Dynamic Diversification of Continuous Data

Marina Drosou
Computer Science Department
University of Ioannina, Greece
mdrosou@cs.uoi.gr

Evaggelia Pitoura
Computer Science Department
University of Ioannina, Greece
pitoura@cs.uoi.gr

ABSTRACT

Result diversification has recently attracted considerable attention as a means of increasing user satisfaction in recommender systems, as well as in web and database search. In this paper, we focus on the problem of selecting the k-most diverse items from a result set. Whereas previous research has mainly considered the static version of the problem, in this paper, we exploit the dynamic case in which the result set changes over time, as for example, in the case of notification services. We define the CONTINUOUS k-DIVERSITY PROBLEM along with appropriate constraints that enforce continuity requirements on the diversified results. Our proposed approach is based on cover trees and supports dynamic item insertion and deletion. The diversification problem is in general NP-complete; we provide theoretical bounds that characterize the quality of our solution based on cover trees with respect to the optimal solution. Finally, we report experimental results concerning the efficiency and effectiveness of our approach on a variety of real and synthetic datasets.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process, Information filtering*

General Terms

Algorithms, Experimentation, Design, Performance

Keywords

Diversity, Continuous, Top-k

1. INTRODUCTION

The abundance of information available online creates the need for developing methods towards selecting and presenting to the user representative subsets. To this end, recently, result diversification has attracted considerable attention as a means of increasing user satisfaction. Result diversification takes many forms including selecting items so that their novelty, coverage, or content dissimilarity is maximized [11].

Previous approaches to result diversification can be roughly divided into those employing *greedy* and *interchange* heuristics for computing solutions. Greedy heuristics (e.g., [25, 15]) build a diverse set incrementally, selecting one item at a time so that some distance function is maximized, whereas interchange heuristics (e.g., [24, 19]) start from a random initial set and try to improve it. There are a couple of approaches that propose indexing to assist diversification. In [22], a Dewey-based tree is used for structural diversity based on attribute priorities and in [16], a spatial index is exploited to locate those nearest neighbors of an item that are the most distant to each other.

Despite the considerable recent interest in diversification, most previous research studies the *static* version of the problem, i.e., the available items out of which a diverse subset is selected do not change over time. Among the few attempts to address the dynamic case, in [10], we have experimented with greedy heuristics, while recently an interchange heuristic was introduced for incrementally constructing a diverse set for a stream of items [20].

In this paper, we propose an index-based approach to the *dynamic* diversification problem, where insertions and deletions of items are allowed and the diverse subset needs to be refreshed to reflect such updates.

We also consider the *continuous* version of the problem, where diversified sets are computed over streams of items. The motivation for this model emanates from many popular proactive delivery paradigms, such as news alerts, RSS feeds and notification services in social networks such as in Twitter [6]. In such applications, users specify their areas of interest and receive relevant notifications. To avoid overwhelming the users by forwarding to them all relevant items, we consider the case in which a representative diverse subset is computed, instead, whose size can be configured by the users themselves. For example, users may choose to set a budget k on the number of items they wish to receive whenever they login to their favorite notification service. For streams, it is important that the items retrieved be the users during consequent logins exhibit some *continuity* properties. For example, the order in which the diverse items are delivered to the users should follow the order of their generation. Also, an item should not appear, disappear and then re-appear in the diverse set.

For the efficient computation of diverse results in a dynamic setting, we propose a solution based on cover trees. Cover trees are data structures proposed for approximate nearest-neighbor search [8]. They were recently used to compute medoids [18] and priority medoids [9] of data.

We focus on the MAXMIN diversity problem defined as the problem of selecting k out of a set of n items so that the minimum distance between any two of the selected items is maximized. The MAXMIN and related problems are known to be NP-hard [13].

We provide theoretical results for the accuracy of the solution achieved using cover trees. We also introduce a batch construction

EDBT 2012, March 26–30, 2012, Berlin, Germany

that results in a cover tree with accuracy provably equivalent to that of the greedy heuristic for any k. For the continuous case, our incremental algorithms produce results of quality comparable to that achieved by re-applying the greedy heuristics to re-compute a diverse set, while avoiding the cost of re-computation. Using the cover tree also allows the efficient enforcement of the continuity requirements. Furthermore, multiple queries with different values of k can be supported.

In a nutshell, in this paper, we:

- introduce the dynamic diversification problem along with continuity requirements appropriate for a streaming scenario,
- propose indexing based on cover trees to address dynamic diversification,
- prove that the solution achieved by the cover tree for the MAXMIN problem is a $\frac{b-1}{2b^2}$-approximation of the optimal solution, where b is the base of the cover tree,
- provide a construction of a cover tree such that each of its levels corresponds to a solution of the greedy heuristic for various k,
- present incremental insertion and deletion algorithms for dynamic size cover trees and
- experimentally evaluate the efficiency and effectiveness of our approach using both real and synthetic datasets.

