Structured Search Result Differentiation

Ziyang Liu
Arizona State University
ziyang.liu@asu.edu

Peng Sun
Arizona State University
peng.sun@asu.edu

Yi Chen
Arizona State University
yi@asu.edu

ABSTRACT

Studies show that about 50% of web search is for *information exploration* purpose, 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 potentially large results in an exploratory search. Such a process is time consuming, labor intensive and error prone. With meta information embedded, keyword search on structured data provides the potential for automating or semi-automating the comparison of multiple results.

In this paper we present an approach for differentiating search results on structured data. 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 to be NP-hard. We propose two local optimality conditions, namely single-swap and multi-swap. Efficient algorithms are designed to achieve local optimality. To show the applicability of our approach, we implemented a system XRed for XML result differentiation. Our empirical evaluation verifies the effectiveness and efficiency of the proposed approach.

1. INTRODUCTION

Studies show that about 50% of keyword searches on the web are for *information exploration* purposes, and inherently have multiple relevant results [3]. 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 website. 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.

For example, consider a customer who is looking for *stores* that sell *cameras* in *Phoenix* and issues a keyword query "*Phoenix, camera, store*". There are many results returned, where the fragments of two results by searching an XML repository 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.

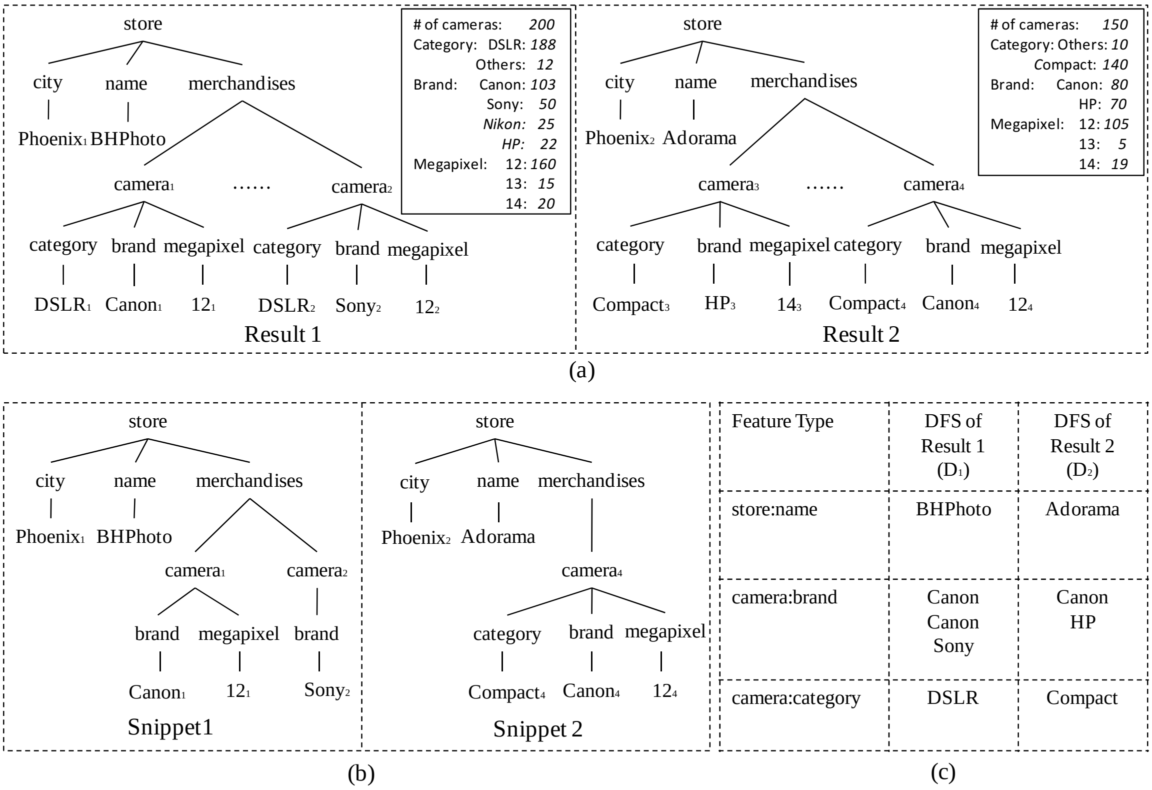
To help users analyze search results, the websites of many banks and online shopping companies, such as Citibank, Best Buy, etc., provide comparison tools for customers to compare specific products based on a set of pre-defined metrics, and have achieved big success. However, in these websites, only pre-defined types of objects (rather than arbitrary search results) can be compared, and the comparison metrics are pre-defined and static. 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 [11], a system for generating result snippets for XML keyword search, 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 DSLR* cameras, with roughly twice as many Canon cameras as Sony cameras; while the store in result 2 mainly sells *Canon* and *HP 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 *category* information about the store, is missing in its snippet due to space limitation. 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 com-

VLDB '09, August 24-28, 2009, Lyon, France



Feature Type	DFS of Result 1 (D_1)	DFS of Result 2 (D_2)
store:name	BHPhoto	Adorama
camera:brand	Canon Canon Sony	Canon HP
camera:category	DSLR	Compact

Figure 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)

parison 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 meta information of results presents a potential to enable result comparison. For example, directly generating a "comparison table" of an apple and an orange based on two textual paragraphs of general 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 paper, we propose techniques for comparison and differentiation of structured search results. The algorithm takes as input a set of structured results, each with a set of features (which will be defined in Section 2), and outputs a Differentiation Feature Set (DFS) for each result to highlight their differences within a size bound. To show the applicability of the proposed techniques, we implemented a system XRed (XML Result Differentiation), which differentiates search results on XML data. Sample DFSs for the query results in Figure 1(a) are shown in Figure 1(c). The contributions of this work include:

- We initiate the problem of differentiating structured search results, which is critical for diverse application domains, such as online shopping, employee hiring, job/ institution hunting, etc.
- We identify three desiderata of selecting Differentiation Feature Set (DFS) from search 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 selecting DFSs: single-swap optimality and multi-swap optimality, and developed efficient algorithms to achieve these criteria.
- We have developed the XRed system, whose effectiveness and efficiency is verified through extensive empirical evaluation.
- Our approach can be used to augment existing search engines for structured data to provide the functionality of helping users easily compare search results.

