Differentiating Search Results on Structured Data

ZIYANG LIU and YI CHEN, Arizona State University

Studies show that about 50% of Web search is for *information exploration* purposes, where a user would like to investigate, compare, evaluate, and synthesize multiple relevant results. Due to the absence of general tools that can effectively analyze and differentiate multiple results, a user has to manually read and comprehend potential large results in an exploratory search. Such a process is time consuming, labor intensive and error prone. Interestingly, we find that the metadata information embedded in structured data provides a potential for automating or semi-automating the comparison of multiple results.

In this article we present an approach for structured data search result differentiation. We define the differentiability of query results and quantify the degree of difference. Then we define the problem of identifying a limited number of valid features in a result that can maximally differentiate this result from the others, which is proved NP-hard. We propose two local optimality conditions, namely single-swap and multi-swap, and design efficient algorithms to achieve local optimality. We then present a feature type-based approach, which further improves the quality of the features identified for result differentiation. To show the usefulness of our approach, we implemented a system CompareIt, which can be used to compare structured search results as well as any objects. Our empirical evaluation verifies the effectiveness and efficiency of the proposed approach.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Selection process*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Keyword search, structured data, databases, XML data, differentiation, comparison, result analysis

ACM Reference Format:
Liu, Z. and Chen, Y. 2012. Differentiating search results on structured data. ACM Trans. Datab. Syst. 37, 1, Article 4 (February 2012), 30 pages.
DOI = 10.1145/2109196.2109200 http://doi.acm.org/10.1145/2109196.2109200

1. INTRODUCTION

Studies show that about 50% of keyword searches on the Web are for *information exploration* purposes, and inherently have multiple relevant results [Broder 2002]. Such queries are classified as *informational queries*, where a user would like to investigate, evaluate, compare, and synthesize multiple relevant results for information discovery and decision making, in contrast to *navigational queries* whose intent is to reach a particular Web site. Without the help of tools that can automatically or semi-automatically analyze multiple results, a user has to manually read, comprehend, and analyze the results in informational queries. Such a process can be time consuming, labor intensive, error prone or even infeasible due to possibly large result sizes.

This material is based upon work partially supported by NSF CAREER award IIS-0845647, IIS-0915438, an IBM Faculty award and a Google Research award.
Authors' addresses: Z. Liu (corresponding author) and Y. Chen, Arizona State University; email: {ziyang.liu, yi}@asu.edu.

DOI 10.1145/2109196.2109200 http://doi.acm.org/10.1145/2109196.2109200

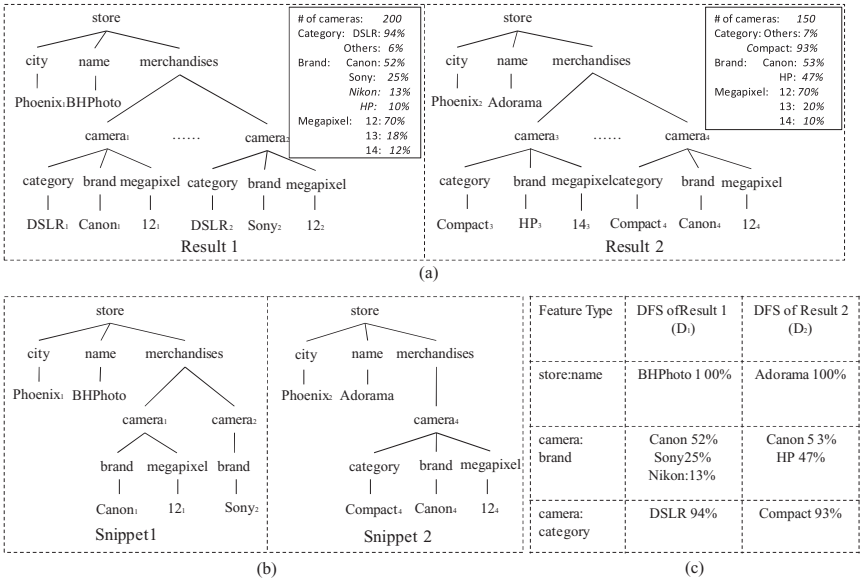

Fig. 1. Two results of query "*Phoenix, camera, store*" (a); their snippets with size limit = 14 (b); and differentiation feature sets with size limit = 5 (c).

For example, consider a customer who is looking for *stores* that sell *camera*s in *Phoenix* and issues a keyword query "*Phoenix, camera, store*". There are many results returned, where the fragments of two results by searching structured data are shown in Figure 1(a) and some statistics information of the results is shown next to the results. As each store sells hundreds of cameras, it is very difficult for users to manually check each result, compare and analyze these results to decide which stores to visit.

To help users analyze search results, the Web sites of banks and online shopping companies, such as Citibank, Best Buy, etc., provide comparison tools for customers to compare specific products based on a set of predefined metrics, and have achieved big success. However, in these Web sites, only predefined types of objects (rather than arbitrary search results) can be compared, and the comparison metrics are predefined and static. Furthermore, usually only a single type of object is compared, for example, a store, but not its related objects, for example, cameras. Such hard coded approaches are inflexible, restrictive, and not scalable.

A general and widely used method that helps users judge result relevance without checking the actual results is to generate snippets. By summarizing each result and its relevance to the query, snippets are very popular and useful, and thus have been supported by not only every Web search engine but also some structured data search engines. However, without considering the relationships among results, in general, the snippets are not helpful to compare and differentiate multiple results.

For example, Figure 1(b) shows the snippets of results in Figure 1(a) generated by eXtract [Huang et al. 2008], which has been developed for generating result snippets for keyword search on structured data, given the upper bound of snippet size of 14 edges. These snippets highlight the most dominant features in the results. As we can see from the statistics information in Figure 1(a), the store in result 1 mainly sells *Canon* and *Sony* cameras, while the store in result 2 mainly sells *Canon*'s *Compact* cameras. However, snippets are generally not comparable. From their snippets, we know result 2 focuses on *Compact* cameras, but have no idea whether or not result 1 focuses on *Compact* or *DSLR*, since the information about *category* of cameras sold by

the store is missing in its snippet. Similarly, result 1 has many *Sony* cameras, but we do not have information about whether result 2 has many *Sony* cameras or not. As we can see, snippets are not designed to help users find out the differences among multiple results.

Although a general tool for informative query result comparison is very useful in diverse domains, it is not supported in existing text search engines. The main reason is that text documents are unstructured, making it extremely difficult if not impossible to develop a tool that automatically compares the semantics of two documents.

On the other hand, when searching structured data, the structural information of the results may provide valuable metadata, and thus present a potential to enable result comparison. For example, directly generating a "comparison table" of an apple and an orange based on two general textual descriptions is difficult, but it becomes possible if the description is presented in structured format, with markups in XML or column names in relational databases to hint their features such as size, color, isFruit, and so on.

However, many challenges remain, even for enabling structured result comparison. For example, which features in the search results should be selected for result comparison? One desideratum is, of course, such features should maximally highlight the differences among the results. Then, how should we define the difference, and the degree of differentiation of a set of features? Another desideratum is, the selected features should reasonably reflect the corresponding results, so that the differences shown in the selected features reflect the differences in the corresponding results. Furthermore, how should we select desirable features from the results efficiently?

In this article, we present the techniques for structured data search result comparison and differentiation, which takes as input a set of structured results, and outputs a Differentiation Feature Set (DFS) for each result to highlight their differences within a size bound. To show the usefulness of our technique in the real world, we develop a structured search result differentiation system named CompareIt, and use both real and synthetic data to evaluate our algorithms in experiments. The CompareIt system can take the results generated by any of the existing keyword search engines on structured data as the input and generate DFSs for result differentiation. In fact, the generated DFSs can also be used to compare results of structured query (e.g., XPath, XQuery, SQL) upon user request. Sample DFSs for the query results in Figure 1(a) are shown in Figure 1(c). The contributions of this work include the following.

—We initiate the study of differentiating keyword search results, which is critical for diverse application domains, such as online shopping, employee hiring, job/ institution hunting, etc. Note that although we use XML-based examples for discussion and experiments, our proposed method is applicable to general query results which have features defined as (entity, attribute, value).
—We identify three desiderata of selecting Differentiation Feature Set (DFS) from query results in order to effectively help users compare and contrast results.
—We propose an objective function to quantify the degree of differentiation among a set of DFSs, and prove that the problem of identifying valid features that maximize the objective function given a size limit is NP-hard.
—We propose two local optimality criteria which judge the quality of an algorithm for generating DFSs: single-swap optimality and multi-swap optimality, and developed efficient algorithms to achieve these criteria.
—We also propose an improved method of generating DFSs: a feature type-oriented approach, which generally generates DFSs with better quality than the local optimal algorithms. We present two algorithms for the feature type-oriented approach: one based on exact calculation and another based on heuristics.

—The effectiveness and efficiency of the proposed approach is verified through extensive empirical evaluation.
—CompareIt can be used to augment any existing structured data management system to provide the functionality of helping users easily compare (selected) query results.

The rest of the article is organized as follows: Section 2 introduces three desiderata of selecting features from results for comparison purpose, formalizes the problem definition, and shows the NP-hardness of the problem. Section 3 discusses two local optimality criteria and presents efficient algorithms to achieve them. Section 4 introduces the feature type-oriented method, an improved approach for generating differentiating feature sets for the selected results. Section 5 reports results of empirical evaluations. Section 6 discusses related works and Section 7 concludes the article.

2. PROBLEM DEFINITION: CONSTRUCTING DIFFERENTIATION FEATURE SETS

In this section we first review some background on data models and features (Section 2.1). Then we discuss three desiderata for *Differentiation Feature Set* (DFS): limited size (Section 2.2.1), reasonable summary (Section 2.2.2), and maximal differentiation (Section 2.2.3). While maximal differentiation is the optimization goal in generating DFSs, limited size and reasonable summary are necessary conditions: the former ensures that the DFSs can be easily checked by a user, and the latter ensures that the comparison based on DFS correctly reflects the comparison of results. Then we formalize the problem of generating optimal DFSs for a set of query results with a size bound and prove the NP-hardness of the problem (Section 2.3).

2.1. Preliminaries

Our proposed approach is applicable to general query results on structured data such as relational database, XML, etc., with the features defined as (entity, attribute, value). Therefore, the first issue is that how can we infer the entity-attribute-value information from structured data (i.e., relational database and XML).

Data in relational database is very well organized based on the traditional entity-relationship model. Therefore, entities can be easily inferred as the entities specified by users. Column names and the values in the cells can be considered as attributes and values, respectively.

XML data is modeled as a rooted labeled tree. Results of a query Q on XML data D is a set of trees that can be obtained using any of the existing XML search engines. Entities and attributes can be identified by the heuristics proposed in Liu and Chen [2007]. Specifically, a node is an entity if it corresponds to a *-node in the DTD, that is, it has siblings with the same label. A node is an attribute if it is not an entity and has only one leaf child.

2.2. Desiderata of Differentiation Feature Sets

Next, we discuss three desiderata for a set of DFSs.

2.2.1. Being Small. To enable users to quickly differentiate query results, the first desideratum of a set of DFSs is: *being small*, so that users can quickly browse and understand them. The upper bound size of a DFS can be specified by the user.

Desideratum 1 (*Small*). The size of each DFS D, denoted as $|D|$, is defined as the number of features in D. $|D|$ should not exceed a user-specified upper bound L, that is, $|D| \leq L$.

2.2.2. Summarizing Query Results. For the comparisons based on DFSs to be valid, a DFS should be a reasonable summary of the corresponding result by capturing the

main characteristics in the result. Otherwise, the differences shown in two DFSs may not be meaningful.

