path construction based on $\mathcal{P}$ (lines 8–10). For the point $\overline{\boldsymbol{p}}*$ used in **Path**, if it is NULL, we set $\overline{\boldsymbol{p}}*$ to any point in $\mathcal{P}$ (line 9). The algorithm **Path** works as follows. Let $\overline{\boldsymbol{p}}$ be the potential output initialized to be $\overline{\boldsymbol{p}}*$ (line 14). We focus on the set $\mathcal{S}_{\boldsymbol{p}}$ and attempt to explore its neighboring sets. Following the constraint pruning strategy, we build a set C to contain all query constraints of $\overline{\boldsymbol{p}}$ (lines 16-17) and all feedback constraints (lines 18-19). Then, we find non-redundant query constraints in C by the ray shooting strategy (line 21). Once a non-redundant constraint $c_{\overline{\boldsymbol{p}},\overline{\boldsymbol{q}}}$ is found, the neighboring set $\mathcal{S}_{\boldsymbol{q}}$ is to be explored and a question is asked to the user. Note that the question utilizes the corresponding points $\boldsymbol{p}$ and $\boldsymbol{q}$ that are in the original dimension (line 22). Based on the user feedback, we add either $\boldsymbol{p} \succ \boldsymbol{q}$ or $\boldsymbol{q} \succ \boldsymbol{p}$ into set $\mathcal{Q}$ (lines 25 and 27). If the user prefers $\boldsymbol{q}$ to $\boldsymbol{p}$, we turn to use point $\overline{\boldsymbol{q}}$ as the potential output (line 28). We focus on the set $\mathcal{S}_{\boldsymbol{q}}$ and explore its neighboring sets. Otherwise, we conduct another ray shooting to find the next non-redundant query constraint of $\overline{\boldsymbol{p}}$. After M times ray shooting, if the potential output $\overline{\boldsymbol{p}}$ does not change (line 20), we return $\overline{\boldsymbol{p}}$ and $\mathcal{Q}$ (line 30).

Consider the case where $|\mathcal{P}| = 1$ (line 11). We turn back to the original space. The algorithm **Path** is called based on the dataset $\mathcal{D}$, and $\boldsymbol{p}*$ represents the point $\overline{\boldsymbol{p}}*$ in the original space (line 12). The point returned by **Path** is the final output (line 13). Note that when **Path** works on the dataset $\mathcal{D}$, all the points used are in the original space. In the pseudocode of **Path**, we use $\overline{\boldsymbol{p}}$ and $\overline{\boldsymbol{q}}$ to highlight the case where we use the set $\mathcal{P}$. When the algorithm works on the dataset $\mathcal{D}$, $\overline{\boldsymbol{p}}$ and $\overline{\boldsymbol{q}}$ should be $\boldsymbol{p}$ and $\boldsymbol{q}$ that are in the original space.

For the approximation path construction, the time complexity is dominated by two steps. Denote by d' the number of dimensions after dimension reduction. Let us analyze these two steps separately. The first is the construction of the point set $\overline{\mathcal{P}}$. Since there are N sampled utility vectors and each requires $O(d'n)$ time to identify its corresponding top-1 point, this step needs $O(Nd'n)$ time in total. The second step is to proceed with the ray shooting. The time complexity is $O(nd)$ as stated before. Compared to the existing ones [34], where the time complexity of the convex polytope is exponential to d and polynomial in n, our algorithm significantly reduces computational cost. The number of interactive rounds needed by the path construction algorithms is analyzed in the following.

Theorem 5.13. *If the dataset $\mathcal{D}$ of size n is uniformly distributed with the same norm, the expected number of questions asked is bounded by $O(d^2 M^{1/d} \log_M n)$.*

6 EXPERIMENTS

We conducted experiments on a Mac with a M3 chip. All programs were implemented in Python.

Datasets. We adopted both synthetic and real datasets commonly used in existing studies [10, 13, 21, 34]. For the synthetic datasets, we used *anti-correlated* and *independent* datasets produced by the generator designed for *skyline* operators [2, 21]. For the real datasets, we used datasets *Player* [7], *Game* [4], and *Music* [5]. Table 2 summarizes the relevant information about these datasets.