The rest of the paper 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 reports results of empirical evaluations. Section 5 discusses related works and Section 6 concludes the paper.

2. PROBLEM DEFINITION: CONSTRUCTING DIFFERENTIATION FEATURE SETS

In this section we first review some background on data models and features. Then we discuss three desiderata for *Differentiation Feature Set (DFS)*: limited size (Section 2.1), reasonable summary (Section 2.2), and maximal differentiation (Section 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 later 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 2.4).

We define a feature in a query result as a triplet (entity, attribute, value), and a feature type as an (entity, attribute) pair. Entities and attributes in relational databases are defined in the Entity-Relationship model. Entities and attributes in XML data can be inferred using the heuristics proposed in [18]. Specifically, an *entity* is inferred as a *-node in the DTD, i.e., the node that have some siblings with the same label, such as *camera* in Figure 1(a). An *attribute* is inferred as a non-entity node with a single leaf child, such as *category*. The child of an attribute node is called the value of the attribute. The remaining nodes are *connection nodes*. Other heuristics for identifying entities and attributes can also be used.

As an example, *(camera, category, DSLR)* in Figure 1 is a feature, whose type is *(camera, category)*.

2.1 Being Small

To help users compare search results, we would choose a subset of features from each result that maximizes the differences of this result and others, named as *Differentiation Feature Set (DFS)* in this paper.

To enable users quickly differentiate query results, the first desideratum of *Differentiation Feature Set (DFS)* is: *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**, i.e.,** $|D| \leq L$. ■

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 do not reflect the actual differences between the corresponding query results.

Example 2.1: Consider again the two results of query "*Phoenix, camera, store*" in Figure 1(a). Both results mainly sell *Canon* cameras. The store in result 1 also sells a couple of *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, the difference is meaningless as they give users the impression that store 1 differs from store 2 by mainly selling HP cameras instead of Canon cameras, which is untrue. 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 sells *Canon* and *HP*, it is undesirable to have 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 distributions of features of the same type in a DFS should roughly reflect their distribution in the data (up to the size limit). ■

As we can see from Example 2.1, a *valid* DFS should be a reasonable summary of the result, so that the comparison/differentiation based on the DFSs is meaningful and correct with respect to their results. There are two aspects of a reasonable summary of a result: important features should be output first, and the distributions of feature occurrences should be preserved. Hence we have the following desideratum.

Desideratum 2: [Validity] A DFS D should be valid wrt a result R and a threshold p (the maximum number of occurrences of a feature in D), denoted as $valid(D, R, p)$, defined by the following rules.

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); and the feature order of a DFS should be consistent with that of the result.
2. **Distribution Preserved:** For any two features of the same type in R, if they are both included in D, then the ratio of their number of occurrences in D must be as close to their ratio in R as possible, subject to the upper bound p.

■

To achieve "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 paper.

To achieve "Distribution Preserved", we first calculate the number of occurrences of each feature in the DFS which is proportional to their occurrences in the result subject to p. If this number is not an integer, we round it to the nearest integer when space allows (otherwise take the floor).

Example 2.2: In the 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 according to

Dominance Ordered, a feature *Nikon* can be included in the DFS only if both features preceding it, *Canon* and *Sony*, are already included.

According to *Distribution Preserved*, query result 1 has 103 cameras of brand *Canon*, and 50 cameras of brand *Sony*. If the maximum number of occurrences of a feature p is 3, then in the DFS, we have $round((103/(103+50)) \times 3) = 2$ *Canon*s, and $round((50/(103+50)) \times 3) = 1$ *Sony*. ■

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 most challenging requirement for a good DFS: *differentiability*, i.e., 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. Let us look at some examples before presenting the formal definition.

Example 2.3: Suppose all the features in each snippet in Figure 1(b) compose a DFS for the corresponding result, denoted by S_1 and S_2. Then we have:
S_1={*store:city:Phoenix, store:name:BHPhoto, camera:brand:Canon, camera:megapixel:12, camera:brand:Sony*}
S_2={*store:city:Phoenix, store:name:Adorama, camera:brand:Canon, camera:megapixel:12,camera:category:Compact*}.

Several observations can be made. First of all, features of different types are not comparable, e.g., we are not able to compare *camera:brand:Canon* in S_1 with *camera:category:Compact* in S_2.

Second, the two results can not be differentiated by features of type *camera:megapixel*, since both of them have exactly the same feature *12* on this type. So does the feature *store:city*.

Third, notice that S_2 has *camera:category:Compact*, but S_1 does not have features of type *camera:category*. Thus the two results cannot be differentiated by this feature type, as the user does not know whether the store in result 1 sells many compact cameras or not. It could be the case that result 1 sells more cameras that are compact than cameras of any other categories. In that case, if we add more features to S_1, then the first feature of type *camera:category* will be *Compact*, thus making S_1 and S_2 indifferentiable by this feature type. This is analogous to "null" values in databases: the absence of a value only means "unknown", but does not necessarily mean that the value is not what we are looking for.