Example 2.1. Consider again the two results of query "*Phoenix, camera, store*" in Figure 1(a). Both results mainly sell *Canon* cameras. Each store also sells some *HP* cameras. Suppose we have the DFS for result 1, D_1={store:brand:HP}, and the DFS for result 2, D_2={store:brand:Canon}. Obviously these two DFSs are different. However, these DFSs are not meaningful, since it gives the user a wrong impression: the difference of these two stores is that the first store mainly sells HP cameras, and the second store mainly sells Canon cameras. Obviously, this is not true. Intuitively, a feature that has more occurrences in the result should have a higher priority to be selected in the DFS, so that the DFS reflects the most important feature in the result, and the differences among DFSs correctly reflect the main differences of their corresponding results.

Furthermore, although both stores in these results sell *Canon* and *HP*, it is undesirable to simply output a single occurrence of *Canon* and *HP* in the DFS of each result. Such DFSs give users the impression that the two stores are similar in terms of their speciality on *Canon* and *HP*. However, the store in result 1 mainly focuses on *Canon* with just a couple of *HP*; whereas the store in result 2 focuses on both *Canon* and *HP*, with roughly the same number of cameras. Intuitively, the DFS should capture the distributions of features of the same type.

As we can see from Example 2.1, a valid DFS should be a reasonable summary of the result, so that the important differences of the results can be captured in the DFSs. Thus we define the *validity* of a DFS as the following.

Desideratum 2 (*Validity*). A DFS D is valid with respect to a result R, denoted as *valid*(D, R), if and only if the following rules are satisfied.
(1) *Dominance Ordered*. A feature can be included in D only if the features of the same type that have more occurrences in R are already included in D. That is, features of the same type should be ordered by dominance (defined as their number of occurrences).
(2) *Distribution Preserved*. A DFS should capture the distributions of features of the same type.

To ensure a DFS satisfies Dominance Ordered, we sort the features of the same type in each result by their number of occurrences. Features of the same type with the same number of occurrences can be sorted in any way that is uniform for all results. We use alphabetical order in this article. There are various ways to achieve Distribution Preserved. In this article, for each feature output in a DFS, we also show its percentage of occurrences within its feature type, such as the DFSs in Figure 1(c). Another way of achieving *Distribution Preserved* is to use font sizes to represent the percentages of features: features with higher percentage are shown in bigger fonts, which is analogous to Tag Cloud. Note that in this case, we may want to use a "weighted size" of each feature: a feature with higher percentage occupies more space of the DFS, since it has a bigger font.

Example 2.2. In result 1 in Figure 1(a), features of type *camera:brand*, in the descending order of their dominance, are *Canon*, *Sony*, *Nikon*, and *HP*. Then in order for its DFS to be valid, feature *Nikon* can be included in the DFS only if both features preceding it, *Canon* and *Sony*, are already included.

2.2.3. Differentiating Query Results. Being small and a good summary are necessary conditions for a DFS, yet they are insufficient.[1] In this section, we propose the unique and

[1]Indeed snippets are generally small and summarize results, nevertheless they are ineffective for result comparison and differentiation, as discussed in Section 1.

most challenging requirement for a good DFS: *differentiability*, that is, a set of features that can differentiate one result from others.

Differentiability of DFSs. We define that two results are comparable by their DFSs if their DFSs have common features types. Two results are differentiable if the DFSs have different characteristics of those shared feature types.

Intuitively, features of different types are not comparable, for example, we are not able to compare *camera:brand:Canon* in result 1 with *camera:category:compact* in result 2. Therefore, we consider each feature type as the differentiation unit. Each feature type can be considered as a vector, in which each component represents a feature of this type in the DFS, and the value of a component is the percentage of the feature. Typical ways of measuring the distance of two vectors include L_1 distance (i.e., Manhattan distance), Euclidean distance, cosine similarity, etc. Intuitively, the degree of difference of a feature type can be considered as the sum of percentage differences of all its values. Therefore, we use L_1 distance to define the degree of difference of a feature type. Other distance metrics can also be used.

There are also some approaches for measuring the distance of probability distributions, most of which can also be applied to compute the degree of difference of a feature type. However, note that some of them are not applicable. For example, some of these metrics are not symmetric, for example, KL divergence [Kullback 1987]. The KL divergence of v_1 and v_2 and that of v_2 and v_1 are generally different, which is not suitable for our approach, since the difference of two feature types should intuitively be symmetric. Some other metrics, for example, Earth Mover's Distance [Rubner et al. 1998], are sensitive to the order of the components in the vector, which is also undesirable for the purpose of computing the difference of a feature type.

Note that there is an important issue when modeling a feature type in a DFS as a vector. Given two DFSs, if a feature is included in both DFSs, its difference is simply the difference of percentage in the two DFSs. However, if it is not included in a DFS, the corresponding values in the vector should *not* always be 0. This is because a DFS only records partial information in the corresponding result, that is, a feature that does not appear in a DFS may have occurrences in the results. If a feature is not included in any DFS, then the corresponding value in the vector should be considered as 0%, since the user cannot see this feature, and thus cannot see its difference in the results. But if a feature is included in some of the DFSs but not the others, we should determine how much difference the user can deduce from the DFSs about this feature. Let us look at an example.

Example 2.3. For the two results in Figure 1(a), suppose feature type *camera:brand* in the two DFSs are as follows.
D_1: Canon: 52%, Sony: 25%
D_2: Canon: 53%, HP: 47%

Then for *Canon*, the difference is 1%. For *Sony*, its percentage is 25% in result 1. In result 2, since *Canon* and *HP* already sum up to 100%, there is no *Sony* in result 2, thus the difference of *Sony* should be 25%. For *HP*, its percentage in result 2 is 47%. Since in result 1, *Canon* and *Sony* sum up to 77%, the percentage of *HP* in result 1 is at most 23%. Thus the percentage difference of *HP* in the two results is at least $47\% - 23\% = 24\%$. For any other feature of this type, since it is not shown in either DFS, its difference is 0. Therefore, for this feature type, by L_1 distance, its degree of difference in the two results is $1\% + 25\% + 24\% = 50\%$.

As we can see, if we simply consider the percentage of HP in D_1 as 0%, then we would conclude that the difference of HP is 47%, but the real difference may be only 24%. In other words, from these two DFSs, the user can only deduce a difference of 24% for HP.

Consider another example, in which the two DFSs are as follows.
D_1: Canon: 40%, Sony: 30%
D_2: HP: 20%

For *Canon*, in result 2, its percentage is at most 20% (since we output the features in the order of their percentage). Thus the difference of *Canon* is at least 20%. Similarly, for *Sony*, its difference is at least 10%. On the other hand, for HP, the percentage in result 1 is at most 30%. Therefore, the difference of *HP* in the two DFSs is 0%, because there is a possibility that its percentage is 20% in result 1. Therefore, the total degree of difference of this feature type is 20% + 10% = 30%.

As we can see, if we consider the percentage of HP in D_1 as 0%, we would get a difference of 20% for HP, which may not be true.

Example 2.3 gives us an idea of how to define the degree of difference of a feature type in two DFSs. Intuitively, the degree of difference of a feature type depends on how many features of this type we can differentiate, and how much difference of each feature can be deduced in the two DFSs. For a feature F, if F appears in both DFSs, then we simply use its percentage difference in the two DFSs. If it does not appear in one or both DFSs, we use the lower bound of difference of this feature type in the two DFSs that we can infer. For a feature type T, we sum up the differences of all features F of type T in the two results. If we consider feature type T as a vector in each DFS with its features as components, then the degree of difference of T is the L_1 distance of these two vectors. The formal definition is given next.

Definition 2.1. Given a feature type T in two DFSs D_1 and D_2, the *degree of difference of T in D_1 and D_2*, denoted by $DoD_T(D_1, D_2)$, is computed as

$$DoD_T(D_1, D_2) = \sum_{F \in T} diff(F),$$

where F is a feature of type T. $diff(F)$ is computed as follows.

—If F is included in both D_1 and D_2, let p_1 and p_2 be the percentage of F in D_1 and D_2. The difference of F is $|p_1 - p_2|$.
—If F is included in D_1 but not D_2, let p_1 be the percentage of F_1 in D_1. We first compute p_2 as the maximum possible percentage of F in D_2. p_2 is the smaller of the following two numbers: (1) the percentage of the last feature of type T output in D_2; (2) 1− the total percentage of all features of type T output in D_2. If $p_1 \geq p_2$, the difference of F is $p_1 - p_2$. Otherwise, the difference of F is 0.
—If F is included in D_2 but not D_1, its difference is measured in a similar way as before.
—If F is not included in either D_1 or D_2, its difference is 0.

Example 2.4. Consider the two DFSs in Figure 1(c). For feature type *store:name*, we have $diff(BHPhoto) = 1$, $diff(Adorama) = 1$, thus $DoD_{store:name}(D_1, D_2) = 2$. For feature type *camera:brand*, $diff(Canon) = 0.01$, $diff(Sony) = 0.25$, $diff(Nikon) = 0.13$, $diff(HP) = 0.37$. Therefore, $DoD_{camera:brand}(D_1, D_2) = 0.76$. For feature type *camera:brand*, $diff(DSLR) = 0.87$ and $diff(compact) = 0.87$, thus $DoD_{camera:category}(D_1, D_2) = 1.74$.

Note that for nonnegative numbers $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$, we have

$$\sum_{i=1}^{n} |a_i - b_i| \leq \sum_{i=1}^{n} a_i + b_i.$$

Therefore, for any feature type T and two DFSs D_1 and D_2, $DoD_T(D_1, D_2)$ is always between 0 and 2.

Given the definition of the DoD of a feature type T in two results, we define the DoD of a feature type T in multiple DFSs as the sum of the DoD of T in every pair of DFSs, and also define the total DoD of multiple DFSs as the sum of the DoD of all feature types in those DFSs.

Definition 2.2. Given a set of DFSs $D_1, \ldots, D_n$, the DoD of a feature type T in these DFSs is defined as

$$DoD_T(D_1, \ldots, D_n) = \sum_{1 \le i \le n} \sum_{i < j \le n} DoD_T(D_i, D_j).$$

The total DoD of these DFSs is defined as

$$DoD(D_1, \ldots, D_n) = \sum_{T} DoD_T(D_1, \ldots, D_n).$$

Example 2.5. In the two DFSs in Figure 1(c), we have $DoD_{store:name}(D_1, D_2) = 2$, $DoD_{camera:brand}(D_1, D_2) = 0.76$ and $DoD_{camera:category}(D_1, D_2) = 1.74$. Thus $DoD(D_1, D_2) = 3.50$.

We have the following Desideratum 3 for differentiation feature sets.

Desideratum 3 (*Differentiability*). Given a set of results $R_1, R_2, \ldots, R_n$, their DFSs, $D_1, D_2, \ldots, D_n$, should maximize the total degree of differentiation defined in Definition 2.2.

We will show in the next subsection that, unfortunately, generating valid and small DFSs that maximize their DoD is NP-hard.

2.3. Problem Definition and NP-Hardness

In this section, we formally define the problem of generating DFSs for search result differentiation and analyze its complexity.

As we discussed in Sections 2.2.1 through 2.2.3, given a set of results, their DFSs should maximize the DoD, that is, the total degree of differentiation, and the DFSs should be valid with respect to the corresponding result, and be small.

Definition 2.3. The *DFS construction problem* $(R_1, R_2, \ldots, R_n, L)$ is the following: given n search results $R_1, R_2, \ldots, R_n$, compute a DFS D_i for each result R_i, such that:

—$DoD(D_1, D_2, \ldots, D_n)$ is maximized.
—$\forall i$, $valid(D_i, R_i)$ holds.
—$\forall i$, $|D_i| \le L$.

THEOREM 2.6. *The DFS construction problem is NP-hard.*

PROOF. We prove the NP-completeness of the decision version of the DFS construction problem by reduction from X3C (exact 3-set cover). The decision version of the DFS construction problem is: given n results $R_1, R_2, \ldots, R_n$, is it possible to generate a DFS D_i for each result R_i, such that $valid(D_i, R_i, p)$, $|D_i| \le L$, and $DoD(D_1, D_2, \ldots, D_n) \ge S$?