Table 2: Real Dataset

Dataset	$n = \lvert\mathcal{D}\rvert$	d
Player [7]	17,386	20
Game [4]	103,687	108
Music [6]	136,736	300

Baselines. We compared our algorithm APC against all existing ones designed for the interactive regret query. (1) UH-Random [34], the state-of-the-art (SOTA) algorithm. It is a random-based algorithm that randomly selects points from a candidate set as the question in each interactive round. (2) UH-Simplex [34], a greedy algorithm that selects a pair of extreme points from the convex hull of the candidate set as the question in each interactive round. (3) SinglePass [38], a heuristic algorithm that selects pairs of points based on a predefined random sequence and rule-based filters. (4) ActiveRanking [19], a machine learning algorithms for learning-to-rank. We adapted it by returning the top-ranked point after obtaining the ranking. (5) PrefLearning [24], a machine learning algorithm for learning the user's preference. We adapted it by returning the point with the highest utilities after obtaining the preference.

Parameter setting. We varied the following parameters: (1) the threshold of regret ratio ϵ; (2) the dataset size n; and (3) the number of dimensions d. Following the typical settings [34, 38], we use the anti-correlated dataset as the default dataset. The default values of parameters on synthetic datasets are $\epsilon = 0.01$ and $n = 100,000$, and $d \in \{4, 30\}$ unless otherwise specified. Here, we use two default values for d because some existing algorithms are not capable of handling the setting $d = 30$. To facilitate comparison, we also include the low-dimensional setting $d = 4$.

Measurement. We ran each algorithm 10 times. We evaluated them using three measurements and reported the average. (1) *The number of questions asked.* The number of interactive rounds needed to find a desired point. (2) *Execution time.* The total time taken by the entire interaction process. (3) *Regret ratio.* The actual regret ratio of the returned point. Due to space limitations, we omit the detailed results of the regret ratio. We report that all algorithms successfully returned tuples satisfying the regret ratio requirement.

6.1 Performance on Synthetic Datasets

Algorithm parameter. Our algorithm involves two key parameters. The first one, N, determines the number of sampled utility vectors when building the set $\mathcal{P}$ (line 4 in Algorithm 2). The second one, M, controls the number of times the ray shooting is performed to find the non-redundant constraints (line 20 in Algorithm 2). We analyze the setting of these two parameters. Due to the space limitation, the results can be found in Appendix C. We set $N = d^2$ and $M = 100d$ in the following experiments.

Interaction process. Figures 13 and 14 show the progress in the interaction process on the 4-dimensional (4D) and 30-dimensional (30D) synthetic datasets, respectively. We report two metrics at the end of each interactive round: (1) the regret ratio of the point $\boldsymbol{p}$ to be returned if the interaction were terminated at that round, and (2) the accumulated execution time. For the first metric, we followed