For the same reason, the two results can not be differentiated by features of type *camera:brand*. Besides *Canon* cameras, S_1 additionally has *Sony*, but S_2 does not have any more features of this type. If more space is allowed, the next feature of type *camera:brand* outputted by S_2 may also be *Sony*. Thus these two results can not be differentiated by this feature type. ■

As we can see, two DFSs are considered as *identical* with respect to a feature type, if their features of this type have exactly the same dominance order and occurrence distribution. Furthermore, they are *indifferentiable* if adding more features of this type to one or both DFSs can make these two DFSs identical.

Now we formally define the differentiability of two DFSs with respect to a feature type as the following.

Definition 2.1: Given DFSs D_1 and D_2 of two results, for a feature type t, let $N_1 = |F(t(D_1))|$ and $N_2 = |F(t(D_2))|$, where $F(t(D))$ denotes the set of distinct features of type t in DFS D. Let $N[F(t(D))]$ be the first N distinct features of type t in D. D_1 and D_2 are *differentiable on feature type t*, or they *can be differentiated by t*, if and only if one of the following three conditions hold:

1. $N_1 \leq N_2$, and $\exists f \in F(t(D_1))$, such that $f \notin N_1[F(t(D_2))]$.
2. symmetrically, $N_2 \leq N_1$, and $\exists f \in F(t(D_2))$, such that $f \notin N_2[F(t(D_1))]$.
3. $\exists f_1, f_2 \in F(t(D_1)) \cap F(t(D_2))$, such that $n_1 : n_2 \neq n_3 : n_4$, where $n_1 = |\{f_1 | f_1 \in F(t(D_1))\}|$, $n_2 = |\{f_2 | f_2 \in F(t(D_1))\}|$, $n_3 = |\{f_1 | f_1 \in F(t(D_2))\}|$ and $n_4 = |\{f_2 | f_2 \in F(t(D_2))\}|$.

■

The first two conditions specify that for a feature type t, two DFSs differ in the order of feature dominance. In particular, condition (1) indicates that there is at least one different feature in the top-N_1 distinct features of type t in D_1 and D_2, i.e., D_1 has at least one feature f in the top-N_1 features that is not in the top-N_1 features of D_2. The third condition states that for a feature type t, two DFSs differ on the occurrence distribution of the corresponding features.

Example 2.4: As we can see, according to Definition 2.1, DFSs S_1 and S_2 in Example 2.3 are not differentiable with respect to feature type *store:city*, *camera:brand*, *camera:megapixel* or *camera:category*.

On the other hand, consider the two DFS D_1 and D_2 in Figure 1(c). Take feature type *camera:category* for example. According to Definition 2.1, $N_1 = N_2 = 1$ and the features in D_1 and D_2 are different. Therefore, for feature type *camera:category*, D_1 and D_2 satisfy conditions 1 and 2, and can be differentiated.

Similarly, for feature type *camera:brand*, D_1 and D_2 also satisfy conditions 1 and 2. Assuming that we use feature *HP* to replace *Sony* in D_1, then they are still differentiable by satisfying condition 3: the ratios between the number of occurrences of *Canon* and that of *HP* are about 2:1 in D_1 and 1:1 in D_2, respectively. ■

Degree of Differentiation between Two DFSs. Having defined the differentiability of DFSs with respect to a feature type, now we quantitatively define the *degree of differentiation* of two DFSs. Intuitively, given two results, the more feature types that can differentiate them, the more differences are indicated.

Definition 2.2: The *degree of differentiation* (DoD) of two DFSs is defined as the number of feature types on which the DFSs are differentiable. ■

Example 2.5: The two DFSs in Figure 1(c), for example, have a DoD of 3, as they can be differentiated on all three feature types *store:name*, *camera:brand* and *camera:category*. ■

Degree of Differentiation of a Set of DFSs. When

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

the user chooses to differentiate a set of results, the DFS of each result should be chosen to maximize the total degree of differentiation of every two results. Therefore, we have the following desideratum 3 for differentiation feature sets:

Desideratum 3: [Differentiability] Given a set of results $R_1, R_2, \cdots, R_n$, their DFSs, $D_1, D_2, \cdots, D_n$, should maximize the *total degree of differentiation* computed as the sum of the degree of differentiation of every pair of DFSs:

$$DoD(D_1, D_2, \cdots, D_n) = \sum_{1 \le i \le n} \sum_{i < j \le n} DoD(D_i, D_j)$$

■

Example 2.6: Suppose another result of the sample query "*Phoenix, camera, store*" has a DFS D_3 ={*store:name: porter's*, *camera:brand:Canon*, *camera:category:Compact*, *camera:category:DSLR* }. Comparing D_1 and D_3, since they are differentiable on feature types *store:name* and *camera:category*, we have $DoD(D_1, D_3)$ =2. D_2 and D_3 are differentiable only on feature type *store:name*, and thus $DoD(D_2, D_3) = 1$. Since $DoD(D_1, D_2) = 3$, the DoD of these three DFSs is 6. ■

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

2.4 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 2.1 - 2.3, given a set of results, their DFSs should maximize the DoD, i.e., 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* $(R_1, R_2, \cdots, R_n, m, L, p)$ is the following: given n search results $R_1, R_2, \cdots, R_n$, each with no more than m feature types, compute a DFS D_i for each result R_i, such that:

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

■

Theorem 2.7: 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 $R_1, R_2, \cdots, R_n$, 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 $DoD(D_1, D_2, \cdots, D_n) \ge S$?

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, i.e., $C = \{C_1, C_2, \cdots, C_l\}$, $|C| = l$, $C_i \subseteq X$ and $|C_i| = 3$.

The X3C problem is whether we can find an exact cover of X in C, i.e., 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. Let the DFS size limit L be 1, i.e., 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 $DoD(R_1, \cdots, R_{3q}) \ge 3q$?