This problem is obviously in NP, as computing the DoD of a set of DFSs can be done in polynomial time. Next we prove the NP-completeness.

Recall that each instance of X3C consists of:

—a finite set X with $|X| = 3q$;
—a collection C of 3-element subsets of X, that is, $C = \{C_1, C_2, \ldots, C_l\}$, $|C| = l$, $C_i \subseteq X$ and $|C_i| = 3$.

The X3C problem is whether we can find an exact cover of X in C, that is, a subcollection C^* of C, such that every element in X is contained in exactly one subset in C^*.

Now we transform an arbitrary instance of X3C to an instance of the DFS construction problem. We construct an instance of the DFS construction problem, in which there are $3q$ query results, and l different feature types. Each $C_i \in C$ corresponds to a feature type t_i, which has three different features: F_{i1}, F_{i2}, F_{i3}. For each $C_i = \{X_a, X_b, X_c\}$ in the X3C instance, let feature type t_i appear once in the X_a-th, X_b-th, and X_c-th results, with feature F_{i1}, F_{i2}, and F_{i3}, respectively. Note that in this way, all features have a percentage of 100% in each results. Let the DFS size limit L be 1, that is, there can only be one feature in each DFS. The question is: can we find a DFS for each of the $3q$ results, such that $DoD(R_1, \ldots, R_{3q}) \geq 6q$?

If we can find an exact cover C^* for the X3C instance, then we select the corresponding q feature types. For each selected feature type, we add its 3 features to the corresponding 3 DFSs. In this way, each DFS has exactly one feature. Each feature type contributes 6 to the DoD, thus the total DoD is $6q$.

If we can find a set of DFSs such that their DoD is $6q$, then it is easy to see that we must find q feature types, and for each feature type, all its 3 features must appear in the corresponding DFSs. Otherwise, if a feature type has only 1 feature appearing in the DFSs, then it does not contribute to the DoD; if it has 2 features appearing in the DFSs, it takes 2 slots but only contributes 2 to the DoD, making the total DoD impossible to reach $6q$.

This means that there is an exact cover for the instance of X3C if and only if we can find a set of DFSs with a DoD of $6q$. Therefore, it is a reduction. Since this reduction obviously can be performed in polynomial time, the decision version of the DFS construction problem is NP-complete, and the DFS construction problem is NP-hard.

3. LOCAL OPTIMALITY AND ALGORITHMS

Due to the NP-hardness of the DFS construction problem, in order to address the problem with good effectiveness and efficiency, we propose two local optimality criteria: single-swap optimality and multi-swap optimality. An algorithm that satisfies a local optimality criterion does not necessarily produce the best possible result, but always produces results that are good in a local sense. We show in Section 3.1 that single-swap optimality can be achieved efficiently in polynomial time. On the other hand, multi-swap optimality is more challenging to achieve, as a naive algorithm would be exponential. We present an efficient dynamic programming algorithm in Section 3.2 that realizes multi-swap optimality.

3.1. Single-Swap Optimality

In this section we present the first local optimality criterion, *single-swap optimality*, for the DFS construction problem, and present a polynomial-time algorithm achieving it.

Definition 3.1. A set of DFSs is *single-swap optimal* for query results $R_1, R_2, \ldots, R_n$ if, by changing or adding one feature in a DFS D_i of R_i, $1 \leq i \leq n$, while keeping $valid(D_i, R_i)$ and $|D_i| \leq L$, their degree of differentiation, $DoD(D_1, D_2, \ldots, D_n)$, cannot increase.

Let us look at an example.

Example 3.1. The two DFSs in Figure 1(c) satisfy single-swap optimality, that is, changing or adding any feature won't increase their DoD. For instance, if we change *camera:brand:HP, 47%* in D_2 to *camera:megapixel:12, 70%*, then the

ALGORITHM 1: Algorithm for Single-Swap Optimality

constructDFS (Query Results: $QR[n]$; Size Limit: L)

1: **for** i = 1 to n **do**
2: arbitrarily generate $DFS[i]$ for $QR[i]$
3: **for** i = 1 to n **do**
4: **for** each feature type t in $DFS[i]$ **do**
5: f = the next feature of type t that is in $result[i]$ but not in $DFS[i]$
6: add f into $DFS[i]$
7: $sizeinc$ = the size of feature f {features may have different sizes, e.g., a feature with long text or large font may have a bigger size}
8: $DFS[i].size+ = sizeinc$
9: **if** $DFS[i].size > L$ **then**
10: remove f from $DFS[i]$
11: $DFS[i].size- = sizeinc$
12: **else**
13: $benefit$ = computeBenefit(DFS, i, t, f, null, null)
14: **if** $benefit > 0$ **then**
15: goto line 3
16: **else**
17: remove f from $DFS[i]$
18: **for** each feature type t' in $result[i]$ **do**
19: f = the last feature of type t in $DFS[i]$
20: f' = the next feature of type t' that is in $result[i]$ but not in $DFS[i]$
21: change the occurrences of f to $occ_{f'}$ occurrences of f' in $DFS[i]$
22: **if** $DFS[i].size > L$ **then**
23: undo the change from f to f'
24: **else**
25: $benefit$ = computeBenefit(DFS, i, t, f, t', f')
26: **if** $benefit > 0$ **then**
27: goto line 3
28: **else**
29: undo the change from f to f'

computeBenefit ($DFS[n]$; i; Feature Type: t; Feature Value: f; Feature Type: t'; Feature Value: f')

1: {This function computes the delta DoD after adding f' to $DFS[i]$ and removing f from $DFS[i]$}
2: $benefit = 0$
3: **for** j = 1 to n **do**
4: **if** $j = i$ **then**
5: continue
6: $newDoD = DoD_{t'}(DFS[i], DFS[j])$ (Definition 2.1) {$newDoD$ is the current DoD of feature type t'}
7: remove f' from $DFS[i]$
8: $oldDoD = DoD_{t'}(DFS[i], DFS[j])$ {$oldDoD$ is the DoD of feature type t' before adding f'}
9: $benefit = benefit + newDoD - oldDoD$
10: add f' to $DFS[i]$
11: $newDoD = DoD_t(DFS[i], DFS[j])$ {$newDoD$ is the current DoD of feature type t}
12: add f to $DFS[i]$
13: $oldDoD = DoD_t(DFS[i], DFS[j])$ {$oldDoD$ is the DoD of feature type t before adding f}
14: $benefit = benefit + newDoD - oldDoD$
15: remove f from $DFS[i]$
16: return $benefit$

$DOD_{camera:brand}(D_1, D_2)$ reduces from 76% to 1%. On the other hand, D_1 and D_2 are still not differentiable on *camera:megapixel*, since there is no feature of this type in D_1. Thus their DoD decreases by 75%.

Single-swap optimality can be achieved by a polynomial-time algorithm: enumeration. The pseudocode of this algorithm is presented in Algorithm 1. There are four steps.

(1) *Initialization.* We start with a randomly generated valid DFS for each result, satisfying the size limit (procedure CONSTRUCTDFS lines 1–2).
(2) *Checking.* Performing an iteration of checking and updating DFSs (lines 3–29). For each DFS $DFS[i]$, we check whether the DoD of all DFSs can increase after adding a feature of type t to $DFS[i]$ (lines 4–17) or switching an existing feature of type t to a new feature of type t' that is currently not in DFS (lines 18–29).
(3) *Updating and Iteration.* If such a DFS is found, then we make the update and restart the iteration in step 2 (lines 15 and 27).
(4) *Termination.* If there is no DFS that can be changed to further improve the DoD, then we terminate and output the DFSs.

As we can see, in the *Initialization* step, DFSs are generated randomly. In fact, the initialization of DFS does not affect the local optimality of the proposed algorithms, but has impact on the generated DFSs and where a local optimal point is achieved. Investigation of good DFS initialization is an orthogonal problem.

Although the high-level description of the algorithm and the example look simple, there are three technical challenges to be addressed. First, when updating a feature in a DFS, we must ensure its validity with respect to the corresponding result and satisfaction of the size limit. Since each DFS must be valid, the addition of a feature to a DFS must be in the dominance order of this feature type, and the removal of features from a DFS must be in the reverse order of feature dominance. For single-swap optimality, we only check whether altering *one* feature can improve the DoD. Thus, to add a feature of type t to a DFS, only the most dominant feature of type t that is not in the DFS can be added; to remove a feature of type t', only the least dominant feature of t' that is in the DFS can be removed. Let us look at an example.

Example 3.2. To explain the single-swap optimal algorithm, we use the two results in Figure 1(a), and another result whose statistics information is shown in Figure 2(a), as a running example. Suppose that the three DFSs are randomly initialized as in iteration 0 in Figure 2(b), and that the size limit for each DFS is 5. The algorithm updates one DFS for one of the three results in the 4 iterations as shown. In iteration 1, the algorithm attempts to add a feature of type *camera:category* to D_1. The feature to be added must be the most dominant one of this type: *DSLR*. The addition can increase $DoD_{camera:category}(D_1, D_3)$, and thus increase the DoD of the three DFSs.

The second challenge is that, due to the interactions among DFSs, one DFS may need to be updated multiple times, where the number of updates cannot be determined before the termination of the algorithm.

Example 3.3. Continuing Example 3.2, in iteration 2, Algorithm 1 tries to add *Compact, 93%*, the most dominant feature of type *camera:category*, to D_2. This increases the total DoD. Note that at this time, adding (*camera: megapixel: 12, 70%*) to D_2 does not increase the DoD, as D_1 has exactly the same (feature, precentage) pair, and D_3 does not have feature type *camera:megapixel*. However, after adding (*camera: megapixel: 10, 70%*) to D_3 in iteration 3, it becomes valuable to add (*camera: megapixel: 12, 70%*) to D_2 in iteration 4, which will increase the total DoD. As we can

of cameras: *120* Category: Compact: *80%*; Others: *20%* Brand: Nikon: *60%*; Kodak: *35%;* Others: *5%* Megapixel: 10: *70%*; 11: *18%*; 12: *12%*

(a) Statistics Information of Result 3

iteration	D_1	D_2	D_3
0	store: name: BHPhoto, 100% camera: brand: Canon, 52% camera: megapixel: 12	store: name: Adorama, 100% camera: brand: Canon, 53% camera: brand: HP, 47%	store: name: Porter's, 100% camera: brand: Nikon, 60% camera: category: Compact, 80%
1	store: name: BHPhoto, 100% camera: brand: Canon, 52% camera: megapixel: 12, 70% *camera: category: DSLR, 94%*	same as above	same as above
2	same as above	store: name: Adorama, 100% camera: brand: Canon, 53% camera: brand: HP, 47% *camera: category: Compact, 93%*	same as above
3	same as above	same as above	store: name: Porter's, 100% camera: brand: Nikon, 60% camera: category: Compact, 80% *camera: megapixel: 10, 70%*
4	same as above	store: name: Adorama, 100% camera: brand: Canon, 53% camera: brand: HP, 47% camera: category: Compact, 93% *camera: megapixel: 12,70%*	same as above

(b) Iterations Performed by Algorithm 1

Fig. 2. Running example of Algorithm 1.

see, after D_2 was first checked and updated in iteration 2, it needs to be updated again to further improve the DoD after other DFSs are updated.

The iteration continues until no DFSs can be added or changed to improve the DoD. Since the number of times that we may update a DFS is unknown, one question is whether the algorithm terminates and how many iterations will be performed. As will be analyzed shortly, this enumeration algorithm is guaranteed to run in polynomial time in terms of the number of results (n) and the number of features (m).

The third challenge is that we need to compute the delta of DoD upon an altered or added feature (Procedure COMPUTEBENEFIT). Note that adding a feature to a DFS may not increase the DoD, and removing a feature from a DFS may not decrease the DoD. After each iteration of Algorithm 1, we compute the DoD of the altered feature type according to Definition 2.1, then update the total DoD of all DFSs.