The rest of this paper is structured as follows. In Section 2, we present our diversification framework and define the CONTINUOUS k-DIVERSITY PROBLEM. Sections 3 and 4 present the cover tree index structure and introduce algorithms for computing diverse items in continuous environments and for dynamic insertions and deletions of items. Section 5 presents our experimental results. In Section 6, we review related work in the area of item diversification and show how our work relates with current approaches. Finally, Section 7 concludes this paper.

2. DIVERSIFICATION MODEL

There have been many proposals for defining diversity. Here, we focus on content diversity. In particular, given a set P of items we seek to select a subset S of P with the k most dissimilar items of P. First, we formally define diverse subsets and then we focus on special issues that arise when items arrive in streams.

2.1 The k-Diversity Problem

Let $P = \{p_1, \ldots, p_n\}$ be a set of n items and k be a positive integer, $k \leq n$. Let also $d : P \times P \rightarrow \mathbb{R}^+$ be a distance metric indicating the dissimilarity exhibited by two items in P. The *diversity* of a set S, $S \subseteq P$, is measured by a function $f : 2^{|P|} \times d \rightarrow \mathbb{R}^+$ which takes into account the dissimilarity of the items in S as indicated by d. Given a budget k on the number of items to select, the k-DIVERSITY PROBLEM aims at selecting the k items of P that exhibit the largest diversity among all possible combinations of items. Formally:

DEFINITION 1 (k-DIVERSITY PROBLEM). *Let P be a set of items, d a distance metric and f a function measuring the diversity of a set of items. Let also k be a positive integer. The* k-DIVERSITY PROBLEM *is to select a subset S^* of P such that:*

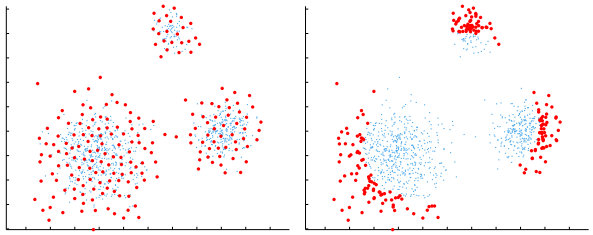
$$S^* = \underset{\substack{S \subseteq P \\ |S|=k}}{\operatorname{argmax}} f(S, d)$$

There are two main variations for content diversity concerning the choice of the function f: (i) maximizing the minimum distance among the selected items (MAXMIN) and (ii) maximizing the average distance among the selected items, which is equivalent to maximizing the sum of their distances (MAXSUM). We formally define the two variations as f_{MIN} and f_{SUM} respectively:

$$f_{\text{MIN}}(S, d) = \min_{\substack{p_i, p_j \in S \\ p_i \neq p_j}} d(p_i, p_j)$$

and

$$f_{\text{SUM}}(S, d) = \sum_{\substack{p_i, p_j \in S \\ p_i \neq p_j}} d(p_i, p_j)$$



(a) MAXMIN. (b) MAXSUM.

Figure 1: MAXMIN vs. MAXSUM solutions for $n = 1000$ and $k = 200$. Diverse items are shown in bold.

For example, Figure 1 depicts the $k = 200$ most diverse items selected from a set with $n = 1000$ items using the corresponding diversification functions. In the rest of this paper, we will focus on the MAXMIN problem, since, in general, this version tends to select items that intuitively provide a better cover of the set P.

In this paper, we further consider the case in which the set P changes over time and we want to refresh the computed k most diverse items to represent the updated set. In general, the insertion (or deletion) of even a single item may result in a completely different diverse set. The following simple example demonstrates this. Consider the set $P = \{(4, 4), (3, 3), (5, 6), (1, 7)\}$ of 2-dimensional points in the Euclidean space and $k = 2$. The two most diverse items of P are $(4, 4)$ and $(1, 7)$. Assume now that the item $(0, 0)$ is added to P. Now, the two most diverse items of P are $(0, 0)$ and $(5, 6)$.

The MAXMIN *Greedy Heuristic.*

The k-DIVERSITY PROBLEM has been shown to be NP-hard [13]. Various heuristics have been proposed, among which a natural greedy heuristic (Algorithm 1) has been shown experimentally to outperform the others in most cases [10, 14]. The algorithm starts by selecting the two items of P that are the furthest apart (line 1). Then, it continues by selecting the items that have the maximum distance from the items already selected, where the distance of an item p from a set of items S is defined as:

$$d(p, S) = \min_{p_i \in S} d(p, p_i)$$

It has been shown (e.g., in [21]) that the solution provided by the greedy heuristic is a $\frac{1}{2}$-approximation of the optimal solution.

2.2 The Continuous k-Diversity Problem

We consider applications where new items are generated in a continuous manner and, thus, the set P of available items changes gradually over time. Since items are continuously generated, we would like to present to the user a continuous view of the most diverse items in this stream. For example, consider a user contin-

Algorithm 1 MAXMIN Greedy Heuristic.