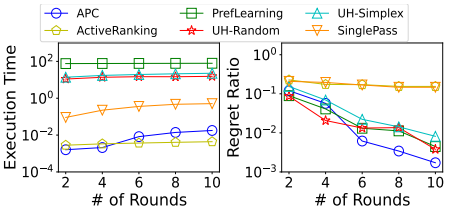

Figure 13: Interaction process in 4D

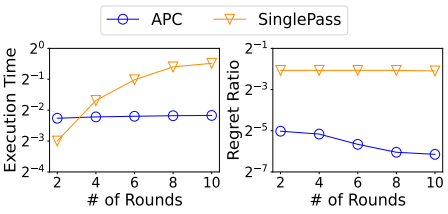

Figure 14: Interaction process in 30D

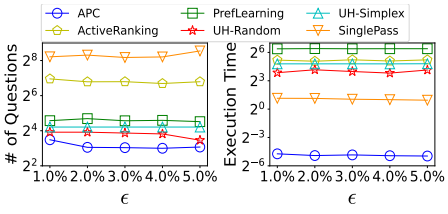

Figure 15: vary ϵ in 4D dataset

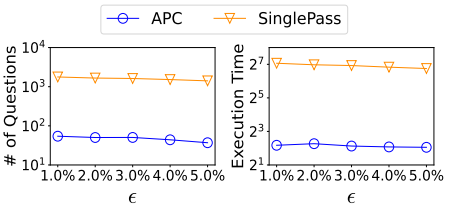

Figure 16: Vary ϵ in 30D dataset

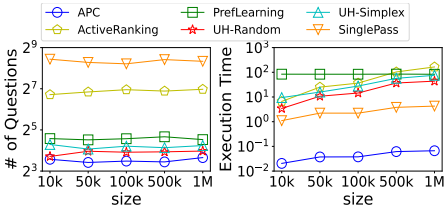

Figure 17: Vary size in 4D dataset

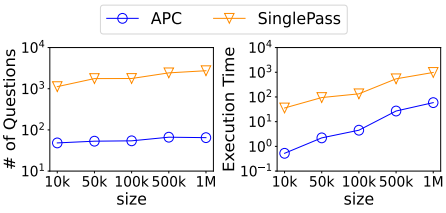

Figure 18: Vary size in 30D dataset

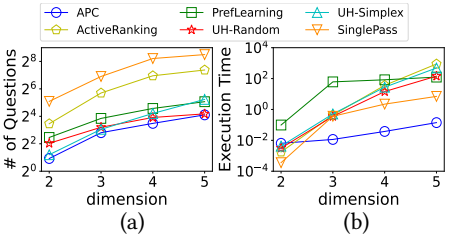

Figure 19: Vary d in low dimensions

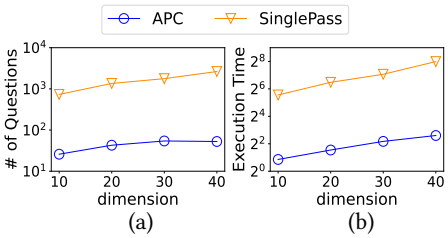

Figure 20: Vary d in high dimensions

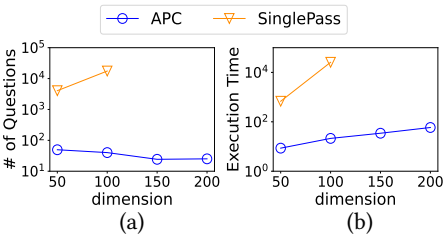

Figure 21: Vary d in high dim (Indep)

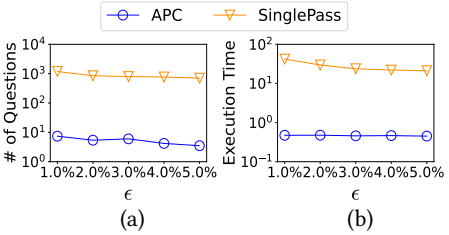

Figure 22: Vary ϵ in Players (20D)

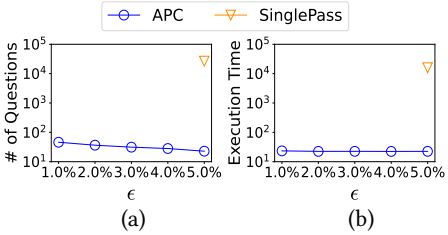

Figure 23: Vary ϵ in Game (108D)

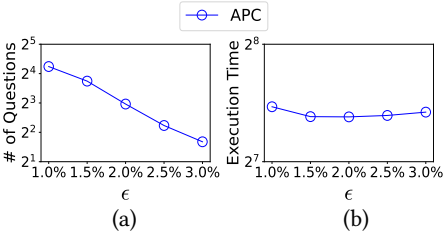

Figure 24: Vary ϵ in Music (300D)

[24, 34, 38] to find the returned point $\boldsymbol{p}$ as follows. For UH-Simplex and UH-Random, they maintain a range $\mathcal{R}$ to contain all possible candidates of the user's utility vector. We find the utility vector $\boldsymbol{u}$, which is in the middle of the range $\mathcal{R}$, and use the point that has the highest utility w.r.t. $\boldsymbol{u}$ as $\boldsymbol{p}$. For SinglePass and ActiveRanking, they maintain a point set to contain all possible candidates of the user's favorite point. We use the one that ranks first as $\boldsymbol{p}$. For PrefLearning, we use the point that has the highest utility w.r.t. its currently estimated utility vector as $\boldsymbol{p}$. For our algorithm APC, since it maintains a potential output $\overline{\boldsymbol{p}^*}$ during the process (line 2 in Algorithm 2), we directly use it as $\boldsymbol{p}$.

Figure 13 shows the results on the 4D dataset. As expected, all algorithms exhibited reduced regret ratios and increased execution times when the number of interactive rounds increased. Our algorithm APC achieved the shortest execution time. After 10 interactive rounds, the time was below 0.02 seconds. It was also effective in reducing the regret ratio. After 6 interactive rounds, its regret ratio dropped below 0.01, which is the lowest among all algorithms.