Now we analyze the complexity of the Algorithm 1. Let n be the number of query results, and m be the number of feature types in a result.

—In each iteration, we check at most n DFSs. For each DFS $DFS[i]$ (in a single iteration), we check at most m^2 feature pairs to see whether an existing feature should be replaced, and check at most m features to see whether a feature should be added. As discussed earlier, for each feature type, we have to check the features with respect to their dominance order, thus there are only m choices of feature swap or addition for one result in one iteration. Each check will compute the delta of DoD by invoking Procedure COMPUTEBENEFIT. Procedure COMPUTEBENEFIT computes the DoD of feature type t, we sort the features of type t in both DFSs, then scan them and compute the DoD of type t according to Definition 2.1. Therefore, it takes $O(L\log L)$ time, where L is the DFS size limit. Thus, COMPUTEBENEFIT takes $O(nL\log L)$, and each iteration takes at most $O(n^2m^2L\log L)$ time.

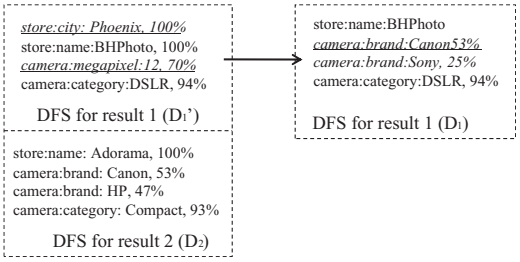

Fig. 3. Single-swap optimality and multiswap optimality.

—In each iteration except the last one, the DoD of the DFSs at least increases by 1%. The maximum possible DoD for each feature type in two results is 2, thus the maximum possible DoD for two results is $2m$, and the maximum DoD of all n DFSs is bounded by $O(n^2m)$. This means we need at most $O(n^2m)$ iterations, and thus the algorithm runs in polynomial time in terms of n and m.

3.2. Multi-Swap Optimality

After discussing *single-swap optimality*, we propose *multi-swap optimality*, a stronger criterion. Then we present an efficient dynamic programming algorithm to achieve it.

Recall that single-swap optimality guarantees that the DoD of a set of DFSs won't increase by changing one feature in a DFS. On the contrary, multi-swap optimality requires that the DoD cannot increase by changing *any number of* features in a DFS, as formally defined next.

Definition 3.2. A set of DFSs is *multiswap optimal* for query results $R_1, R_2, \ldots, R_n$ if, by making any changes to a DFS D_i of R_i, $1 \leq i \leq n$, while keeping $valid(D_i, R_i)$ and $|D_i| \leq L$, $DoD(D_1, D_2, \ldots, D_n)$ cannot increase.

Example 3.4. Figure 3 is an example of DFSs achieving single-swap optimality but not multi-swap optimality. D_1' and D_2 are DFSs of the two results in Figure 1(a) (suppose the percentage of feature *camera:brand:Canon* is 53% in Result 1). As we can see, $DoD(D_1, D_2)$ cannot be improved by changing or adding a single feature in either DFS. However, if we change (*store:city:Phoenix, 100%*) and (*camera:megapixel:12*, 70%) into (*camera:brand:Canon*, 53%) and (*camera:brand:Sony*, 25%), then feature type *camera:brand* now have a nonzero DoD in the two DFSs, and thus $DoD(D_1, D_2)$ increases.

In fact, achieving multi-swap optimality is more challenging than achieving single-swap optimality. Consider an enumeration-based algorithm, adapted from Algorithm 1. While keeping the *Initialization*, *Updating and Iteration*, and *Temination* steps the same, the *Checking* step is different. Instead of checking whether adding a single feature or swapping a single feature in a DFS can improve the DoD, we now need to check every possible combination of features in a DFS. Since the number of features in a query result is bounded by m, there can be up to 2^m different combinations of features in its corresponding DFS, leading to an exponential time complexity.

To efficiently achieve multi-swap optimality, we have designed a dynamic programming-based algorithm that runs in polynomial time with respect to n (the number of query results) and m (the maximum number of features in a result). We address the technical challenges in step 2 *Checking*: verifying whether there exists any change to a DFS, referred to as "target DFS", that can improve the total DoD. Instead of enumerating changes to a DFS (as the number of possible changes are exponential),

ALGORITHM 2: Algorithm for Multi-Swap Optimality

CONSTRUCTDFS (Query Results: $QR[n]$; Size Limit: L)

1: **for** $i = 1$ to n **do**
2: arbitrarily generate $DFS[i]$ for $QR[i]$
3: $DoD[i] = 0$
4: **for** $i = 1$ to n **do**
5: **for** $j = 1$ to n **do**
6: **for** each feature common feature type t in $DFS[i]$ and $DFS[j]$ **do**
7: $DoD[i] += DoD_t(DFS[i], DFS[j])$ (Definition 2.1)
8: **for** $i = 1$ to n **do**
9: $DoD', newDFS =$ CHECKDFS($QR[i], DFS, i, L$)
10: **if** $DoD' > DoD[i]$ **then**
11: $DFS[i] = newDFS$
12: $DoD[i] = DoD'$
13: goto line 9

CHECKDFS (Query Result: QR; DFSs: $DFS[n]$; i; Size Limit: L)

1: {This function constructs the optimal $DFS[i]$ given the other DFSs}
2: $t =$ number of feature types in QR
3: **for** $l = 1$ to L **do**
4: compute $s_{1,l}$ according to Figure 4
5: Suppose $s_{1,l}$ is maximized by outputting x features of type 1
6: $best_{1,l} = x$
7: **for** $k = 2$ to t **do**
8: **for** $l = 1$ to L **do**
9: compute $s_{k,l}$ according to Figure 4
10: Suppose $s_{k,l}$ is maximized by outputting x features of type k
11: $best_{k,l} = x$
12: $k = t$
13: $l = L$
14: $newDFS = \emptyset$
15: **while** $k > 0$ and $l > 0$ **do**
16: output x features of type k in $newDFS$
17: $k--$
18: $l -= x$
19: $DoD' = 0$
20: **for** $j = 1$ to n **do**
21: $DoD' +=$ the degree of differentiation between $DFS[i]$ and $DFS[j]$
22: return $DoD', newDFS$

our algorithm directly generates a valid multi-swap optimal target DFS, given the other DFSs.

To generate such a target DFS, we first need to determine for each feature type, what are the choices of selecting features to compose a valid DFS. Intuitively, for each feature type, there are multiple choices of including its features in the DFS, each with a different number of features included. To measure the effect of each choice, we define *benefit* and *cost* of a feature type. Specifically, if we include x features into the target DFS, then the cost is x, and the increase of DoD obtained by adding these x features is considered as benefit y.

Example 3.5. We use the query results in Figure 1(a) to explain the benefits and costs of a feature type. For feature type *camera:brand*, we have
$D_2 = \{Canon, 53\%, HP, 47\%\}$.
Consider D_1 as the target DFS. According to Figure 1(a), the list of features of this type in the order of their dominance in result 1 is $\{Canon, Sony, Nikon, HP\}$.

$$s_{k,l} = \begin{cases} \max\{b_{ki} \mid c_{ki} \le l\} & k = 1 \\ \max\{s_{k-1,l}, \max\{s_{k-1,l-c_{ki}} + b_{ki} \mid c_{ki} \le l\}\} & k > 1 \end{cases}$$

Fig. 4. Recurrence relation.

(1) If we have $D_1 = \{Canon,\ 52\%\}$, then cost=1, benefit=1%. This is because the percentage of *Canon* in D_1 and D_2 are 52% and 53%, respectively. Note that the difference of HP is 0%, since the maximum possible percentage of *HP* in D_1 is 48%.

(2) If we have $D_1 = \{Canon,\ 52\%, Sony,\ 25\%\}$, then cost=2, and benefit=50%, as illustrated in Example 2.3.

(3) If we have $D_1 = \{Canon,\ 52\%, Sony,\ 25\%, Nikon\ 13\%\}$, then cost=3, and benefit=76%, as now $diff(Nikon) = 13\%$ (increased by 13% compared with cost=2) and $diff(HP) = 37\%$ (increased by 13% compared with cost=2).

As we can see, for each feature type, there is a list of choices of how many features can be selected in a DFS, each with a benefit and a cost. We denote the aforesaid three choices as (1, 1%), (2, 50%), and (3, 76%), respectively.

Given the choices of generating valid DFSs discussed before, our goal is to calculate the optimal valid target DFS that can maximize the *DoD*, given the DFSs of the other results. We use $s_{m,L}$ to denote the maximum *DoD* that can be achieved by a valid optimal target DFS, where m is the total number of feature types in the result and L is the DFS size limit.

$s_{m,L}$ can be computed using dynamic programming. We give an arbitrary order to the feature types in the query result of target DFS. Let $s_{k,l}$ denote the maximum *DoD* that can be achieved by considering the first k feature types in the result, with DFS size limit l. Each $s_{k,l}$ is calculated using the recurrence relation discussed in the following.

—If $k = 1$, $s_{k,l}$ = the maximal benefit of the first feature type that can be achieved with cost not exceeding l.
—If $k > 1$, then we have multiple choices. We can choose not to include any feature of the k-th feature type at all, thus $s_{k,l} = s_{k-1,l}$. Otherwise, for the k-th feature type, suppose the list of feature selections that comprise a valid and small DFS is denoted as a list of benefit and cost pairs: (b_1, c_1), (b_2, c_2), and so on. We can choose any item in this list. For instance, if we choose to output c_1 features, then we can increase the benefit with b_1, but to accommodate the cost c_1, the first $k-1$ feature types can only include $l - c_1$ features, that is, $s_{k,l} = s_{k-1,l-c_1} + b_1$.

Therefore, the recurrence relation for calculating $s_{k,l}$ is shown in Figure 4, where we assume that the k-th feature type has p_k different benefit and cost pairs, (b_{k1}, c_{k1}), (b_{k2}, c_{k2}), ... (b_{kp_k}, c_{kp_k}), and $1 \le i \le p_k$.

The dynamic programming procedure that computes the optimal valid $DFS[i]$ is given in Algorithm 2 procedure CHECKDFS. We first compute $s_{1,l}$ for each l (lines 3–6), then compute $s_{k,l}$ as discussed. Meanwhile, we record array *best*, which is used to reproduce the optimal DFS, $newDFS$ (lines 12–18). Finally, DoD' is calculated by comparing $newDFS$ with every other DFS (lines 19–22).

The entire algorithm for multi-swap optimality is presented in Algorithm 2. Similar to Algorithm 1, it begins with randomly generating a DFS for each result (Procedure CONSTRUCTDFS lines 1–3). Then it computes $DoD[i]$, the total DoD between $DFS[i]$ and other DFSs (lines 4–7). In each iteration (lines 8–13), instead of tentatively making changes to each DFS as what Algorithm 1 does, this algorithm directly generates a valid multi-swap optimal $newDFS$ given the other DFSs, whose DOD is DoD', by invoking Procedure CHECKDFS. If DoD' is bigger than $DoD[i]$, then $DFS[i]$ is replaced

D1	D2
store:city:Phoenix, 100%	store:city:Phoenix, 100%
camera:brand:Canon, 52%	camera:brand:Canon, 53%
camera:megapixel:12, 70%	camera:megapixel:12, 70%

Fig. 5. A possible initialization of DFSs for the results in Figure 1(a).

by $newDFS$ with $DoD[i]$ updated (lines 10–13). Similar to Algorithm 1, Algorithm 2 terminates when no DFS can be changed to further improve the DoD.