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 3 to the DoD, thus the total DoD is $3q$.

If we can find a set of DFSs such that their DoD is $3q$, 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 1 to the DoD, making the total DoD impossible to reach $3q$.

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 $3q$. 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 $R_1, R_2, \cdots, R_n$ if, by changing or adding one feature in a DFS D_i of R_i, $1 \le i \le n$, while keeping $valid(D_i, R_i, p)$ and $|D_i| \le L$, their degree of differentiation, $DoD(D_1, D_2, \cdots, D_n)$, cannot increase. ■

Let us look at an example.

Example 3.1: The two DFSs in Figure 1(c) satisfy single-swap optimality, i.e., changing or adding any feature won't increase their DoD. For instance, if we change *camera:brand: Sony* in D_1 to *camera:megapixel:12*, then D_1 and D_2 are no longer differentiable on feature type *camera:brand*. 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_2. Thus their DoD decreases by 1. ∎

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:       $occ_f$ = the number of occurrences of $f$ in $DFS[i]$, calculated according to Distribution Preserved
 7:       add $f$ into $DFS[i]$
 8:       if $DFS[i].size > L$ then
 9:          remove $f$ from $DFS[i]$
10:       else
11:          $benefit$ = COMPUTEBENEFIT($DFS, i, t, f$, null, null)
12:          if $benefit > 0$ then
13:             goto line 3
14:          else
15:             remove $f$ from $DFS[i]$
16:       for each feature type $t'$ in $result[i]$ do
17:          $f$ = the last feature of type $t$ in $DFS[i]$
18:          $f'$ = the next feature of type $t'$ that is in $result[i]$ but not in $DFS[i]$
19:          $occ_{f'}$ = the number of occurrences of $f$ in $DFS[i]$, calculated according to Distribution Preserved
20:          change the occurrences of $f$ to $occ_{f'}$ occurrences of $f'$ in $DFS[i]$
21:          if $DFS[i].size > L$ then
22:             undo the change from $f$ to $f'$
23:          else
24:             $benefit$ = COMPUTEBENEFIT($DFS, i, t, f, t', f'$)
25:             if $benefit > 0$ then
26:                goto line 3
27:             else
28:                undo the change from $f$ to $f'$

COMPUTEBENEFIT ($DFS[n]$; $i$; Feature Type: $t$; Feature Value: $f$; Feature Type: $t'$; Feature Value: $f'$)
 1: $benefit = 0$
 2: for $j = 1$ to $n$ do
 3:    if $j = i$ then
 4:       continue
 5:    if DIFFERENTIABLE($DFS[i], DFS[j], t'$) = true then
 6:       remove $f'$ from $DFS[i]$
 7:       if DIFFERENTIABLE($DFS[i], DFS[j], t'$) = false then
 8:          $benefit - -$ {Since $DFS[i]$ and $DFS[j]$ are distinguishable on $t'$ with $f'$, but not without $f'$, we decrease the benefit by 1}
 9:       add $f'$ into $DFS[i]$
10:    if DIFFERENTIABLE($DFS[i], DFS[j], t$) = false then
11:       add $f$ to $DFS[i]$
12:       if DIFFERENTIABLE($DFS[i], DFS[j], t$) = true then
13:          $benefit + +$
14:       remove $f$ from $DFS[i]$
15: return $benefit$