Figure 14 presents the results on the 30D dataset. There was only one baseline SinglePass included since the other two existing algorithms failed to complete within a reasonable time (exceeding 10^4 seconds). Our algorithm APC again achieved the best overall performance. For example, it took 0.22 seconds to complete 10 interactive rounds with a regret ratio below 0.015. This was 69% faster than SinglePass, whose regret ratio was more than 39% higher. Moreover, as the number of interactive rounds increases, SinglePass exhibited a steep rise in execution time without an obvious improvement in regret ratio. In contrast, APC increased execution time marginally

while significantly reducing the regret ratio. These results verified that our algorithms progressed well during the interaction process.

Varying threshold ϵ. We studied the impact of the threshold ϵ on all algorithms by varying it from 0.01 to 0.05.

Figure 15 shows the performance on a 4-dimensional synthetic dataset. Figures 15(a) and 15(b) report the number of interactive rounds and the execution time, respectively. As shown there, our algorithms APC consistently outperformed existing algorithms in all cases. For instance, when $\epsilon = 0.03$, APC reduced the interactive rounds by 43% compared to existing algorithms and was two orders of magnitude faster than them. Besides, all algorithms successfully returned the points within the regret ratio below the threshold.

Figure 16 shows the performance on a 30-dimensional synthetic dataset. We only included the existing algorithm SinglePass since the others could not run in high dimensions. Our algorithm APC showed clear advantages in the number of interactive rounds and execution time (Figures 16(a) and (b)). For example, when $\epsilon = 0.05$, APC only required 37.0 interactive rounds within 4.1 seconds, while algorithm SinglePass needed 1409.2 interactive rounds and 107.6 seconds. All algorithms successfully returned the points within the regret ratio below the threshold.

Varying dataset size n. In Figures 17 and 18, we evaluated the scalability of all algorithms w.r.t. the dataset size n, by varying n from 10K to 1M on the 4-dimensional and 30-dimensional datasets, respectively. Across all dataset sizes, our algorithm APC consistently required the fewest interactive rounds. For instance, on the 4-dimensional dataset, when n = 1M, APC required 20% fewer interactive rounds compared to the best existing algorithm UH-Random. On the 30D dataset, the advantage was even more significant: when n =1M, APC achieved a dramatic 42x reduction in terms of the number of interactive rounds. For the execution times, all algorithms became slower when n increased, as expected. For instance, on the 30-dimensional dataset, SinglePass took 35.7 seconds for n = 10K and ballooned to 981.9 seconds for n = 1M. This is because, with a larger dataset size, an algorithm needs to handle more points to derive questions for interaction, leading to longer execution times. Nevertheless, APC remained highly efficient. On the 30-dimensional dataset, the execution time of APC was only 59.0 seconds when n =1M. Regarding the regret ratios, all algorithms met the requirement, indicating their correctness.

Varying dimensionality d. In Figures 19 and 20, we evaluated the scalability of algorithms w.r.t. the dimensionality on the anti-correlated datasets. We varied d from 2 to 5 on the low-dimensional datasets and from 10 to 40 on the high-dimensional datasets, respectively. As expected, all algorithms exhibited increased interactive rounds and execution times as d increased, reflecting the greater complexity of learning the user's utility vector in high-dimensional settings. Nonetheless, our algorithm APC consistently outperformed the baselines in all cases. For example, on the 10-dimensional dataset, APC took 25.9 interactive rounds in 1.8 seconds. In comparison, SinglePass took 734.1 interactive rounds in 47.1 seconds. The performance gap widened in higher dimensions. When d reached 40, SinglePass ballooned to over 250 seconds with more than 2,600 rounds, whereas APC completed within only 6.1 seconds with 52.7 rounds. Note that we only increased the number of dimensions to 40. This is because the generator [2] could not produce higher-dimensional anti-correlated datasets (e.g., $d = 50$). To further verify the scalability of our algorithm, we adopted the widely used independent datasets that were also produced by this generator. In Figure 21, we extended d from 50 up to 200. SinglePass was only able to run up to $d = 100$ (taking over 34,500 seconds), beyond which it became infeasible. In contrast, APC remained highly efficient. It completed in just 21.3 seconds at $d = 100$, which is four orders of magnitude faster than SinglePass.

6.2 Performance on Real Datasets

We evaluated the performance of our algorithms against existing ones by varying the threshold ϵ. Figures 22, 23, and 24 show the results on datasets *Player*, *Game*, and *Music*, respectively. These datasets represent increasing levels of dimensionality, with *Player* including 20 dimensions, *Game* containing over 100, and *Music* reaching up to 300. The results show that our algorithm APC performed well in terms of both the number of interactive rounds and execution time on real datasets.