Input: A set of items P, an integer k.
Output: A set S with the k most diverse items of P.

```
1: p*, q* ← argmax_{p,q∈P, p≠q} d(p, q)
2: S ← {p*, q*}
3: while |S| < k do
4:     p* ← argmax_{p∈P} d(p, S)
5:     S ← S ∪ {p*}
6: end while
7: return S
```

uously receiving a representative, i.e., most diverse, subset of the stream of the tweets generated by the users she follows.

We adopt a sliding-window model where the k most diverse items are computed over sliding windows of length w in the input data. The length of the window w can be defined either in time units (e.g., "the k most diverse items in the last hour") or in number of items (e.g., "the k most diverse items among the 100 most recent ones"). Without loss of generality, we assume item-based windows. In this model, an item is selected, if and only if, it is part of the k most diverse items of the last w items.

We allow windows not only to slide but also to jump, i.e., to move forward more than one position in the stream of items each time. Assuming windows of length w and a jump step of length h, with $h \leq w$, consequent windows overlap and share $w-h$ common items. We call these windows *jumping windows* (Figure 2). Two consequent jumping windows correspond, for example, to the view of the available items between two visits of the user to her RSS feeder application. Between these two consequent visits, a number of items cease to be valid, a number of new items have been generated, while a number of older items are still valid and available to the user. Note that, for $h = 1$, jumping windows behave as regular sliding windows, while for $h = w$, windows are disjoint and we get periodic behavior with a period of length w.

Formally, let $\mathcal{P}$ be a stream of items. We denote the i^{th} jumping window of $\mathcal{P}$ as P_i and write $P_i = (p_1, \ldots, p_w)$, where $p_1, \ldots, p_w$ are the items that belong to P_i in order of their generation time. The UNCONSTRAINED CONTINUOUS k-DIVERSITY PROBLEM is to select a subset S_i^* of P_i for each P_i, such that:

$$S_i^* = \underset{\substack{S_i \subseteq P_i \\ |S_i|=k}}{\operatorname{argmax}} f(S_i, d)$$

Constrained Continuous k-Diversity Problem.

Since users may expect some continuity in the diverse sets they see in consequent retrievals, we consider the following additional requirements on how the items in diverse sets change over time. First, we want to avoid having diverse items which are still valid in the current window disappear. This may lead to confusing results, where an item appears in one window, disappears in the next one and then appears again. Thus, an item that was chosen as diverse will continue to be considered as such throughout the rest of its lifespan. We call this the *durability* requirement.

Second, we want the order in which items are chosen as diverse to follow the order they appear in the stream. This means that, once an item p is selected as diverse, we cannot later on select an item older than p as diverse. We call this the *freshness* requirement. This is a desirable property in case of notification services, such as news alerts and RSS feeds, since the items selected to be forwarded to the users follow the chronological order of their publication. Raising this requirement can result in out-of-order delivery of items which may seem unnatural to the users.

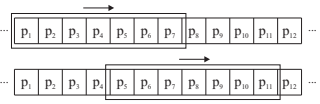


Figure 2: A jumping window with $w = 7$ and $h = 4$.

Based on the above observations, we formally define the CONSTRAINED CONTINUOUS k-DIVERSITY PROBLEM as follows:

DEFINITION 2. (CONSTRAINED CONTINUOUS k-DIVERSITY PROBLEM). *Let $\mathcal{P}$ be a stream of items and P_{i-1}, P_i be two consequent jumping windows. Let also d be a distance metric, f a function measuring the diversity of a set of items and k a positive integer. The* CONSTRAINED CONTINUOUS k-DIVERSITY PROBLEM *is to select a subset S_i^* of P_i for each P_i, such that:*

$$S_i^* = \underset{\substack{S_i \subseteq P_i \\ |S_i|=k}}{\operatorname{argmax}} f(S_i, d)$$

and also, given the diverse subset S_{i-1}^* for P_{i-1}, the following two constraints hold:

(i) $\forall p_j \in S_{i-1}^* \cap P_i \Rightarrow p_j \in S_i^*$ (durability requirement),

(ii) Let p_l be the newest item in S_{i-1}^*. Then, $\nexists p_j \in P_i \backslash S_{i-1}^*$ with $j < l$, such that, $p_j \in S_i^*$ (freshness requirement).

3. INDEX-BASED DIVERSIFICATION

In this paper, we aim at developing a diversification method that can be applied to dynamic environments. To this end, we employ a tree structure to index the available items. Our approach is based on the cover tree, defined next.

3.1 The Cover Tree

The Cover Tree (CT) [8] for a data set P is a leveled tree where each level is a "cover" for all levels beneath it. Each level of the tree is associated with an integer $\ell \in (-\infty, \infty)$. ℓ decreases as the tree is descended. Intuitively, the lowest level contains all items in the set and, as we move up the tree, subsets of the items are promoted based on their distances. Items at higher levels of the tree are farther apart from each other than items at lower levels of the tree. An example is shown in Figure 3. Each node in the tree is associated with exactly one item $p \in P$, while each item may be associated with multiple nodes in the tree. However, each item is associated with at most a single node at each level.