DIFFERENTIABLE (DFS: $D_1$; DFS: $D_2$; Feature Type: $t$)
 1: {This can be checked according to Definition 2.1. The pseudo code is omitted.}
```

Single-swap optimality can be achieved by a polynomial-time algorithm: enumeration. The pseudo code 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 3-28). 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 4-15) or switching an existing feature of type t to a new feature of type t' that is currently not in DFS (lines 16-28).

3. *Updating and Iteration.* If such a DFS is found, then we make the update and restart the iteration in step 2 (lines 13 and 26).

4. *Termination.* If there is no DFS that can be changed to further improve the DoD, then we terminate and output the DFSs.

# of cameras:	*120*
Category:	Compact: *100*; Others: *20*
Brand:	Nikon: *70*; Kodak: *35;* Others: *15*
Megapixel:	10: *80*; 11: *25*; 12: *15*

(a) Statistics Information of Result 3

iteration	D_1	D_2	D_3
0	store: name: BHPhoto camera: brand: Canon camera: megapixel: 12	store: name: Adorama camera: brand: Canon camera: brand: HP	store: name: Porter's camera: brand: Nikon camera: category: Compact
1	store: name: BHPhoto camera: brand: Canon camera: megapixel: 12 *camera: category: DSLR*	same as above	same as above
2 (failed)	store: name: BHPhoto camera: brand: Canon *camera: brand: Canon* *camera: brand: Sony* camera: megapixel: 12 camera: category: DSLR	same as above	same as above
3	same as iteration 1	store: name: Adorama camera: brand: Canon camera: brand: HP *camera: category: Compact*	same as above
4	same as above	same as above	store: name: Porter's camera: brand: Nikon camera: category: Compact *camera: megapixel: 10*
5	same as above	store: name: Adorama camera: brand: Canon camera: brand: HP camera: category: Compact *camera: megapixel: 12*	same as above

(b) Iterations Performed by Algorithm 1

Figure 2: Running Example of Algorithm 1

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. Recall that a feature type can have multiple features included in a DFS. According to the *Dominance Ordered* rule, 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. As we can see, their initial *DoD* 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 make D_1 and D_3 differentiable on this type, and increase the DoD to 6. ■

According to the *Distribution Preserved* rule, the addition of a new feature of type t to a DFS may result in the addition of multiple occurrences of some existing features of t. Symmetrically, the removal of a feature of type t' from a DFS will result in all occurrences of this feature to be removed. Meanwhile, the number of occurrences of other features of type t' may also need to be divided by their common divisor. As we can see, changing or adding one feature may result in an increase of the DFS size. Thus it is necessary to check the size of the updated DFS. Any change that violates the constraint of size limit will be denied.

Example 3.3: Continuing our example, in iteration 2, the algorithm attempts to add feature *Sony* of type *camera:brand* into D_1. According to Figure 1(a), the ratio of *Canon* and *Sony* of this feature type is roughly 2:1 in query result 1. Thus, to add *Sony*, an extra *Canon* must be added together according to rule *Distribution Preserved*. Although such addition can increase $DoD(D_1, D_2)$, it will make the size of D_1 exceed the size limit 5, hence the attempted addition is rolled back. ■

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 3.4: Continuing Example 3.3, since the previous attempt fails, Algorithm 1 tries to add feature *Compact*, the most dominant feature of type *camera:category*, to D_2. This succeeds and $DoD(D_1, D_2)$ is increased by 1. Similarly, in iteration 3, *camera:megapixel:10* is added to D_3, making D_1 and D_3 differentiable and increasing *DoD* by 1. At this time, it becomes valuable to add *camera:megapixel:12* to D_2 in iteration 4. Adding such a feature before we add *camera:megapixel:10* to D_3 (iteration 3) does not increase DoD, but does increase $DoD(D_2, D_3)$ by 1 at this point. As we can 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. Now $DoD(D_1, D_2, D_3)$ becomes 9. ■

The iteration continues till 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 the removal of a feature f' of type t' may or may not decrease the *DoD*. The *DoD* is decreased by one if two DFSs originally can be differentiated by feature type t', but are no longer differentiable with respect to type t' after the removal of f' (lines 5-9). Similarly, we measure the possible increase to the DoD upon the addition of a feature (lines 10-14).

Now we analyze the complexity of the Algorithm 1. Recall that n is the number of query results, and m is 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* by checking whether $DFS[i]$ can be differentiated on type t and t' from each of the rest $(n-1)$ DFSs. To check whether two DFSs can be differentiated given a feature type t, we need to sort the features of type t in both DFSs, then scan them and determine the differentiability according to Definition 2.1. Therefore, it takes $O(L\log L)$ time, where L is the snippet size limit. Thus, COMPUTEBENEFIT takes $O(nL\log L)$, and each iteration takes at most $O(n^2m^2L\log L)$ time.
- In each iteration except the last one, the *DoD* of the DFSs at least increases by 1. The maximum possible *DoD* is bounded by $O(n^2m^2)$, since the degree of differentiation of every two DFSs is at most $O(m^2)$, and there are $O(n^2)$ pairs. This means we need at most $O(n^2m^2)$ 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 below.

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

Example 3.5: 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). As we can see, $DoD(D_1', D_2) = 2$ cannot be improved by changing

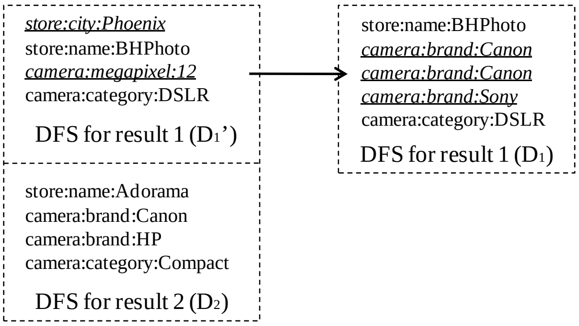


Figure 3: Single-Swap Optimality and Multi-Swap Optimality

or adding a single feature in either DFS. However, if we change features *store:city:Phoenix* and *camera:megapixel:12* into 2 *camera:brand:Canon*s and 1 *camera:brand:Sony*, then feature type *camera:brand* becomes differentiable in the two DFSs, and the *DoD* becomes 3. ■

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 the result size n, there can be up to 2^n 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), our algorithm directly generates a valid *multi-swap optimal* target DFS, given the others 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. As discussed in Section 3.1, for a DFS to be valid with respect to the corresponding query result, features of the same type have to be added to the DFS in the order of feature dominance, with distribution preserved subject to a size threshold. Given these constraints, there are still many choices of including the feature types to generate a valid 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 3.6: We use the query results in Figure 1(a) and Figure 2(a) to explain the benefits and costs of a feature type. For feature type *camera:brand*, we have
$D_2 = \{Canon, HP\}$, $D_3 = \{Nikon\}$,
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 $\{Canon, Sony, ...\}$, and the ratio of *Canon* and *Sony* is roughly 2:1.

(1) If we have $D_1 = \{Canon\}$, then cost=1, benefit=1. Note that D_1 is indifferentiable with D_2 at this point, but is differentiable with D_3, hence a benefit of 1.

(2) If we have $D_1 = \{Canon, Canon, Sony\}$, then cost=3, as 3 features are included, and benefit=2, since D_2 and D_1, as well as D_3 and D_1 are differentiable on this feature type.

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 above two choices as (1, 1) and (3, 2), respectively.

Also note that not every possible cost corresponds to a valid DFS. For instance, there is no valid DFS with benefit of 3 and cost of 2. ■

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:             if DIFFERENTIABLE($DFS[i], DFS[j], t$) then
 8:                 $DoD[i] + +$
 9: for $i = 1$ to $n$ do
10:     $DoD'$, $newDFS$ = CHECKDFS($QR[i], DFS, i, L$)
11:     if $DoD' > DoD[i]$ then
12:         $DFS[i] = newDFS$
13:         $DoD[i] = DoD'$
14:         goto line 9
CHECKDFS (Query Result: $QR$; DFSs: $DFS[n]$; $i$; Size Limit: $L$)
 1: $t$ = 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 $t$ 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: $k = t$
12: $l = L$
13: $newDFS = \emptyset$
14: while $k > 0$ and $l > 0$ do
15:     output $x$ features of type $k$ in $newDFS$
16:     $k - -$
17:     $l - = x$
18: $DoD' = 0$
19: for $j = 1$ to $n$ do
20:     $DoD' + =$ the degree of differentiation between $DFS[i]$ and $DFS[j]$
21: return $DoD'$, $newDFS$
```

Given the choices of generating valid DFSs discussed above, 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, i.e., $s_{k,l} = s_{k-1,l-c_1} + b_1$.