On the *Player* dataset, APC required only 7.4 interactive rounds when $\epsilon = 0.01$, compared to 1183.6 interactive rounds needed by SinglePass, which is a 99.3% reduction. Moreover, APC completed in under 0.5 seconds, while SinglePass required over 40 seconds, which is a 98.7% reduction. On the more challenging *Game* and *Music* datasets, the performance gap widened significantly. SinglePass failed to complete within a reasonable time. It largely exceeded 10^4 seconds in all cases (except on the *Game* dataset when $\epsilon = 0.05$). In comparison, APC remained a small number of interactive rounds within a few seconds. These findings verified the effectiveness of our algorithms in real-life scenarios.

6.3 Summary

The experiments demonstrated the superiority of our algorithm APC over the best-known existing ones. (1) *Efficiency in Interaction and Runtime.* APC required fewer interactive rounds and less time than existing algorithms. For example, on the 30-dimensional dataset with $\epsilon = 0.05$, while existing algorithms required at least 1409.2 interactive rounds, APC needed only 37.0 rounds. (2) *Strong Scalability.* Our algorithms scaled well on the dataset size and the dimensionality. For instance, on the independent dataset with $d = 100$, APC needed 21.3 seconds, while existing ones required over 10^4 seconds. (3) *Real-World Usage.* Our algorithms showed great promise in real-world applications. For example, on dataset *Player* with $\epsilon = 0.01$, APC achieved a 99.3% reduction in the number of interactive rounds compared to existing algorithms.

7 CONCLUSION

In this paper, we consider the interactive regret query in high-dimensional space. Unlike the existing algorithms that approximate the user's utility vector from a global perspective, we propose an algorithm to construct a path that approximates the user's utility vector from a local perspective. Experiments show that our algorithm outperforms existing ones, not only in low-dimensional settings, but also in high-dimensional settings, regarding the number of interactive rounds and the execution time. For future work, we consider that each interactive question consists of multiple tuples.

REFERENCES

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

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

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

[4] The Game Dataset. 2025. https://www.kaggle.com/datasets/fronkongames/steam-games-dataset

[5] The Movie Dataset. 2025. https://www.cse.cuhk.edu.hk/systems/hash/gqr/datasets.html

[6] The Music Dataset. 2025. https://www.cse.cuhk.edu.hk/systems/hash/gqr/datasets.html

[7] The Player Dataset. 2025. https://www.kaggle.com/datasets/vivovinco/19912021-nba-stats?select=players.csv

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

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

[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*. Curran Associates Inc., Red Hook, NY, USA, 2240–2248.

[12] Daniel M. Kane and Jelani Nelson. 2014. Sparser Johnson-Lindenstrauss Transforms. *J. ACM* 61, 1, Article 4 (2014), 23 pages.

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

[14] Kasper Green Larsen and Jelani Nelson. 2017. Optimality of the Johnson-Lindenstrauss Lemma. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*. 633–638.

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

[16] Andrzej Maćkiewicz and Waldemar Ratajczak. 1993. Principal components analysis (PCA). *Computers & Geosciences* 19, 3 (1993), 303–342.

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

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

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

[20] Arkadi S Nemirovski and Michael J Todd. 2008. Interior-point methods for optimization. *Acta Numerica* 17 (2008), 191–234.

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

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

[23] Florian A Potra and Stephen J Wright. 2000. Interior-point methods. *Journal of computational and applied mathematics* 124, 1-2 (2000), 281–302.

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

[25] Weicheng Wang, Victor Junqiu Wei, Min Xie, Di Jiang, Lixin Fan, and Haijun Yang. 2025. Interactive Search with Reinforcement Learning. In *In IEEE ICDE International Conference on Data Engineering*.

[26] Weicheng Wang and Raymond Chi-Wing Wong. 2022. Interactive mining with ordered and unordered attributes. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2504–2516.

[27] Weicheng Wang, Raymond Chi-Wing Wong, H. V. Jagadish, and Min Xie. 2024. Reverse Regret Query. In *In IEEE ICDE International Conference on Data Engineering*.

[28] Weicheng Wang, Raymond Chi-Wing Wong, Jinyang Li, and H.V. Jagadish. 2025. Interactive Learning for Diverse Top-k Set. In *In IEEE ICDE International Conference on Data Engineering*.

[29] Weicheng Wang, Raymond Chi-Wing Wong, and Min Xie. 2021. Interactive Search for One of the Top-k. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 13.