In the following, when clear from context, we will use p to refer to both the item p and a node in the tree at a specific level that is associated with p.

Let C_ℓ be the set of nodes at level ℓ of a cover tree. The cover tree of *base* b, $b > 1$, obeys the following invariants for all ℓ:

1. **Nesting**: $C_\ell \subset C_{\ell-1}$, i.e., once an item p appears in the tree at some level, then every lower level has a node associated with p.

2. **Covering**: For every $p_i \in C_{\ell-1}$, there exists a $p_j \in C_\ell$, such that, $d(p_i, p_j) \leq b^\ell$ and the node associated with p_j is the parent of the node associated with p_i.

3. **Separation**: For all distinct $p_i, p_j \in C_\ell$, it holds that, $d(p_i, p_j) > b^\ell$.

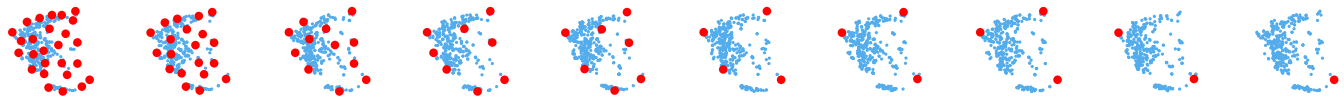


Figure 3: An example of the 10 first levels of a cover tree for a set of items in the 2-dimensional space. Bold points represent the items (i.e., nodes) at each level, moving from lower to higher levels, as we move from left to right.

The cover tree was originally proposed with base $b = 2$. In this paper, we use a more general base b, $b > 1$. Generally, larger base values result in shorter and wider trees, since fewer nodes are able to "cover" the nodes beneath them. The value of b determines the granularity with which we move from one level to the next, i.e., how many more items become visible when we descend the tree.

Due to the invariants of the cover tree, if an item p appears first in level ℓ of the tree, then p is a child of itself in all levels below ℓ. An *implicit* representation of a cover tree, i.e., storing all nodes, requires space depending not only on the number of items but also on their pairwise distances, since those distances determine the number of required levels. However, self-children are not required to be explicitly stored since we can always deduce their existence in lower levels. The *explicit* representation of a cover tree removes all nodes which have only self-children in their subtrees (or only self-parents in case of leaf nodes). The explicit representation of a cover tree for P, with $|P| = n$, requires $O(n)$ space. While in our experiments we used the explicit representation, for ease of presentation, we shall use the implicit representation when describing the algorithms and assume an infinite number of levels, with $\ell = -\infty$ representing the lowest level that includes all items in P and $\ell = +\infty$ representing the root level, unless otherwise specified.

We make the following observation which is important for diversification: items at sibling nodes of higher levels of a cover tree are further apart from each other than those at lower levels due to the separation invariant. Thus, by selecting items from higher levels of the tree, we can retrieve results that exhibit higher diversity.

3.2 Computing Diverse Subsets

Next, we show how we can exploit the cover tree to retrieve the k most diverse items of a set P. Let S be a solution for the k-DIVERSITY PROBLEM and ℓ be the highest level in which all items in S appear in the cover tree (this condition must hold at some level due to the nesting invariant). Then, the diversity of S is larger than b^ℓ, due to the separation invariant. Therefore, we aim at selecting a subset S of the nodes of the tree that appear as high as possible in the tree.

Algorithm 2 selects the k most diverse items using a cover tree. We start from the root of the tree and descend until we reach the first level ℓ such that $|C_\ell| \leq k$ and $|C_{\ell-1}| > k$. At each level, we add to the solution all items found. Note that, due to nesting, an item that appears in any level m of the tree also appears in all levels below m. The remaining $k - |C_\ell|$ items are selected from $C_{\ell-1}$. The selection is done in a greedy manner so that the most diverse amongst the items of $C_{\ell-1}$ are selected (lines 7-11).

The two requirements of Definition 2 can be easily enforced using the cover tree. For the durability requirement, items that are selected as diverse are marked so and remain part of the diverse set until they expire. Let r be the number of such items. In this case, Algorithm 2 just selects $k - r$ additional items from the tree. For the freshness requirement, non-diverse items that are older than the newest diverse item in the current diverse set are marked as "invalid" in the cover tree. These items are not considered further as candidates for inclusion.

Algorithm 2 Diverse Item Computation Using a Cover Tree.

Input: A cover tree T, an integer k.
Output: A set S with the k most diverse items in T.