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}), $\cdots$ (b_{kp_k}, c_{kp_k}), and $1 \leq i \leq p_k$.

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

Figure 4: Recurrence Relation

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 2-5), then compute $s_{k,l}$ as discussed. Meanwhile, we record array *best*, which is used to re-produce the optimal DFS, $newDFS$ (lines 11-17). Finally, DoD' is calculated by comparing $newDFS$ with every other DFS (lines 18-21).

The entire algorithm for multi-swap optimality is presented in Algorithm 2. Similar as 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 4-8). In each iteration (lines 9-14), 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 by $newDFS$ with $DoD[i]$ updated (lines 11-14). Similar as 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 2-17), with complexity $O(m'L)$. Lines 18-20 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 4-8). Similar as Algorithm 1, the iteration in lines 9-14 is executed at most $O(n^2m^2)$ times. Therefore, the total complexity of Algorithm 2 is $O(n^2m^2L\log L(mn + m'))$.

As to be shown in Section 4, the algorithm is in fact quite efficient in practice, as the number of iterations is generally far less than n^2m^2.

4. EVALUATION

To verify the effectiveness and efficiency of our proposed approach, we implemented the XReD system, which differentiates search results on XML data, and performed empirical evaluation from three perspectives: the *quality* of DFSs, the *time* for generating the DFSs and the *scalability* upon the increase of query result size, number of query results, and the DFS size limit.

4.1 Environments and Setup

The evaluations were performed on a laptop with Intel Core(TM) 2 CPU 1.66GHZ, 2GB memory, running Windows XP Professional.

We used two data sets in our evaluation: a film data set and a camera retailer data set. The film data set records information about movies, which has the combined information of 7 XML documents.[2] Camera retailer is a synthetic data set that has a similar shape as the one in Figure 1. The value of each node is randomly generated. The test query set is shown in Table 1. The queries are chosen to represent a variety of query patterns, e.g., they include single-keyword and multi-keyword queries; queries with relatively small results (such as QF_6) and large results (such as QC_1); queries with relatively few results (such as QC_8) and with many results (such as QF_5, which has 52 results). The query results of these queries are generated using one of the existing keyword search approaches [18]. The size of a query result varies from 1KB to 9KB.

For each query, we generate DFSs for all the results using three approaches: the single-swap optimal algorithm (Algorithm 1), the multi-swap optimal algorithm (Algorithm 2), and a global optimal algorithm. The global optimal algorithm searches the entire solution space (with some pruning) to find the optimal DFS for each query result, which takes exponential time. Specifically, the pruning used in the global optimal algorithm is: if two DFSs are already differentiable by a feature type, then we do not further check other features of this type for this pair of DFSs. All three approaches initialize the DFS of a query result such that they are valid and do not exceed the size limit. For each query, we choose a DFS size limit equal to 10% of the number of feature types in all the results.

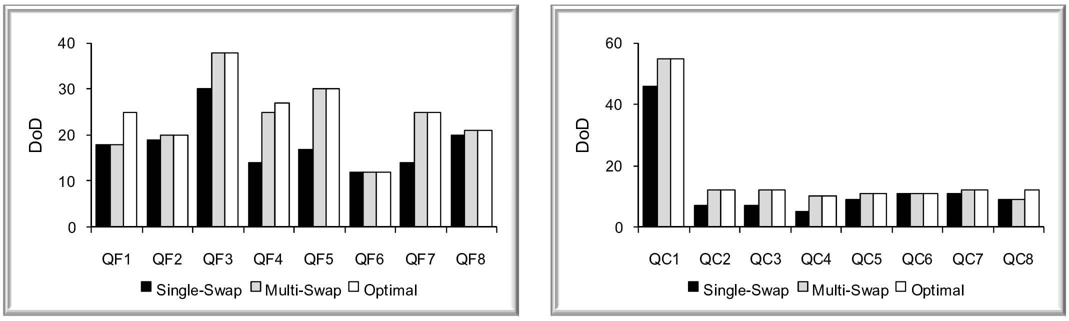
In the next subsections we report the evaluation results for DFS quality, DFS processing time and scalability, respectively.

4.2 Quality

For each query, the quality of the DFSs for its results is measured by their degree of differentiation (DoD). The qualities of the DFSs generated by the three approaches are shown in Figure 5.