[30] Weicheng Wang, Raymond Chi-Wing Wong, and Min Xie. 2023. Interactive Search with Mixed Attributes. In *In IEEE ICDE International Conference on Data Engineering*.

[31] Min Xie. 2022. Accelerated Algorithms for α-Happiness Query. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*. Springer, 53–68.

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

[33] Min Xie and Yang Liu. 2022. Favorite+: Favorite Tuples Extraction via Regret Minimization. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 5049–5053.

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

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

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

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

[38] Guangyi Zhang, Nikolaj Tatti, and Aristides Gionis. 2023. Finding Favourite Tuples on Data Streams with Provably Few Comparisons. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3229–3238.

Table 3: Frequently Used Notations

Notation	Definition
$\mathcal{D}$ and $\boldsymbol{p}$	The dataset and its contained point.
n and d	The dataset size and the number of dimensions.
$f_{\boldsymbol{u}}$ and $\boldsymbol{u}$	The utility function and the utility vector.
$regratio(\boldsymbol{p}, \boldsymbol{u})$	The regret ratio of point $\boldsymbol{p}$ w.r.t. $\boldsymbol{u}$
ϵ	The regret ratio threshold.
$C_{\boldsymbol{p}}$	The non-redundant constraints set of $\boldsymbol{p}$.
$\mathcal{S}_{\boldsymbol{p}}$	The set of utility vectors satisfying $C_{\boldsymbol{p}}$.
Q	The set of collected user feedback.

A FREQUENTLY USED NOTATIONS

Table 3 summarizes the frequently used notations in this paper.

B NON-REDUNDANT CONSTRAINT

Consider a point $\boldsymbol{p} \in \mathcal{D}$. For each point $\boldsymbol{q} \in \mathcal{D} \setminus \{\boldsymbol{p}\}$, we build a constraint $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0$ and utilize linear programming (LP) to check if it is redundant. Specifically, we define a variable w to be the objective value. For each point $\boldsymbol{q}' \in \mathcal{D} \setminus \{\boldsymbol{p}, \boldsymbol{q}\}$, we build a constraint $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}') > w$. For point $\boldsymbol{q}$, we build a constraint $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) < -w$. Formally, we construct an LP as follows.

$$\begin{aligned} &\text{maximize } w \\ &\text{subject to } \boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}') > w, \text{ where } \boldsymbol{q}' \in \mathcal{D} \setminus \{\boldsymbol{p}, \boldsymbol{q}\} \\ &\qquad \boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) < -w \end{aligned}$$

If the optimal value $w^* > 0$, it implies there exists a utility vector $\boldsymbol{u}$ that satisfies all constraints $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}') > 0$, where $\boldsymbol{q}' \in \mathcal{D} \setminus \{\boldsymbol{p}, \boldsymbol{q}\}$, while violating $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0$. Thus, the constraint $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0$ is non-redundant. Conversely, if no such $w > 0$ exists (i.e., the LP is infeasible), the constraint is redundant.

C EXPERIMENTS

We analyze the setting of the two parameters N and M as follows.

Parameter N. For parameter N, a small one may lead to a small point set $\mathcal{P}$, requiring the algorithm to call **Path** multiple times. Conversely, a large N would increase the computational cost for building $\mathcal{P}$. Since the number of sampled utility vectors must satisfy Q, and the number of qualified utility vectors typically grows with the number of dimensions d, we varied N by setting it to be $N'd^2$, where $N' = \{1, 2, 3, 4, 5\}$, and reported the performance of our algorithm on the 4D synthetic dataset. The results are shown in Figure 25. We observe that different values of N yield similar performance, indicating sufficient exploration with moderate sampling. Therefore, we fix $N = d^2$ in the other experiments.

Parameter M. We evaluated our algorithm APC by varying M on the 4D synthetic dataset. For parameter M, a small one may lead to the loss of many non-redundant constraints, potentially degrading result quality. Meanwhile, as discussed in [8], the number of non-redundant constraints is associated with the number of dimensions. In this case, we set M to be $M'd$, where $M' \in \{100, 200, ..., 500\}$. The results are shown in Figure 26. As can be observed, a large M leads to many ray shooting operations, resulting in a high cost. However, the actual regret ratios remain nearly unchanged. This means M is large enough to capture the non-redundant constraints. Hence, we fix $M = 100d$ in all subsequent experiments.