1: $\ell \leftarrow +\infty$
2: **while** $|T.C_{\ell-1}| \leq k$ **do**
3: $\quad \ell \leftarrow \ell - 1$
4: **end while**
5: $S \leftarrow T.C_\ell$
6: $Candidates \leftarrow T.C_{\ell-1}$
7: **while** $|S| < k$ **do**
8: $\quad p^* \leftarrow \text{argmax}_{p \in Candidates}\, d(p, S)$
9: $\quad S \leftarrow S \cup \{p^*\}$
10: $\quad Candidates \leftarrow Candidates \backslash \{p^*\}$
11: **end while**
12: **return** S

3.3 Approximation Bound

The next theorem characterizes the quality of the solution of the diversity algorithm that selects items from the top levels of any cover tree.

THEOREM 1. *Let P be a set of items, $k \geq 2$, $d^{OPT}(P,k)$ be the optimal minimum distance for the* MAXMIN *problem and $d^{CT}(P,k)$ be the minimum distance of the diverse set computed by the diversity algorithm on a cover tree for P (Algorithm 2). It holds that $d^{CT}(P,k) \geq \alpha\, d^{OPT}(P,k)$, where $\alpha = \frac{b-1}{2b^2}$.*

PROOF. Let $S^{OPT}(P,k)$ be an optimal set of k diverse items. To prove Theorem 1, we shall bound the level where the least common ancestor (LCA) of any pair of items p_1, $p_2 \in S^{OPT}(P,k)$ appears in the cover tree. Assume that the LCA of any two items p_1, p_2 in the optimal solution appears for the first time at level m. That is, m is the lowest (furthest from the root) level that such an LCA appears.

Let us now compute a bound on m. Assume that the LCA of any two items p_1, $p_2 \in S^{OPT}(P,k)$ appears at level m. Let p be this ancestor. From the triangle inequality, $d(p_1,p) + d(p_2,p) \geq d(p_1,p_2)$. Since p_1, $p_2 \in S^{OPT}(P,k)$, it holds that, $d(p_1,p_2) \geq d^{OPT}(P,k)$. Thus:

$$d(p_1,p) + d(p_2,p) \geq d^{OPT}(P,k) \tag{1}$$

From the covering invariant of the cover tree, it holds that, $d(p_1,p) \leq \sum_{j=-\infty}^{m} b^j \leq \frac{b^{m+1}}{b-1}$. Similarly, $d(p_2,p) \leq \frac{b^{m+1}}{b-1}$. Substituting in (1), we get that $2\,\frac{b^{m+1}}{b-1} \geq d^{OPT}(P,k)$. Solving for m, we have $m \geq \log_b\left(\frac{b-1}{2} d^{OPT}(P,k)\right) - 1$.

Since m is the first level that the LCA of any two items in the optimal solution appears, from the covering property, it holds that at level $m-1$, there are at least k items, i.e., the distinct ancestors of the k items in the optimal solution. Thus, there are at least k items at level

$$m - 1 = \log_b\left(\frac{b-1}{2} d^{OPT}(P,k)\right) - 2 \tag{2}$$

This means that the cover tree algorithm will select items from this

or a higher level. From the separation invariant of the cover tree, we have $d^{CT}(P,k) \geq b^{m-1}$. Using (2), we get that $d^{CT}(P,k) \geq b^{\log_b(\frac{b-1}{2} d^{OPT}(P,k))-2} \Rightarrow d^{CT}(P,k) \geq \frac{b-1}{2} d^{OPT}(P,k)\ b^{-2}$, which proves the theorem. □

Note that the approximation bound holds independently of how the items at the lower level are selected. In Algorithm 2, this selection is done greedily (lines 7-11), but this does not affect the theoretical bound.

3.4 Changing *k*

The cover tree can be used to provide results for multiple queries with different k. Thus, each user can individually tune the amount of diverse items she wishes to receive. Furthermore, the cover tree supports a "zooming" type of functionality. Assume that a user selects a specific value for k. After receiving the k most diverse items, she can request a larger number of closer to each other items by choosing a larger k ("zoom-in") or a smaller number of further apart items by choosing a smaller k ("zoom-out").

We can exploit the nesting invariant to achieve a sense of continuity in the following sense. Let S be the set of the k most diverse items and let ℓ be the highest level of the cover tree at which all items of S appear. For $k' > k$, to construct the set S' with the k' most diverse items, we select items from level ℓ or lower (Algorithm 2), thus, we can enforce the condition $S' \supset S$, since the items in S appear in all levels $m \leq \ell$. For $k' < k$, to construct the set S' with the k' most diverse items, and also enforce $S' \subset S$, we may select those items of S that appear at levels higher than ℓ.

4. COVER TREE CONSTRUCTION

Given a set of items P, there may be many different cover trees that maintain the three invariants. In this section, we first present an algorithm for a batch construction of a cover tree for P appropriate for the MAXMIN problem and then consider dynamic insertions and deletions.

4.1 Batch Construction

In the batch construction of a cover tree, the tree is built bottom-up. The algorithm proceeds greedily by promoting to higher levels the items with the largest possible distance from the already selected ones as long as the cover tree invariants are not violated.