As verified empirically, the single-swap optimal algorithm always exhibits an inferior quality to the multi-swap and global optimal algorithms. This is because the simple-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 QF_4,

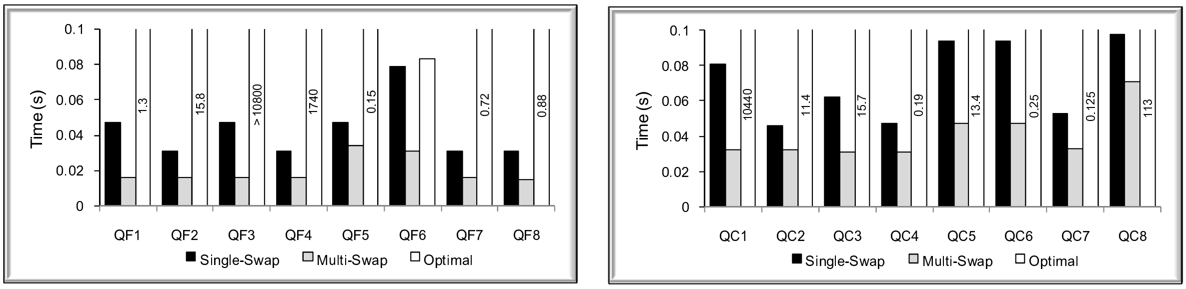
[2] http://infolab.stanford.edu/pub/movies



(a) Film Data Set

(b) Camera Retailer Data Set

Figure 5: Quality of DFSs Generated by Single-Swap, Multi-Swap and Global Optimal Algorithms



(a) Film Data Set

(b) Camera Retailer Data Set

For the processing times that exceed the scope of the vertical axis, we annotate the time within the corresponding bars.

Figure 6: DFS Processing Time by Single-Swap, Multi-Swap and Global Optimal Algorithms

Table 1: Data and Query Sets

Film	
QF_1	famous, director
QF_2	Italy, film
QF_3	1940, writers
QF_4	Hitchcock, studio, Paramount
QF_5	site, Russia, film
QF_6	Metro, studio, film
QF_7	Austria, films
QF_8	Epic, director, studio
Camera	
QC_1	store
QC_2	retailer, Sony, DSLR
QC_3	megapixel, 13, DSLR, retailer
QC_4	Texas, compact
QC_5	DSLR, Canon, Olympus, Nikon
QC_6	12, 14, 13, Kodak
QC_7	compact, Olympus, store
QC_8	retailer, 14, Kodak

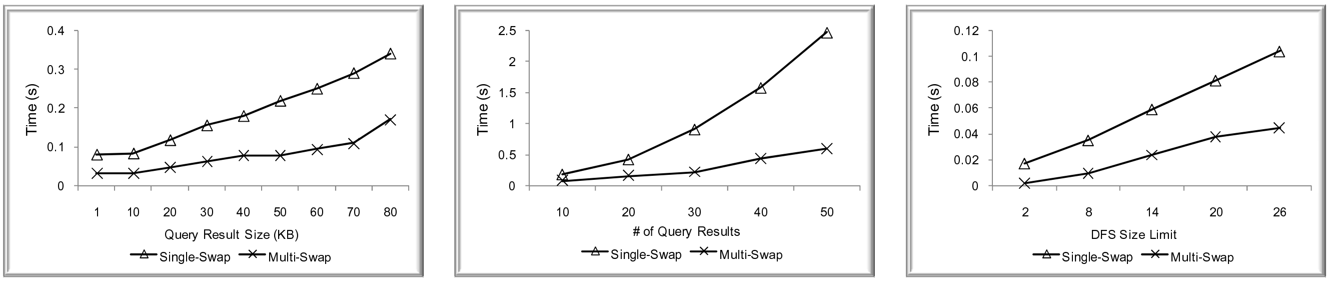
QF_7 and QC_4, the single-swap algorithm only achieves half of the *DoD* compared with those achieved by the other two approaches.

On the other hand, the multi-swap optimal algorithm achieves the same *DoD* as the global optimal algorithms for most of the queries, as it is capable of changing multiple features at a time in a DFS, hence produces the optimal DFS given the DFSs of the other results. However, in a few occasions it is slightly outperformed by the global algorithm such as QF_1, QF_4 and QC_8, i.e., the DFSs it generates are not necessarily global optimal. The reason is that since the multi-swap algorithm updates one DFS at a time, it may miss the opportunity to improve the *DoD* by updating multiple DFSs at the same time. For instance, consider QF_1, there is no feature of type "film:category" in any initial DFS. In the iterations of the multi-swap optimal algorithm, DFSs are updated one by one. Adding features of this type to a single DFS will not increase the *DoD*, since no other DFS contains this feature. Thus features of this type can not be introduced to the DFSs in the multi-swap algorithm. However, by introducing features of type "film:category" in the DFSs of multiple query results, the optimal algorithm can improve the *DoD* and achieve a better quality.

4.3 Processing Time

To evaluate the efficiency of our algorithms, we measure the times that all three approaches take to generate DFSs for the results of test queries in Table 1, which is shown in Figure 6.

As we can see, the multi-swap optimal algorithm not only achieves a better quality than the single-swap optimal algorithm, but generally has a better efficiency. The single-swap algorithm enumerates all possible changes to a single feature in a single DFS in each iteration, and has the iteration repeat till 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 can be potentially more expensive. However, by exploiting dynamic programming, overlapping subproblems are identified in achieving



(a) Size of Query Results

(b) Number of Query Results

(c) DFS Size Limit

Figure 7: Scalability

the optimal solution, and thus repetitious computation is avoided to gain a better performance.

The global algorithm, on the other hand, searches the entire solution space for the optimal set of DFSs. Although some pruning are performed, the huge size of the solution space makes it very inefficient. As we can see, for some queries, their DFSs cannot be generated in several hours, such as QF_3, QC_1. For most queries, it is more than 10 times slower than the other two algorithms.

4.4 Scalability

We have tested the scalability of the single-swap and multi-swap optimal algorithms on the camera data set over three parameters: *Size of Query Result*, *Number of Query Results* and *DFS Size Limit*. Since the performance of the global optimal algorithm is significantly worse than the other two approaches and deteriorates rapidly with these parameters, we do not include it in the scalability test. We use QC_1 for the scalability evaluation.