Now we analyze the complexity of Algorithm 2. Let n, m, m', L denote the number of results, number of feature types, number of features, and DFS size limit, respectively. In procedure CHECKDFS, we first compute $newDFS$ using the equation in Figure 4 (lines 2–18), with complexity $O(m'L)$. Lines 19–21 of CHECKDFS compute the DoD of two DFSs. Since determining whether two DFSs can be differentiated on a given feature type takes $O(L\log L)$ time, the complexity of $newDFS$ is $O(m'L + mL\log L)$. In CONSTRUCTDFS, we first compute the DoD of every two results in $O(nmL\log L)$ (lines 4–7). Similar to Algorithm 1, the iteration in lines 8–13 is executed at most $O(n^2m)$ times. Therefore, the total complexity of Algorithm 2 is $O(n^2mL\log L(mn + m'))$.

As to be shown in Section 5, the algorithm is in fact quite efficient in practice, as the number of iterations is generally far less than n^2m.

4. FEATURE TYPE-ORIENTED DFS CONSTRUCTION

We have shown how to achieve two local optimality criteria in Section 3. Both of them consider one result at a time: single-swap optimal tries to change one feature in one result, and multi-swap optimal tries to change multiple features in one result. These two algorithms have the following disadvantage: the quality of the algorithms depends on the initialization. Specifically, if a good feature type does not have enough occurrences in the initialization, then it will not be chosen during the DFS generation.

Example 4.1. Consider the two results shown in Figure 1(a). Consider an initialization of the two DFSs as shown in Figure 5. In this case, no matter how we change a single DFS, we cannot increase the DoD. Specifically, for feature type *store:city*, both results have the same feature which is *Phoenix*. For *camera:brand*, if we only change one or more features in a single DFS, the DoD of this feature type will remain at 1%, and will not increase. Only when the DFSs of both results have two features included, they can have a larger DoD. It is the same for *camera:megapixel*. Besides, some feature types like *store:name* is not in the initial DFSs, and adding it to the DFS of one result does not increase the DoD either. Therefore both Algorithms 1 and 2, which only change one DFS at one time if the change increases the DoD, will not change the initial setting of both DFSs, resulting in low DoD of 1%.

Note that in contrast to single-swap and multi-swap optimality, another possible local optimality criteria is: the DoD of all DFSs cannot increase by changing any one feature in multiple DFSs. The preceding problem can be largely solved by achieving this local optimality criteria. Unfortunately, using the same proof as the one for Theorem 2.3, it is easy to see that achieving this local optimality criterion is NP-hard.

In observance of the previous problem, in this section, we propose heuristics algorithms which, although not necessarily achieving the aforesaid local optimality criteria, are superior to Algorithms 1 and 2 in that they consider multiple results together when generating DFSs. As we can see from the preceding example, Algorithms 1 and 2 may miss a good feature type if it does not have enough occurrences in all DFSs in the initialization. Thus our new algorithm considers a feature type at each time, rather

than a result. We refer to the algorithms based on this idea as *feature type-oriented DFS construction algorithms*.

The intuition of the feature type-oriented algorithms is that we can compute how "good" a feature type is, then select the feature types according to certain criteria. However, one barrier of this idea is: under the current problem setting, feature types cannot be completely considered as independent, which makes the problem much harder. For example, consider feature types t_1 and t_2. Let us assume that t_1 and t_2 can significantly differentiate many results, thus they both have a "high" quality. However, suppose that to significantly differentiate many results, both t_1 and t_2 require a large presence in a DFS D_i. Since D_i has a size limit, it may not be able to accommodate many occurrences of both t_1 and t_2, which means using t_1 and t_2 together may not be a good idea.

With this observation, to handle the interaction between feature types, we first attempt to solve an alternative problem, which is the same as the problem defined in Definition 2.3, except that there is a single size limit for *all* DFSs, rather than a size limit for each DFS. If the size limit of the individual DFS is L and there are n results, then we consider the size limit for all DFSs as $n \times L$. For this problem, we can measure the quality of each feature type independently of other feature types, and select the feature types accordingly. After we get a solution to this problem, since an individual DFS does not have a size limit, there may be some DFSs whose sizes exceed L and some other DFSs whose sizes are smaller than L. If this happens, then we iteratively remove some features from each DFS whose size exceeds L, and greedily add some features for each DFS whose size is smaller than L, which will be discussed later.

Apparently, the quality of a feature type depends on how many occurrences it is allowed to have. Recall that in Algorithm 2, for each feature type in the result being processed, we compute a set of (benefit, cost) pairs, then use dynamic programming to find the optimal number of occurrences of each feature type. The same idea can be adopted here. We can compute a set of (benefit, cost) pairs for each feature type with respect to *all* results. In other words, each (benefit, cost) pair denotes the DoD contributed by the feature type (benefit) if we allow it to have a certain number of occurrences (cost) in *all* results. After we compute the (benefit, cost) pairs for all feature types, since we consider a single size limit for all DFSs, we can use the same recurrence relation as in Figure 4 to compute the optimal number of occurrences of each feature type.

However, computing (benefit, cost) pairs for a feature type with respect to all results is much harder than doing so with respect to one result. In a single result, given a fixed number k of occurrences of a feature type, the features that can be output are fixed: we output the features one by one in the order of their dominance, thus we can only output the top-k most dominant features. On the other hand, given a fixed number of occurrences of a feature type in all results, there are many possibilities to assign these slots to all results, and the best slot assignment given a certain number of slots needs to be computed. We discuss two ways to compute the (benefit, cost) pairs for a feature type in the next two subsections: exact computation or heuristics computation.

The pseudocode of the framework of the feature type-oriented algorithm is shown in Algorithm 3. We use $\mathcal{L} = n \times L$ as the total size limit for all DFSs, where L is the size limit for each individual DFS. For each feature type in the results, we compute a set of (benefit, cost) pairs with respect to all results (line 4, which will be detailed in the next subsections), then use dynamic programming (procedure *DP*) to find the optimal number of occurrences of each feature type in all results. Note that since this approach considers a single size limit for all DFSs, it may generate some DFSs whose sizes are larger than L. In this case, we perform a postprocessing for these DFSs. For each DFS whose size exceeds L, we iteratively remove some features from it. Specifically, each time we pick one feature such that removing this feature will cause the smallest loss of DoD. We do so until it has exactly L features. Similarly, for each DFS whose size

ALGORITHM 3: Feature Type-Oriented DFS Construction

constructDFS (Query Results: $QR[1 \ldots n]$; Size Limit: L)

```
1: L = n × L
2: ftype[1...m] =all feature types in QR[1...n]
3: for i = 1 to m do
4:    benefit[1...L], cost[1...L] = computeBenefitCost_exact/heuristics( ftype[i], QR, L)
5: DFS[1...n] = DP(benefit, cost)
6: for each i = 1 to n do
7:    while DFS[i].size > L do
8:       F = a feature in DFS[i], such that removing F from DFS[i] causes the smallest loss of
         DoD
9:       remove F from DFS[i]
10:   while DFS[i].size < L do
11:      F = a feature in DFS[i], such that adding F to DFS[i] gives the largest increase of
         DoD
12:      if F = null then
13:         break
14:      add F to DFS[i]
```

DP ($benefit[1 \ldots n], cost[1 \ldots n]$)

```
1: m = number of feature types in QR
2: for l = 1 to L do
3:    compute s_{1,l} according to Figure 4
4:    Suppose s_{1,l} is maximized by outputting x features of type 1
5:    best_{1,l} = x
6: for k = 2 to m do
7:    for l = 1 to L do
8:       compute s_{k,l} according to Figure 4
9:       Suppose s_{k,l} is maximized by outputting x features of type k
10:      best_{k,l} = x
11: k = m
12: l = L
13: while k > 0 and l > 0 do
14:    output x features of type k in all DFSs
15:    k − −
16:    l− = x
```

is smaller than L, we iteratively add some features into it, until its size is L, or all features in the corresponding result have been added.

Since procedure DP computes a two-dimensional array with size $nL \times m$, the time complexity of procedure DP is $O(mnL)$. The complexity of $constructDFS$ depends on line 4, which is executed m times, and will be discussed later.

4.1. Exact Computation of (Benefit, Cost) Pairs

To compute the exact (benefit, cost) pairs, we compute the benefit for all possible costs, that is, from 1 to $\mathcal{L}$. For each possible cost c, we enumerate all possible ways to assign these c slots to the n results. The number of ways to assign c slots to n results equals $\binom{c+n-1}{n-1}$. To see this, note that assigning c slots to n results such that each result has ≥ 0 slots is equivalent to assigning $c+n$ slots to n results such that each result has ≥ 1 slots. The latter problem can be considered as: there are $c+n$ points on the x-axis, each representing a slot. We insert $n-1$ "baffles", such that each baffle is placed between two adjacent points, and no two baffles coincide. What is the number of ways to place all baffles? Note that for each placement of the $n-1$ baffles, we get an

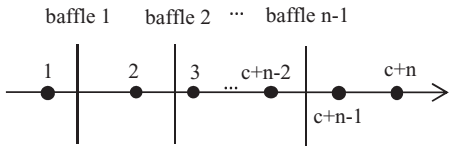

Fig. 6. Points and baffles.

ALGORITHM 4: Exact Computation of Benefit and Cost

computeBenefitCost_exact ($ftype, result[1 \ldots n], \mathcal{L}$)

1: **for** $c = 1$ to $\mathcal{L}$ **do**
2: $DoD =$ computeBenefit($ftype, result, c$)
3: $benefit[c] = DoD, cost[c] = c$

computeBenefit ($ftype, result[1 \ldots n], c$)

1: {consider $n + c - 1$ points on the x-axis and $n - 1$ baffles}
2: {A qualified baffle assignment: (1) each baffle is placed between two adjacent points; (2) no two baffles are placed between the same two points}
3: {The baffles are recorded in $baffle[1 \ldots n-1]$. $baffle[i] = j$ means the ith baffle is placed between points j and $j + 1$}
4: $bestBenefit = 0$
5: **for** each qualified baffle assignment $baffle[1 \ldots n-1]$ **do**
6: $size[1] = baffle[1]$
7: $size[n] = n - baffle[n-1]$
8: **for** $i = 2$ to $n - 1$ **do**
9: $size[i] = baffle[i] - baffle[i-1]$
10: **for** $i = 1$ to n **do**
11: assign $size[i]$ slots to $DFS[i]$
12: $benefit = 0$
13: **for** $i = 1$ to n **do**
14: **for** $j = i + 1$ to n **do**
15: $benefit+ = DOD_{ftype}(DFS[i], DFS[j])$
16: **if** $benefit >= bestBenefit$ **then**
17: $bestBenefit = benefit$
18: return $bestBenefit$

assignment of the slots: the number of points in between two baffles are the number of slots assigned to the corresponding result. For example, in Figure 6, results 1 and 2 are assigned 1 slot each; result n is assigned 2 slots. Therefore, the number of ways to assign the slots is equivalent to selecting $n - 1$ from $c + n - 1$, that is, $\binom{c+n-1}{n-1}$.

Therefore, the exact computation of (benefit, cost) pairs is to enumerate all $\binom{c+n-1}{n-1}$ ways of assigning c slots for each c ($1 \leq c \leq \mathcal{L}$). Note that this number is only exponential with respect to n (the number of results), while polynomial with respect to all other parameters. Since in reality a user will unlikely select a large number of results for comparison, this algorithm should work well practically.

The pseudocode of the exact computation of (benefit, cost) pairs is presented in Algorithm 4. All baffle assignment are enumerated for each cost $c(1 \leq c \leq \mathcal{L})$, and the assignment that has the largest benefit (DoD) is recorded for each c.