This process is shown in Algorithm 3. First, the lowest level ℓ of the cover tree is formed by adding to it all items in P (lines 1-5). Then, we select items from the lowest level whose distance is more than $b^{\ell+1}$, i.e., they cannot be a child of each other at the new level $\ell+1$ due to the separation invariant (lines 7-17). The selected items form the new level $\ell+1$ and the remaining items are distributed among them so that the covering invariant holds (lines 18-21). The nesting invariant clearly holds as well since every item is either promoted or assigned to some parent in the new level.

In order to construct the cover tree in a way that maximizes the diversity of the items in the higher nodes of the tree, we employ the following heuristic. When selecting which items from C_ℓ to promote to the next level $C_{\ell+1}$, we follow a greedy approach; we start by promoting the two items that are the furthest apart and then add in rounds the item that has the largest minimum distance from the already promoted ones (line 14). We refer to the cover tree constructed using the above heuristic as the Batch Cover Tree (or BCT) for P and b.

To further improve the tree, when distributing the remaining items of C_ℓ to the nodes of $C_{\ell+1}$, we assign each of them to its closest candidate parent (line 19). We call this step *nearest parent* heuristic. The motivation for this heuristic is to reduce the overlap among the areas covered by sibling nodes.

Algorithm 3 Batch Cover Tree Construction.

Input: A set of items P, a base b.
Output: A cover tree T of base b for P.

1: $\ell \leftarrow \lfloor \log_b (\min_{p,q \in P} d(p,q)) \rfloor$
2: $Q_\ell \leftarrow \emptyset$
3: **for all** $p \in P$ **do**
4: $\quad Q_\ell \leftarrow Q_\ell \cup \{p\}$
5: **end for**
6: **while** $|T.C_\ell| > 1$ **do**
7: $\quad T.C_{\ell+1} \leftarrow \emptyset$
8: $\quad Candidates \leftarrow T.C_\ell$
9: $\quad p^*, q^* \leftarrow \text{argmax}_{p,q \in Candidates}\, d(p,q)$
10: $\quad T.C_{\ell+1} \leftarrow T.C_{\ell+1} \cup \{p^*, q^*\}$
11: $\quad Candidates \leftarrow Candidates \backslash \{p^*, q^*\}$
12: $\quad$ **while** $Candidates \neq \emptyset$ **do**
13: $\quad\quad Candidates \leftarrow Candidates \backslash \{p : \exists q \in T.C_{\ell+1}$ with $d(p,q) \leq b^{\ell+1}\}$
14: $\quad\quad p^* \leftarrow \text{argmax}_{p \in Candidates}\, d(p, T.C_{\ell+1})$
15: $\quad\quad T.C_{\ell+1} \leftarrow T.C_{\ell+1} \cup \{p^*\}$
16: $\quad\quad Candidates \leftarrow Candidates \backslash \{p^*\}$
17: $\quad$ **end while**
18: $\quad$ **for all** $p \in T.C_\ell$ **do**
19: $\quad\quad q^* \leftarrow \text{argmin}_{q \in T.C_{\ell+1}}\, d(p, T.C_{\ell+1})$
20: $\quad\quad$ make q parent of p
21: $\quad$ **end for**
22: $\quad T.C_\ell \leftarrow T.C_{\ell+1}$
23: $\quad \ell \leftarrow \ell + 1$
24: **end while**
25: **return** $T.C_\ell$

We shall prove that the items at each level ℓ of a BCT are the result of applying the greedy heuristic (Algorithm 1) on P when k is set equal to the number of items of this level (i.e., for $k = |C_\ell|$. To do so, we shall use the following observation for the greedy algorithm.

OBSERVATION 1. *Let $S^{GR}(P,k)$ be the result of the greedy heuristic for k and P. For any $k > 2$, it holds that, $S^{GR}(P,k+1) \supset S^{GR}(P,k)$.*

THEOREM 2. *For any batch cover tree T for a set of items P, it holds that,*

$$\forall \text{ level } \ell \text{ of } T, C_\ell = S^{GR}(P, |C_\ell|)$$

where C_ℓ is the set of items at level ℓ of T.

PROOF. We shall prove the theorem by induction on the level ℓ. The theorem holds trivially for ℓ equal to the lowest level of the tree, since this level includes all items in P. Assume that it holds for level ℓ. We shall show that it also holds for level $\ell+1$.

Consider the construction of level $\ell+1$. From the induction step, it holds that, $C_\ell = S^{GR}(P, |C_\ell|)$.

Let p be the *first* item in C_ℓ such that p is the best candidate, i.e., has the maximum minimum distance from the items already selected, but cannot be moved to $C_{\ell+1}$ because it is covered by an item already selected to be included in $C_{\ell+1}$. Let C', $C' \subset C_\ell$, be the set of items already selected to be included in $C_{\ell+1}$. This means that, for p, it holds: $\min_{q' \in C'} d(p,q') \geq \min_{q' \in C'} d(p',q')$, for all $p' \in C_\ell \backslash C'$ (1) and, also, $\exists\ q \in C'$ such that $d(p,q) \leq b^{\ell+1}$ (2). From (1) and (2), we get that for all $p' \in C_\ell \backslash C'$, $\exists\ q \in C'$ such that $d(p',q) \leq b^{\ell+1}$, that is, all remaining items are also already covered by items in C'.