D PROOF

Proof of Lemma 5.3. Consider any constraint $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0$ in set $C_{\boldsymbol{p}}$. If point $\boldsymbol{p}$ is more preferred by the user than point $\boldsymbol{q}$, then $\boldsymbol{u}^* \cdot \boldsymbol{p} > \boldsymbol{u}^* \cdot \boldsymbol{q}$, where $\boldsymbol{u}^*$ is the user's utility vector. The user's utility vector $\boldsymbol{u}^*$ satisfies the constraint. Since this is achieved for all constraints in $C_{\boldsymbol{p}}$, the user's utility vector $\boldsymbol{u}^*$ must be in set $\mathcal{S}_{\boldsymbol{p}}$. By definition, the regret ratio of point $\boldsymbol{p}$ is smaller than ϵ for any utility vector $\boldsymbol{u} \in \mathcal{S}_{\boldsymbol{p}}$. Therefore, the regret ratio of point $\boldsymbol{p}$ is smaller than ϵ w.r.t. the user's utility vector $\boldsymbol{u}^*$. □

Proof of Lemma 5.2. Consider any constraint $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0$, where $\boldsymbol{q} \in \mathcal{D} \setminus \{\boldsymbol{p}\}$. We have the following derivation.

$$\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}) > 0 \Rightarrow \frac{\boldsymbol{u} \cdot (\boldsymbol{q} - \boldsymbol{p})}{\boldsymbol{u} \cdot \boldsymbol{q}} < \epsilon$$

Since this is enforced for every point $\boldsymbol{q} \in \mathcal{D} \setminus \{\boldsymbol{p}\}$, for any utility vector $\boldsymbol{u}'$ in set $\mathcal{S}_{\boldsymbol{p}}$, we guarantee $\frac{\boldsymbol{u}' \cdot (\boldsymbol{q}' - \boldsymbol{p})}{\boldsymbol{u}' \cdot \boldsymbol{q}'} < \epsilon$, where $\boldsymbol{q}'$ is the top-1 point over $\mathcal{D}$ w.r.t. $\boldsymbol{u}'$. Thus, the regret ratio of $\boldsymbol{p}$ is smaller than ϵ w.r.t. any utility vectors in set $\mathcal{S}_{\boldsymbol{p}}$. □

Proof of Lemma 5.5. Let $c_{\boldsymbol{p},\boldsymbol{q}^*} : \boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}^*) > 0$ be the constraint that leads to the smallest upper bound for λ. Then, let $t = -\frac{\boldsymbol{u}_{\boldsymbol{p}} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}^*)}{\boldsymbol{v} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}^*)} + \mu$, where $\mu > 0$ is a sufficiently small real number. Consider the utility vector $\boldsymbol{u}(t) = \boldsymbol{u}_{\boldsymbol{p}} + t\boldsymbol{v}$. This utility vector $\boldsymbol{u}_t$ must satisfy all constraints in set $C_{\boldsymbol{p}}$ except the one $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}^*) > 0$. By definition, the constraint $\boldsymbol{u} \cdot (\boldsymbol{p} - (1-\epsilon)\boldsymbol{q}^*) > 0$ is non-redundant. □

Proof of Lemma 5.6. Since each ray is randomly sampled, the probability that a ray passes through the region is: $\mathbb{P}_1 = \frac{V(T)}{V(\mathcal{S})}$, where $V(S) = \frac{2\pi^{d/2}}{\Gamma(d/2)}$ is the volume of the surface. Therefore, the probability that it does not pass through the region is $\mathbb{P}_2 = 1 - \frac{V(T)}{V(\mathcal{S})}$. Since the M rays are drawn independently, the probability that all the rays do not pass through the region is $\mathbb{P} = \left(1 - \frac{V(T)}{V(\mathcal{S})}\right)^M = \left(1 - \frac{V(T) \cdot \Gamma(d/2)}{2\pi^{d/2}}\right)^M$. □