Now we analyze the complexity of Algorithm 4. For each cost c, we enumerate $\binom{c+n-1}{n-1} = O(c^n)$ baffle positions. For each baffle position, we need to check whether

Table I. An Illustration of the Heuristics Method for Computing (benefit, cost) Pairs

	D_1	D_2	DoD
$OptAsgnmt[0]$	empty	empty	0
$OptAsgnmt[1]$	camera:brand:Canon 52%	empty	0
$OptAsgnmt[2]$	camera:brand:Canon 52%	camera:brand:Canon 53%	1%
$OptAsgnmt[3]$	camera:brand:Canon 52% camera:brand:Sony 25%	camera:brand:Canon 53%	1%
$OptAsgnmt[4]$	camera:brand:Canon 52% camera:brand:Sony 25%	camera:brand:Canon 53% camera:brand:HP 47%	50%
$OptAsgnmt[5]$	camera:brand:Canon 52% camera:brand:Sony 25% camera:brand:Nikon 13%	camera:brand:Canon 53% camera:brand:HP 47%	76%

each pair of DFSs are differentiable. Let m denote the maximum number of features of each feature type. Since checking the differentiability of a feature type takes $O(m)$ time, checking the differentiability of all pairs of DFSs takes $O(n^2m)$ time. Therefore, the total complexity of Algorithm 4 is $O(\sum_{c=1}^{n\times L} c^n \times n^2m)$.

4.2. Heuristics Computation of (Benefit, Cost) Pairs

Although Algorithm 4 is only exponential with respect to n, it may still be inefficient in some situations. The main reason that may lead to its inefficiency is that for every possible cost, it needs to compute the optimal slot assignment from scratch, rather than incrementally. It has to do so because this problem does not have the optimal substructure property, in other words, to compute the optimal assignment of cost $c+1$, we are unable to reuse the optimal assignment of c or any other cost, as their optimal assignment may be totally different. When the number of results increases, both n and c increase (as each DFS has a fixed size limit), thus the processing time may increase very quickly.

Now we discuss an algorithm that heuristically computes the (benefit, cost) pairs with much better efficiency. The idea is to reuse the optimal assignment of lower costs when computing the optimal assignment of a higher cost, under the assumption that the optimal assignment of a higher cost likely does not differ too much from the optimal assignment of the lower cost. To do so, we use a vector $OptAsgnmt[1 \ldots \mathcal{L}]$, which records the optimal assignment of all costs we have computed so far. We compute $OptAsgnmt[i]$ by greedily adding a feature from $OptAsgnmt[i-1]$. For each i, suppose $OptAsgnmt[i]$ gives us a DoD of d_i, then we record a (benefit, cost) pair (d_i, i).

We start from processing $OptAsgnmt[0]$. $OptAsgnmt[0]$ is trivial: there is no (feature, percentage) pair in any result. To get $OptAsgnmt[i](1 \geq 1)$, we attempt to add a (feature, percentage) pair to one of the DFSs from the best assignment of cost $i-1$, that is, $OptAsgnmt[i-1]$. Suppose adding a (feature, percentage) pair to the jth result will give us the largest increase in DoD, then we add a (feature, percentage) pair to jth DFS from $OptAsgnmt[i-1]$ and consider it as $OptAsgnmt[i]$. Then, we go to the next cost, $i+1$, and process $OptAsgnmt[i+1]$.

Example 4.2. Consider the two results in Figure 1(a), and feature type camera:brand (other feature types are processed in the same way). Table I shows the construction of $OptAsgnmt$ for feature type camera:brand. Initially, $OptAsgnmt[0]$ outputs no features of this type in either DFS, thus DoD = 0. Now we try to add a (feature, percentage) pair in a DFS in $OptAsgnmt[0]$, for example, add $(canon, 52\%)$ in D_1, as shown in Table I. At this time, the DoD is still 0. We also attempt to add a (feature, percentage) pair to D_2 in $OptAsgnmt[0]$. Since this does not increase the DoD either, we do not update $OptAsgnmt[1]$ and its DoD. Then, from $OptAsgnmt[1]$, we continue to output one (feature, percentage) pair in a DFS. If we add a (feature, percentage) pair $(Sony, 25\%)$ in D_1 in $OptAsgnmt[1]$, since D_2 is empty, the DoD is still 0. On the other hand, if we add a (feature, percentage) pair $(Canon, 53\%)$ in D_2, we get

ALGORITHM 5: Heuristics Computation of Benefit and Cost

computeBenefitCost_heuristics ($ftype, result[1 \ldots n], \mathcal{L}$)

1: $OptAsgnmt[0] = \{0, 0, \ldots, 0\}$
2: $OptDoD[0] = 0$
3: **for** $i = 1$ to $\mathcal{L}$ **do**
4: $\quad OptDoD[i] = -1$
5: **for** $c = 1$ to $\mathcal{L} - 1$ **do**
6: $\quad$ **for** $i = 1$ to n **do**
7: $\quad\quad$ add a feature to $DFS[i]$ from $OptAsgnmt[c-1]$
8: $\quad\quad currDoD = DOD_{ftype}(DFS[1], \ldots, DFS[n])$
9: $\quad\quad$ **if** $currDoD \geq OptDoD[c]$ **then**
10: $\quad\quad\quad OptDoD[c] = currDoD$
11: $\quad\quad\quad OptAsgnmt[c] =$ current slot assignment
12: $\quad$ remove the newly added feature from $DFS[i]$
13: **for** $c = 1$ to $\mathcal{L}$ **do**
14: $\quad$ **if** $OptDoD[c] \geq 0$ **then**
15: $\quad\quad benefit[c] = OptDoD[c]$
16: $\quad\quad cost[c] = c$

a DoD of 1%. Therefore, in $OptAsgnmt[2]$, we assign one slot to each DFS. The process continues as shown in Table I.

The pseudocode of this algorithm is presented in Algorithm 5. We use $OptAsgnmt[c]$ to record the optimal slot assignment for cost c, and use $OptDoD[c]$ to record the benefit (i.e., DoD) achieved by $OptAsgnmt[c]$. We first initialize $OptAsgnmt[0]$ (line 1) and $OptDoD[c]$ for each c (line 4). Then start from $c = 1$, for each c, we compute the assignments of cost c based on $OptAsgnmt[c-1]$ (lines 5–12). We try to add a feature to each DFS, and compare the DoD obtained by all these n choices. Then, we select a DFS D such that adding a feature to D will give us the largest DoD. We add a feature to D and consider it as $OptAsgnmt[c]$ (lines 9–11). Finally, we record $\mathcal{L}$ (benefit, cost) pairs, that is, $(OptDoD[1], 1), \ldots, (OptDoD[\mathcal{L}], \mathcal{L})$ (lines 13–16).

Now we analyze the complexity of Algorithm 5. For each cost c, we try to add a feature to each of the n results, then see whether this result is differentiable with any other results. Recall that checking whether a feature type can differentiate two results takes $O(m)$ time, and there are in total $n\mathcal{L}$ different costs. Therefore, the total complexity is $O(n^2\mathcal{L}m)$.

5. EVALUATION

To verify the effectiveness and efficiency of our proposed approach, we implemented the CompareIt system and performed empirical evaluation from three perspectives: the usefulness of DFSs, the quality of DFSs, the time for generating the DFSs, and the scalability upon the number of query results and the DFS size limit.

5.1. Environments and Setup

The evaluations were performed on a desktop with Intel Core(TM) 2 Quad CPU 2.66 GHZ, 8GB memory, running Windows 7 Professional.

We used two datasets in our evaluation: a movie dataset and a retailer dataset. The movie dataset records information about movies, which was extracted from IMDB.[2] The retailer data is a synthetic dataset that records the information of apparel retailers and their stores. The schema of the retailer data is shown in Figure 7. The

[2]ftp://ftp.sunet.se/pub/tv+movies/imdb/.

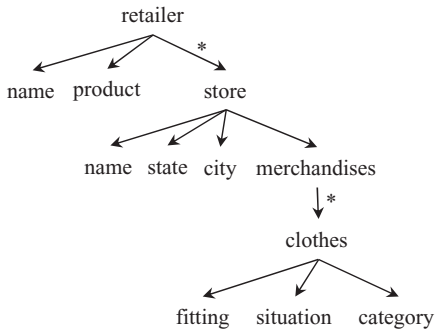

Fig. 7. Schema of the retailer data.

value of each node is randomly generated without functional dependencies. The test query set is shown in Table I. The query results of these queries are generated using one of the existing keyword search approaches [Liu and Chen 2007].

To verify the usefulness of DFSs, we performed a user study on Amazon Mechanical Turk, in which we compare differentiating results using DFSs with differentiating results using result snippets, and using results themselves. The detailed setting of the user study is presented in Section 5.2. For all other tests, for each query we generate DFSs for the first five results using five approaches: the single-swap optimal algorithm (Algorithm 1), the multi-swap optimal algorithm (Algorithm 2), the two feature type-oriented algorithms (Algorithms 4, denoted as FTO-Exact and 5, denoted as FTO-Heuristics), and a beam search algorithm. The beam search algorithm first generates a set of initial states, and in each iteration, takes the top-k states and generates successors states of the top-k states, then takes the top-k successor states and repeats the iteration, until the search is finished. For our problem, each initial state contains one DFS for each result which contains a single feature. Given a state s containing a set of DFSs, each successor state contains one DFS for each result such that each DFS contains one more feature than the corresponding DFS in s. Thus beam search takes L iterations for our problem where L is the DFS size limit. We set k as 20 in the experiment. The DFS size limit is set as 10.

5.2. Usefulness of DFS

In this test we performed a user study with 50 users on Amazon Mechanical Turk, aiming at verifying the usefulness of DFSs given existing techniques for constructing result comparison tables. We compare with two result comparison approaches: (1) showing the snippets of the results to the user, as developed in Liu et al. [2010]; (2) showing the results to the user, which is done by Web sites of banks such as chase.com (for comparing accounts, credit cards, etc.) and online retailers such as bestbuy.com (for comparing products). We made two modifications to approach (2). First, since a query result may have multiple occurrences of a feature (e.g., a store sells multiple DSLR cameras), we do not show users the entire result, but show them each distinct feature with its percentage, such as the infoboxes next to the results in Figure 1(a). Second, since many results are too big for the user to read, for each result, we only show the first few features (the number of features shown is the same as the DFS size limit) to the user (denoted as "result prefix").

For each of the 16 queries, we selected 2 or 3 results, and showed the users all distinct features together with their percentages in these results. Then, for each distinct feature type and each pair of results, the users are asked to select one of the following options. (A) This feature type has the same features in the two results.

Table II. Data and Query Sets

Film	
QM_1	director, UK
QM_2	Italy, movie
QM_3	Austria, romance
QM_4	director, Yinka Adebeyi, Anthony Ainley, Sean Adames
QM_5	2002, Sci Fi, director
QM_6	UK, comedy, Anita
QM_7	1960, France, comedy
QM_8	actor, 2004, drama
Camera	
QR_1	store
QR_2	retailer, pants, children
QR_3	men, category, outwear, retailer
QR_4	Texas, pants
QR_5	men, outwear, footwear, shirts
QR_6	men, women, children, outwear
QR_7	casual, shirts, store
QR_8	retailer, casual, shirts

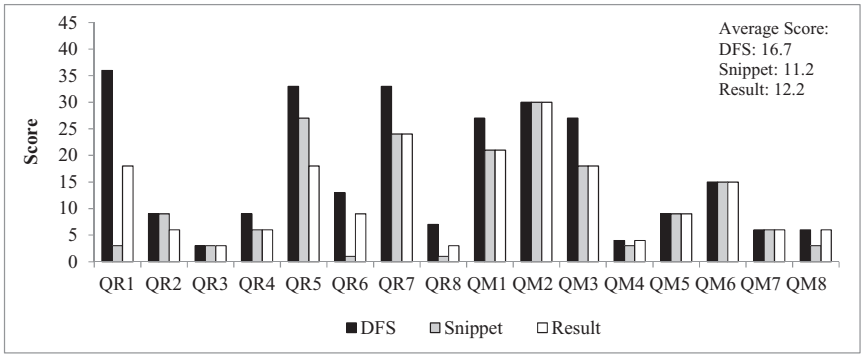

Fig. 8. The differentiation powers of DFSs, snippets, and result prefixes.