Thus, p is the last item that is considered for inclusion in $C_{\ell+1}$, since all other remaining items in C_ℓ are already covered. There-

fore, to construct $C_{\ell+1}$, the items from C_ℓ to be included in level $\ell+1$ are considered in the same order as in the greedy heuristic, until one item that violates the separation criterion (it is covered by the selected items) is encountered. When this happens the selection stops. By the induction step and Observation 1, this concludes the proof. □

Note that, we have made an implicit assumption that no ties are present when selecting items. In the absence of ties, both the greedy heuristic and the cover tree algorithm select items deterministically. We can raise this assumption, by considering that if ties exist, these are resolved in a specific order that may vary depending on the nature of the items, for instance, by selecting the most recent among the items.

Regarding the complexity of Algorithm 3, most computational steps are shared among levels. Each level $C_{\ell+1}$ is a subset of C_ℓ and, more specifically, it consists of the items of C_ℓ in the order that they were inserted into C_ℓ until the first item whose minimum distance from the already selected items of C_ℓ at the point of its insertion is smaller than $b^{\ell+1}$. Therefore, it suffices to perform these computational steps only once (at the lower level) and just maintain the order at which each item was selected from the lowest level for promotion to the next one.

Finally, from Theorem 2, we get that:

COROLLARY 1. *Let P be a set of items, $k \geq 2$, $d^{GR}(P,k)$ be the minimum distance of the diverse set computed by the greedy heuristic (Algorithm 1) and $d^{BCT}(P,k)$ be the minimum distance of the diverse set computed by the cover tree algorithm (Algorithm 2) when applied on a batch cover tree for P. It holds that $d^{GR}(P,k) = d^{BCT}(P,k)$.*

As a final remark, another way to view the batch cover tree is as caching the results of the greedy heuristic for all k and indexing them for efficient retrieval.

4.2 Dynamic Construction

In dynamic environments, it is not efficient to re-construct a batch cover tree whenever an item is inserted or deleted. Thus, we construct a cover tree for $\mathcal{P}$ *incrementally* as new items arrive and old ones expire. Next, we present insertion and deletion algorithms for a cover tree of base b. We refer to such trees as Incremental Cover Trees.

Incremental Insertion.

The procedure for inserting a new item p into a cover tree is shown in Algorithm 4. Algorithm 4 is based on the insertion algorithm in [8] and subsequent corrections in [17]. We have extended the original algorithms to work for any $b > 1$. The algorithm takes as input the new item p and a set of candidate nodes Q_ℓ at level ℓ under which the new item could be inserted. The tree is descended until a level is found in which p is separated from all other items (lines 2-4). Each time the tree is descended, only the nodes that cover p are considered (line 5). When the first level in which p is separated from all other items is located, the node that covers p and is closest to it is selected as its parent (lines 8-9).

Next, we prove the correctness of the algorithm for any $b > 1$.

THEOREM 3. *Algorithm 4 with input an item p and the root level C_∞ at level ∞ of a cover tree T of base b for P returns a cover tree of base b for $P \cup \{p\}$.*

PROOF. Assuming that p is not already in the tree, since ℓ can range from $+\infty$ to $-\infty$, there is always a (sufficiently low) level

Algorithm 4 Insertion.

Input: An item p, a set of nodes $T.Q_\ell$ of a cover tree T at level ℓ.
Output: A cover tree T.

1: $C \leftarrow \{children(q) : q \in T.Q_\ell\}$
2: **if** $d(p,C) > b^\ell$ **then**
3: **return** true
4: **else**
5: $T.Q_{\ell-1} \leftarrow \{q \in C : d(p,q) \leq \frac{b^\ell}{b-1}\}$
6: $flag \leftarrow Insertion(p, T.Q_{\ell-1}, \ell-1)$
7: **if** $flag$ **and** $d(p,T.Q_\ell) \leq b^\ell$ **then**
8: $q^* \leftarrow \text{argmin}_{q \in T.Q_\ell, d(p,q) \leq b^\ell} d(p,q)$
9: make p a child of q^*
10: **return** false
11: **else**
12: **return** flag
13: **end if**
14: **end if**

Algorithm 5 Deletion.

Input: An item p, sets of nodes $\{T.Q_\ell, T.Q_{\ell+1}, \ldots, T.Q_\infty\}$ of a cover tree T at level ℓ.
Output: A cover tree T.