Proof of Lemma 5.7. Denote by $C'_{\boldsymbol{p}}$ the combined constraint set that contains all the feedback constraints and the query constraints in $C_{\boldsymbol{p}}$, and by $S'_{\boldsymbol{p}}$ the set of utility vectors satisfying the constraints in $C'_{\boldsymbol{p}}$. If any ray that would originally be bounded by the query constraint $c_{\boldsymbol{p},\boldsymbol{q}}$ is now instead bounded by some feedback constraints, the query constraint $c_{\boldsymbol{p},\boldsymbol{q}}$ does not affect the set $S'_{\boldsymbol{p}}$, i.e., $c_{\boldsymbol{p},\boldsymbol{q}}$ is redundant in set $C'_{\boldsymbol{p}}$. If the user prefers $\boldsymbol{p}$ to $\boldsymbol{q}'$ for any constraint $c_{\boldsymbol{p},\boldsymbol{q}'}$ in $C_{\boldsymbol{p}} \setminus \{c_{\boldsymbol{p},\boldsymbol{q}}\}$, the user's utility vector must be in $S'_{\boldsymbol{p}}$. Therefore, skipping the questions with $\boldsymbol{p}$ and $\boldsymbol{q}$ does not affect the correctness of determining whether point $\boldsymbol{p}$ can be returned. □

Proof of Lemma 5.11. Based on Lemma 5.10, for any two points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$, we have $|\overline{\boldsymbol{p}_i} \cdot \overline{\boldsymbol{u}^*} - \boldsymbol{p}_i \cdot \boldsymbol{u}^*| \leq \frac{\varepsilon}{2}(\|\boldsymbol{p}_i\|^2 + \|\boldsymbol{u}^*\|^2)$ and $|\overline{\boldsymbol{p}_j} \cdot \overline{\boldsymbol{u}^*} - \boldsymbol{p}_j \cdot \boldsymbol{u}^*| \leq \frac{\varepsilon}{2}(\|\boldsymbol{p}_j\|^2 + \|\boldsymbol{u}^*\|^2)$, respectively. Combining these two inequities, we have $|\overline{\boldsymbol{u}^*} \cdot (\overline{\boldsymbol{p}_i} - \overline{\boldsymbol{p}_j}) - \boldsymbol{u}^* \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j)| \leq \frac{\varepsilon}{2}(\|\boldsymbol{p}_i\|^2 + \|\boldsymbol{p}_j\|^2) + \varepsilon\|\boldsymbol{u}^*\|^2$. □

Proof of Lemma 5.12. It follows that $\overline{\boldsymbol{u}^*} \cdot (\overline{\boldsymbol{p}_i} - \overline{\boldsymbol{p}_j}) + \frac{\varepsilon}{2}(\|\boldsymbol{p}_i\|^2 + \|\boldsymbol{p}_j\|^2) + \varepsilon > 0$ □

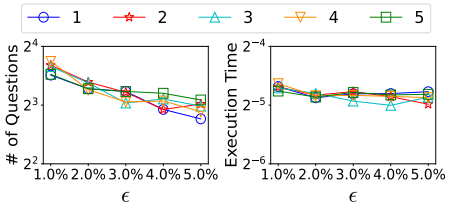

Figure 25: Vary N in 4D dataset

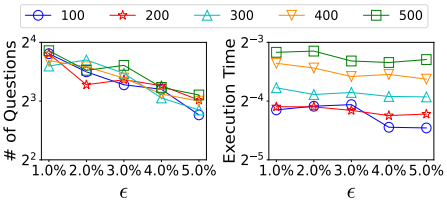

Figure 26: Vary M in 4D dataset

Proof of Theorem 5.13. Suppose that the points in the dataset $\mathcal{D}$ are uniformly distributed and have the same norm. If we uniformly sample M utility vectors from $\mathcal{Q}$ to build $\mathcal{P}$, since $M << n$, we have $|\mathcal{P}| = M$. These points can also be regarded as uniformly distributed. We can roughly assume the set $\mathcal{S}_p$ of each point $p \in \mathcal{P}$ is a cube and bounded by $O(d)$ constraints, i.e., $\mathcal{C}_p$ contains $O(d)$ non-redundant constraints. In this case, after $O(d)$ questions, we either move to another set $\mathcal{S}_q$ or confirm the user's utility vector is in $\mathcal{S}_p$.

Since these M points are uniformly distributed, the sets $\mathcal{S}_p$ are also uniformly distributed. We can consider that they form a large cube and each dimension is divided into $M^{1/d}$ segments. In the worst case, moving from the one cube to the correct one may require traversing all such segments in every dimension. Therefore, the path length is bounded by $O(dM^{1/d})$.

Once the target region is located, we proceed with another round of sampling and path construction. Each round reduces the unresolved candidate space by a factor of M. Thus, after $O(\log_M n)$, we can locate the target point. The total number of questions asked is $O(d) \cdot O(dM^{1/d}) \cdot O(\log_M n) = O(d^2 M^{1/d} \cdot \log_M n)$. □