Size of Query Result. The scalability test with respect to query result size is shown in Figure 7(a). There are 10 query results, each of which is replicated between 10 and 80 times to make the size of the query result larger each time, ranging from 1KB to 80KB. The DFS size limit is fixed to be 10 features in a DFS. As we can see, the processing times of both single-swap and multi-swap optimal algorithms grow almost linearly; and the processing time of the multi-swap optimal algorithm grows slower than that of the single-swap optimal algorithm. For query results of 80KB, multi-swap algorithm requires less than 0.15 second for DFSs generation.

Number of Query Results. In order to increase the number of query results, we simulate query results whose features are randomly chosen from all the features in the 10 results of QC_1. These randomly generated results have similar size and similar features as the existing ones, and were treated, together with the original results, as the results of QC_1. The DFS size limit is 10 features in a DFS. The performance of the two algorithms is shown in Figure 7(b). As we can see, the processing time of both algorithms increases quadratically, which is reflected by the complexity of the two algorithms. The multi-swap optimal algorithm has consistently better efficiency than the single-swap algorithm. Note that in practice a user rarely can compare and analyze more than 20 results at the same time. The multi-swap optimal algorithm only spends 0.6 seconds in generating DFSs for 50 results and thus is practically efficient.

DFS Size Limit. In this test we evaluate the processing times of the two algorithms with respect to the increase of DFS size limit (i.e., the maximum number of features allowed in a DFS), while keeping the query result size to be 1KB, and the number of query results to be 10. The average number of features in a query result is 28, and the DFS size varies from 2 to 26. When the DFS size limit increases, both algorithms take more iterations to update the DFSs and improving the DoD, and thus require longer processing time. The result in Figure 7(c) suggests that the processing times of both algorithms increase linearly, while the multi-swap algorithm is much more efficient, with less than 0.05 second to generate DFSs for 10 results with DFS size limit of 26.

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

5. RELATED WORK

In this section, we review the literature in several categories.

Attribute Selection in Tables. There are works on relational databases that select important attributes from relations [5, 21]. [5] selects a set of attributes from ranked results in order to "explain" the ranking function. [21] 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 structured search results, with the goal of differentiating results in a small space.

Keyword Search and Result Ranking on Structured Data. Many approaches have been proposed for supporting keyword search on XML data [4, 8, 9, 14, 15, 16, 17, 19, 22, 24, 18], and keyword search on graphs / relational databases [10, 23, 1, 2, 12, 20, 6, 16, 13, 7]. Various ranking schemes have been proposed in these studies, including IR-style ranking (term frequency, document frequency, etc.), result size, page rank variants, etc..

Result Snippets. The problem of generating snippets for keyword search on XML data is discussed in eXtract [11]. 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 as a text search engine, to complement imperfect ranking schemes and enable the 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, the 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 XReD system addresses this open problem. It automatically highlights the differences among a set of results concisely, and enables the users to easily compare a set of results.

The XReD system can take the results generated by any of the existing XML keyword search engine 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 and XQuery) upon user request.

6. 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 paper we initiate a novel problem: how to design tools that automatically differentiate structured 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, i.e., 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 propose two local optimality criteria, single-swap optimality and multi-swap optimality. Then we design efficient algorithms for achieving these criteria. Our proposed techniques are applicable to general query results which have features defined as (entity, attribute, value). We also implemented the XReD system, which can be used to augment any existing XML keyword search engine, and whose efficiency and effectiveness has been verified through extensive experiments.

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. Furthermore, there can be other functions that measure the degree of differentiation. Approximation algorithms for DFS constructions are also in demand.

7. ACKNOWLEDGEMENT

This work was supported by NSF grant number IIS-0740129, and NSF CAREER award IIS-0845647.

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *Proceedings of ICDE*, pages 5–16, 2002.

[2] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, 2002.

[3] A. Broder. A Taxonomy of Web Search. *SIGIR*, 2002.

[4] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A semantic Search Engine for XML, 2003.

[5] G. Das, V. Hristidis, N. Kapoor, and S. Sudarshan. Ordering the attributes of query results. In *SIGMOD*, pages 395–406, 2006.

[6] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding Top-k Min-Cost Connected Trees in Databases. In *Proceedings of ICDE*, 2007.

[7] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD Conference*, pages 927–940, 2008.

[8] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proceedings of SIGMOD*, pages 16–27, 2003.

[9] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword Proximity Search in XML Trees. *IEEE Transactions on Knowledge and Data Engineering*, 18(4), 2006.

[10] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *Procs. VLDB*, 2002.

[11] Y. Huang, Z. Liu, and Y. Chen. Query Biased Snippet Generation in XML Search. In *SIGMOD*, 2008.

[12] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[13] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proceedings of PODS*, 2006.

[14] L. Kong, R. Gilleron, and A. Lemay. Retrieving Meaningful Relaxed Tightest Fragments for XML Keyword Search. In *Proceedings of EDBT*, 2009.

[15] G. Li, J. Feng, J. Wang, and L. Zhou. Effective Keyword Search for Valuable LCAs over XML Documents. In *Proceedings of CIKM*, 2007.

[16] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.

[17] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.

[18] Z. Liu and Y. Chen. Identifying Meaningful Return Information for XML Keyword Search. In *Proceedings of SIGMOD*, 2007.

[19] Z. Liu and Y. Chen. Reasoning and Identifying Relevant Matches for XML Keyword Search. In *Proceedings of VLDB*, 2008.

[20] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, 2007.

[21] M. Miah, G. Das, V. Hristidis, and H. Mannila. Standing out in a crowd: Selecting attributes for maximum visibility. In *ICDE*, 2008.

[22] C. Sun, C.-Y. Chan, and A. Goenka. Multiway SLCA-based Keyword Search in XML Data. In *Proceedings of WWW*, 2007.

[23] Vagelis Hristidis and Luis Gravano and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, 2003.

[24] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD*, 2005.