(B) This feature type slightly different features in the two results.
(C) This feature type significantly different features in the two results.

For each approach, if it shows the difference of a feature type which got option (A), (B), or (C) by most of the users, we give it a score of 0, 1, and 3, respectively.

The scores of each approach on each query is shown in Figure 8. For the DFS approach, we used the FTO-Heuristics method. As we can see, DFS shows significantly more differences than the other two methods. When the user compares the results by reading the result statistics itself, since a result may have many features, the users may often be able to read the first few features. However, the first few features may not show the differences of the results. For result snippets, as discussed in Section 1, although they output selected features in the results, the criteria of feature selection is based on whether a feature summarizes a single result, rather than whether a feature differentiates multiple results. Therefore, snippets are not designed for result differentiation and may not be helpful for users to compare the search results. Note that the snippet method often has a worse performance than result statistics. This is because the snippets of different results may have completely different feature types, which are not comparable. On the other hand, the result statistics method uses the first several features in each result, which usually have the same type and thus it has a better chance of differentiating results.

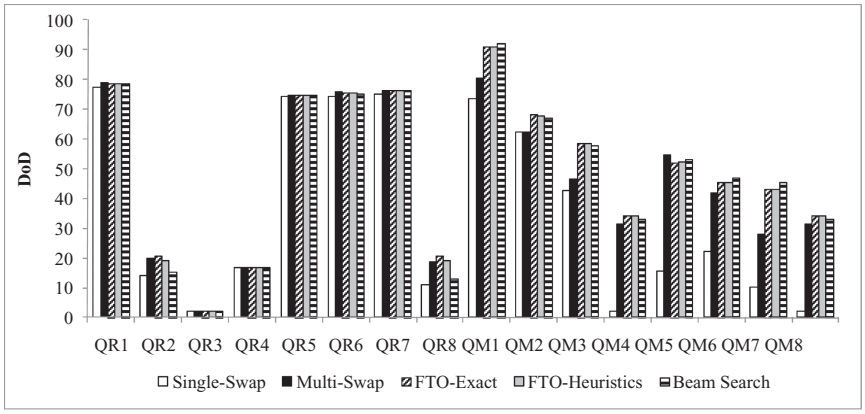

Fig. 9. Quality of DFSs.

5.3. Quality of DFS

For each query, the quality of the DFSs for its results is measured by their degree of differentiation (DoD) (Definition 2.2).

As we can see from Figure 9, the multi-swap optimal algorithm usually exhibits a superior quality to the single-swap algorithm. This is because the single-swap algorithm can only change one feature in a DFS at one time, and terminates if it cannot find such a change that can improve the DoD. For several queries such as QM_4, QM_5, and QM_8, the single-swap algorithm only achieves 5% to 30% of the DoD achieved by the multi-swap algorithm.

The feature type-oriented algorithms generally achieve a higher DoD compared with the swap-based algorithms. As discussed in Section 4, these algorithms evaluate the quality of each feature type and select the feature types in the order of their quality, thereby avoiding the problem of missing good feature types if they are not chosen initially. For queries such as QM_3 and QM_7, the feature type-oriented algorithms achieve a significantly higher DoD than the swap-based algorithms. Note that the performance of swap-based algorithms are closer to the feature type-oriented algorithms on the shopping data, since there are fewer distinct feature types in the shopping data compared with the movie data, thus the initialization of the swap-based algorithms will likely select all or most of the feature types into the DFSs, which results in a good quality of the swap-based algorithms.

The FTO-Exact approach achieves slightly higher DoD than the FTO-Heuristics approach on four queries (QR_2, QR_8, and QM_2), since it is able to compute the exact set of (benefit, cost) pairs by enumeration. However, for query QM_5, note that FTO-Exact has a slightly lower DoD than that of FTO-Heuristics. This is because both algorithms start with a single DFS size limit for all DFSs, rather than a size limit for each DFS. Thus some of the initial DFSs they generate may have a size larger than L, the size limit for each individual DFS in the problem. If this happens, both algorithms perform a postprocessing, which greedily removes some features from each DoD whose sizes exceed the limit, and greedily adds some features to each DoD whose sizes are smaller than the limit. Since this is a greedy procedure, the FTO-Heuristics approach may happen to get a better set of DFSs, which is the case for QM_5. However, for this query, FTO-Exact indeed achieves a better DoD for the initial set of DFSs generated (i.e., without postprocessing).

The beam search algorithm has a similar DoD as FTO-Exact and FTO-Heuristics for all queries. It has slightly higher DoD than FTO algorithms for one-third of the

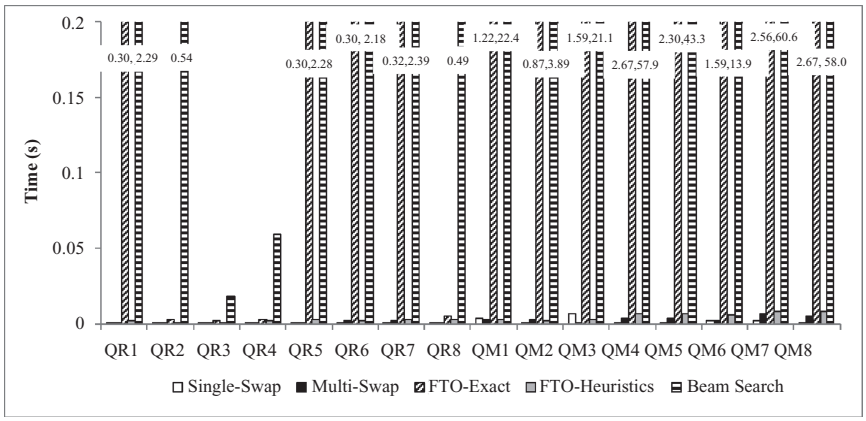

Fig. 10. Processing time of generating DFSs.

queries and has slightly lower DoD for another one-third of the queries, indicating that the beam search algorithm generally has a good quality. However, as to be shown in the efficiency test, the beam search algorithm is much slower since it needs to generate a large number of states.

5.4. Processing Time

To evaluate the efficiency of our algorithms, we measure the times that these approaches take to generate DFSs for the results of test queries in Table II, which is shown in Figure 10.

As we can see, the single-swap optimal algorithm generally achieves a better efficiency compared with the multi-swap optimal algorithm. The single-swap algorithm enumerates all possible changes to a single feature in a single DFS in each iteration, and has the iteration repeat until no further improvements can be made. The multi-swap algorithm checks possible changes of any number of features in a single DFS in an iteration, which involves computing a set of (benefit, cost) pairs and a dynamic programming process, and can be potentially more expensive. However, by exploiting dynamic programming, overlapping subproblems are identified in achieving the optimal solution, and thus repetitious computation is avoided, thus the processing time of the multi-swap algorithm is still very short. FTO-Exact has the lowest efficiency as its complexity is exponential to the number of results. On the other hand, FTO-Heuristics generally has a better efficiency compared with the multi-swap algorithm, as it directly evaluates each feature type and constructs the DFS accordingly, thereby avoiding iteratively modifing a DFS. The beam search algorithm is much slower even compared to the FTO-Exact algorithm. This is because the beam search algorithm needs to generate a large number of states to complete the search. In fact, even the number of initial states is huge: each initial state corresponds to a set of DFSs, each with one feature, thus the number of initial states is bounded by T^R, where T is the number of feature types and R is the number of results. The beam search algorithm is slower on movie data than on retailer data since the movie data has more feature types. On movie data, the average DFS generation time for the beam search algorithm is 35 seconds.

5.5. Scalability

We have tested the scalability of the single-swap optimal, multi-swap optimal, FTO-Exact and FTO-Heuristics algorithms over two parameters: *Number of Query*

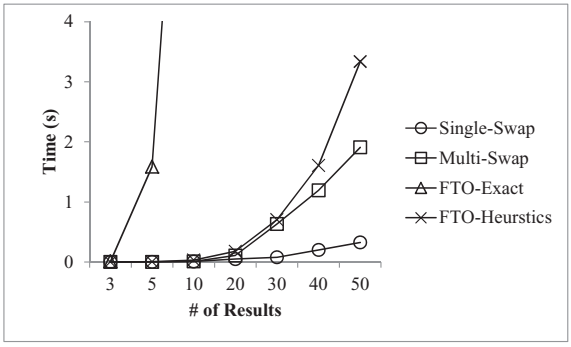

Fig. 11. Processing time with respect to the number of results.

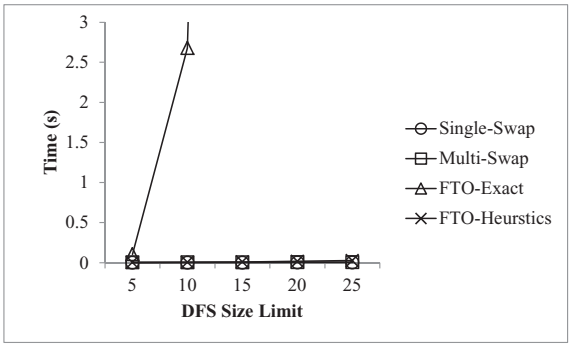

Fig. 12. Processing time with respect to DFS size limit.

Results and *DFS Size Limit*. Since beam search is extremely inefficient, we do not test its scalability.

Number of Query Results. In order to increase the number of query results, we varied the number of results generated for QM_6 from 3 to 50. The DFS size limit is set as 10. The performance of the algorithms is shown in Figure 11. As we can see, the processing times of all algorithms increase with more results. The single-swap algorithm has the best scalability, as its complexity is proportional to n^2L, where n is the number of results and L is the size limit. The processing time of FTO-Heuristics increases faster than the single-swap algorithm, since its complexity is actually proportional to $n^2\mathcal{L} = n^3L$. That being said, it still has a reasonable efficiency: in practice a user would rarely choose more than 50 results for comparison, while the DFS generation time of FTO-Heuristics for 50 results is less than 4 seconds. The multi-swap algorithm also increases faster than the single-swap optimal algorithm, since it changes multiple features of a DFS in each iteration. The processing time of FTO-Exact quickly deteriorates, as it is exponential to the number of results.

DFS Size Limit. In this test we evaluate the DoD and processing times of the four algorithms with respect to the increase of DFS size limit (i.e., the maximum number of features allowed in a DFS). We use the 5 results generated for QM_8. The efficiency of each approach is shown in Figure 12.

When the DFS size limit increases, the processing time of single-swap, multi-swap, and FTO-Heuristics algorithms slightly increases, but the processing times of all these approaches are close to 0. On the other hand, since the complexity of the

FTO-Exact approach is proportional to L^n, its processing time increases very quickly. As we can see, although the FTO-Exact approach has the best quality among all four approaches, it is practical only if both the number of results and the DFS size limit are small.

To summarize, the FTO-Heuristics algorithm works best among these algorithms. It can achieve almost the same quality as that of FTO-Exact in most cases, but is much more efficient and scalable. It has superior quality and comparable efficiency compared with the single-swap and multi-swap optimal algorithm.

6. RELATED WORK

This manuscript is an extension of our earlier conference article [Liu et al. 2009]. Significant extensions have been made in this article which include the following.

(1) We revised the presentation of DFSs. In Liu et al. [2009], a DFS consists of features; a feature may occur multiple times if it has a higher percentage than another feature. In this article, we introduced a more direct way of DFS presentation: the DFS consists of (feature, percentage) pairs, that is, each feature in the DFS is associated with its percentage within the feature type in the result. Correspondingly, in this article, we introduced a new measure of Degree of Differentiation (DoD) which replaced the original measure, as presented in Section 2.2.3. In Liu et al. [2009], a feature type either can differentiate or cannot differentiate two results. A feature type can differentiate two results if its features have a different orders or different ratios of percentage in the two DFSs. Now, instead of using either 0 or 1, we propose a measure of feature type DoD with finer granularity. Instead of using the order of the features of type T to determine whether T can differentiate two results, this new measure is based on the percentage of feature occurrences of type T. Given two DFSs, it considers feature type T as a vector in each DFS, whose components are features of type T, and the value of each component is the percentage of the corresponding features in the DFS. Then, we use a modified version of L1 distance (which considers the features of type T that are output in at least one DFS) to measure the distance of the two vectors, which is considered as the DoD of the feature type T.

(2) We modified the single-swap and multi-swap algorithms (Section 3) to incorporate the new measure of degree of difference.

(3) Despite the effectiveness and efficiency of the single-swap optimal and multi-swap optimal algorithms, we observe that the quality of the DFSs they generate has a considerable dependency on the random initialization. Unless every feature type in the optimal solution has sufficient occurrences in the initialization, these two algorithms cannot achieve optimality. We address this problem in Section 4 by proposing a new method which tackles the DFS generation problem from a new angle, that is, a feature type-oriented approach. This approach evaluates the quality of each feature type by computing a set of (benefit, cost) pairs for each feature type, then uses dynamic programming to select a set of feature types in the DFSs subject to the size limit. In this way, we avoid the dependency on the random initialization, thus good feature types will likely be deterministically included in the DFSs. We present two methods to compute the (benefit, cost) pairs for each feature type: an exact method and a heuristics method.

(4) In order to verify the usefulness of the DFSs generated by our approach, we performed a user study on Amazon Mechanical Turk with 50 users. We compare result differentiation using the DFSs we generate with using the result snippets and using the results themselves, as presented in Section 5.2.

(5) We empirically evaluated the feasibility of the feature-type-oriented approach compared with the ones presented in Liu et al. [2009], which verified its effectiveness and efficiency.

Next we review the literature in several categories.

Attribute Selection in Tables. There are works on relational databases that select important attributes from relations [Das et al. 2006; Miah et al. 2008]. Das et al. [2006] select a set of attributes from ranked results in order to "explain" the ranking function. Miah et al. [2008] select attributes of a tuple that can best "advertise" this tuple. Specially, it takes as input a relational database, a query log, and a new tuple, and computes a set of attributes that will rank this tuple high for as many queries in the query log as possible. On the other hand, our work selects features from tree-structured query results, with the goal of differentiating these trees in a small space.

Keyword Search on XML Data. Many different approaches have been proposed for identifying relevant keyword matches [Cohen et al. 2003; Guo et al. 2003; Hristidis et al. 2006; Kong et al. 2009; Li et al. 2007, 2008, 2004; Liu and Chen 2008; Sun et al. 2007; Xu and Papakonstantinou 2005], as well as relevant nonmatch nodes [Liu and Chen 2007] in developing XML keyword search engines.

Ranking Schemes. Ranking schemes have been studied for XML keyword search [Barg and Wong 2001; Bao et al. 2009; Cohen et al. 2003; Guo et al. 2003], considering factors like IR-style ranking (term frequency, document frequency, etc.), adopting PageRank to XML data, result size and match distance, etc.

Result Snippets. The problem of generating snippets for keyword searches is discussed in eXtract [Huang et al. 2008]. eXtract selects dominant features from each result to generate a small and informative snippet tree. Snippets are displayed with a link to each result, similar to a text search engine, to complement imperfect ranking schemes and enable users to quickly understand each query result.

Note that although result ranking, result snippet generation, as well as result differentiation are all helpful in keyword search, they are orthogonal problems and are useful in different aspects. Result ranking attempts to sort the query results in the order of expected relevance, so that the most relevant results can be easily discovered by the user from a large set of results. Due to the imperfectness of ranking, users still need to manually check some results to find the most desirable ones. Result snippets help users easily judge the relevance of a query result by providing an informative summary of the result. When there are multiple relevant query results (which is the case for informational queries, as discussed in Section 1), a user typically would like to compare and analyze a set of results. Since snippets aim at summarizing each individual result, they are generally unable to differentiate a set of results. Our proposed CompareIt system addresses this open problem. It automatically highlights the differences among a set of results concisely, and enables users to easily compare a set of results.

Faceted Search. Faceted search provides a classification of the data and enables effective data navigation. There are several approaches for automatically constructing faceted navigation interfaces given the set of query results, which aim at reducing the user's expected navigational cost in finding the relevant results [Li et al. 2010; Kashyap et al. 2010; Chakrabarti et al. 2004; Chen and Li 2007].

Faceted search techniques cannot be used for result differentiation for two reasons. First, the goals of facet search and result differentiation are different, and thus the methods are different. The goal of faceted navigation is to help users to find a relevant result as soon as possible by designing a navigation tree for result selection. The facets selected in the navigational tree thus may not maximize the difference among the results. For example, consider 10 results R1 – R10, and two facets f_1 and f_2. f_1 has two values: R1–R5 has one value, and R6 – R10 has another value. f_2 has different values in all 10 results. Suppose we are allowed to pick only one facet. Faceted navigation will pick f_1, because the navigation cost of f_1 is 7 (the user first looks at the 2 values of f_1, then picks a value and sees the corresponding 5 results), while the navigation

cost of f_2 is 11. However, for result differentiation purposes, we should pick f_2, as its value shows the differences among all results.

Second, existing works on faceted search only consider entities that have atomic facets, that is, each facet is a simple value, but cannot be, say, a distribution. On the other hand, we support a more general case, results/entities are compared by not only their own attributes (which are most likely to have atomic values), but also their relationships to other entities (which involve distributions). For example, in our running example, when we compare two camera stores, we consider not only store names, location (atomic attributes), but also the collection of cameras that they sell (distributions). Even though most likely every store has DSLR and compact cameras, the distribution of the number of DSLR cameras and the number of compact cameras can be largely different, which is considered in our approach.

7. CONCLUSIONS AND FUTURE WORK

Informational queries are pervasive in Web search, where a user would like to investigate, evaluate, compare, and synthesize multiple relevant results for information discovery and decision making. In this article we initiate a novel problem: how to design tools that automatically differentiate structured data search results, and thus relieve users from labor-intensive procedures of manually checking and comparing potentially large results. Towards this goal, we define Differentiation Feature Set (DFS) for each result and quantify the degree of differentiation. We identify three desiderata for good DFSs, that is, differentiability, validity, and small size. We then prove that the problem of constructing DFSs that are valid and can maximally differentiate a set of results within a size bound is an NP-hard problem. To provide practical solutions, we first propose two local optimality criteria, single-swap optimality and multi-swap optimality, and design efficient algorithms for achieving these criteria. Then we design an improved feature type-oriented method which evaluates the quality of feature types using two alternative methods: exact computation and heuristics computation. The feature type-oriented method achieves an improved DFS quality by avoiding dependency on the random initialization, which has a significant impact on DFS quality. Experiments verified the efficiency and effectiveness of the proposed approaches. Our proposed method is applicable to general query results which have features defined as (entity, attribute, value), and thus can be used to augment any existing XML keyword search engine.

As a new area, result differentiation has many open problems that call for research, which will be investigated in our future work. For instance, when selecting features from a DFS, we may further consider whether it is interesting to the user. The interest of a feature may be solicited from the user, or mined from query logs. Furthermore, there can be other functions that measure the degree of differentiation, such as the one that differentiates DFSs by features instead of feature types. Besides, other algorithms to tackle this NP-hard problem can also be useful.

REFERENCES

BAO, Z., LING, T. W., CHEN, B., AND LU, J. 2009. Effective XML keyword search with relevance oriented ranking. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

BARG, M. AND WONG, R. K. 2001. Structural proximity searching for large ccollections of semi-structured data. In *Proceedings of the CIKM Conference*. ACM Press, New York, 175–182.

BRODER, A. 2002. A taxonomy of web search. *ACM SIGIR Forum 36,* 2, 3–10.

CHAKRABARTI, K., CHAUDHURI, S., AND WON HWANG, S. 2004. Automatic categorization of query results. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. 755–766.

CHEN, Z. AND LI, T. 2007. Addressing diverse user preferences in SQL-query-result navigation. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. 641–652.

COHEN, S., MAMOU, J., KANZA, Y., AND SAGIV, Y. 2003. XSEarch: A semantic search engine for XML. http:www.vldb.org/conf/2003/papers/S.3P.2.pdf.

DAS, G., HRISTIDIS, V., KAPOOR, N., AND SUDARSHAN, S. 2006. Ordering the attributes of query results. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. 395–406.

GUO, L., SHAO, F., BOTEV, C., AND SHANMUGASUNDARAM, J. 2003. XRANK: Ranked keyword search over XML documents. In *Proceedings of the ACM SIGMOD on Management of Data*.

HRISTIDIS, V., KOUDAS, N., PAPAKONSTANTINOU, Y., AND SRIVASTAVA, D. 2006. Keyword proximity search in XML trees. *IEEE Trans. Knowl. Data Engin. 18,* 4.

HUANG, Y., LIU, Z., AND CHEN, Y. 2008. Query biased snippet generation in XML search. In *Proceedings of the ACM SIGMOD on Management of Data*.

KASHYAP, A., HRISTIDIS, V., AND PETROPOULOS, M. 2010. FACeTOR: Cost-driven exploration of faceted query results. In *Proceedings of the CIKM Conference*.

KONG, L., GILLERON, R., AND LEMAY, A. 2009. Retrieving meaningful relaxed tightest fragments for xml keyword search. In *Proceedings of the ACM EDBT Conference*. 815–826.

KULLBACK, S. 1987. The Kullback-Leibler distance. In *The American Statistician*.

LI, C., YAN, N., ROY, S. B., LISHAM, L., AND DAS, G. 2010. Facetedpedia: Dynamic generation of query-dependent faceted interfaces for wikipedia. In *Proceedings of the International Conference on World Wide Web (WWW)*. 651–660.

LI, G., FENG, J., WANG, J., AND ZHOU, L. 2007. Effective keyword search for valuable LCAs over XML documents. In *Proceedings of the CIKM Conference*.

LI, G., OOI, B. C., FENG, J., WANG, J., AND ZHOU, L. 2008. EASE: Efficient and adaptive keyword search on unstructured, semi-structured and structured data. In *Proceedings of the ACM SIGMOD on Management of Data*.

LI, Y., YU, C., AND JAGADISH, H. V. 2004. Schema-Free XQuery. In *Proceedings of the International Conference on Very Large Database (VLDB)*.

LIU, Z. AND CHEN, Y. 2007. Identifying meaningful return information for XML keyword search. In *Proceedings of the ACM SIGMOD on Management of Data*.

LIU, Z. AND CHEN, Y. 2008. Reasoning and identifying relevant matches for XML keyword search. In *Proceedings of the International Conference on Very Large Database (VLDB)*.

LIU, Z., HUANG, Y., AND CHEN, Y. 2010. Improving XML search by generating and utilizing informative result snippets. *ACM Trans. Datab. Syst. 35,* 3.

LIU, Z., SUN, P., AND CHEN, Y. 2009. Structured search result differentiation. *Proc. VLDB 2,* 1, 313–324.

MIAH, M., DAS, G., HRISTIDIS, V., AND MANNILA, H. 2008. Standing out in a crowd: Selecting attributes for maximum visibility. In *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 356–365.

RUBNER, Y., TOMASI, C., AND GUIBAS, L. J. 1998. A metric for distributions with applications to image databases. In *Proceedings of the International Conference on Computer Vision (ICCV)*. 59–66.

SUN, C., CHAN, C.-Y., AND GOENKA, A. 2007. Multiway SLCA-based keyword search in XML data. In *Proceedings of the International Conference on Data Engineering (WWW)*.

XU, Y. AND PAPAKONSTANTINOU, Y. 2005. Efficient keyword search for smallest LCAs in XML databases. In *Proceedings of the ACM SIGMOD on Management of Data*.

Received April 2010; revised May 2011; accepted October 2011