1: $C \leftarrow \{children(q) : q \in T.Q_\ell\}$
2: $T.Q_{\ell-1} \leftarrow \{q \in C : d(p,q) \leq \frac{b^\ell}{b-1}\}$
3: $Deletion(p, \{T.Q_{\ell-1}, T.Q_\ell, \ldots, T.Q_\infty\}, \ell-1)$
4: **if** $d(p,C) = 0$ **then**
5: delete p from level $\ell-1$ and from $Children(Parent(p))$
6: **for** $q \in Children(p)$ in greedy order **do**
7: $\ell' \leftarrow \ell - 1$
8: **while** $d(q, T.Q_{\ell'}) > b^{\ell'}$ **do**
9: add q into level ℓ'
10: $T.Q_{\ell'} \leftarrow T.Q_{\ell'} \cup \{p\}$
11: $\ell' \leftarrow \ell' + 1$
12: **end while**
13: $q^* \leftarrow \text{argmin}_{q' \in T.Q_{\ell'}} d(p', q)$
14: make q a child of q^*
15: **end for**
16: **end if**

of the tree where the condition of line 2 first holds. Let $\ell-1$ be that level. Since $\ell-1$ is the highest level that the condition holds, then it must hold that $d(p,Q_\ell) \leq b^\ell$. Therefore, the second condition of line 7 always holds and we can always find a parent for the new node that was inserted, thus maintaining the covering invariant. Whenever a new node is added at some level, it is also added in all lower levers as a child of itself, thus maintaining the nesting invariant. It remains to prove that this does not violate the separation invariant. To do this, consider some other item q in level $\ell-1$. If $q \in C$, then $d(p,q) > b^{\ell-1}$. If not, then there is a higher level $\ell' > \ell$ where some ancestor of q, say q' was eliminated by line 5, i.e., $d(p,q') > \frac{b^\ell}{b-1}$. Using the triangle inequality, we have that $d(p,q) \geq d(p,q') - d(q,q') = d(p,q') - \sum_{j=\ell}^{\ell'-1} b^j = d(p,q') - \left(\frac{b^\ell - b^{\ell'}}{1-b}\right) > \frac{b^\ell}{b-1} + \frac{b^\ell - b^{\ell'}}{b-1} = \frac{b^\ell}{b-1} > b^{\ell-1}$. Thus, the separation invariant is maintained as well. □

Note that, in selecting a parent for p we use a *nearest parent* heuristic (as in the batch construction) to assign p to its closest parent (line 8). This step is not necessary for the correctness of the insertion. Instead, we could select any node for which $d(p,q) \leq b^\ell$ as a parent of p. We choose q^* to better separate the items of each tree level.

For clarity of presentation, Algorithm 4 assumes that the new item p can be inserted in some existing level of the tree. In case

Table 1: Computational cost of GR and BCT.

b	*Uniform* (1000 items)			*Clustered* (1000 items)			*Cities* (1000 items)		
	step (i) (GR)	step (ii) - np	step (ii)	step (i) (GR)	step (ii) - np	step (ii)	step (i) (GR)	step (ii) - np	step (ii)
1.3	166827666	0.57%	0.40%	166730842	0.58%	0.42%	165827166	0.58%	0.41%
1.5	166347120	0.56%	0.40%	166416457	0.56%	0.42%	165827166	0.56%	0.41%
1.7	166347120	0.55%	0.39%	165943822	0.55%	0.41%	167101212	0.55%	0.40%
b	***Forest* (1000 items)**			***Faces* (300 items)**			***Flickr* (400 items)**		
	step (i) (GR)	**step (ii) - np**	**step (ii)**	**step (i) (GR)**	**step (ii) - np**	**step (ii)**	**step (i) (GR)**	**step (ii) - np**	**step (ii)**
1.3	167020404	0.57%	0.43%	4499652	1.94%	1.49%	10684011	1.25%	1.00%
1.5	167020404	0.56%	0.43%	4499652	1.92%	1.47%	10737077	1.11%	0.92%
1.7	166565859	0.54%	0.42%	4499652	1.91%	1.47%	10448280	1.04%	0.89%

of static data, where we have the prior knowledge of the distances among the items, we are able to determine the maximum and minimum levels of the tree beforehand. However, when the indexed data change dynamically, this is not the case. We have modified our algorithm to meet this extra challenge. More specifically, whenever a new item arrives that has a larger distance from the root node than $b^{\ell_{max}}$, where ℓ_{max} is the maximum level of the tree, we promote both the root node and p to a new higher level and repeat this process until one of the two nodes can cover the other. Also, whenever a new item p must be indexed in some lever lower than ℓ_{min}, where ℓ_{min} is the minimum level of the tree, we copy all nodes of $C_{\ell_{min}}$ to a new level $C_{\ell_{min}-1}$ until the new item p is separated from all other items in the new level. Note that, since the explicit representation of the tree is stored, this duplication of levels is only virtual and can be performed very efficiently.

Incremental Deletion.

The procedure for deleting items from a cover tree is shown in Algorithm 5. The procedure descends the tree searching for the item p to be removed, keeping note of the candidate nodes of each level that may have p as a descendant. After p is located, it is removed from the tree. In addition, all its children are reassigned to some of the candidate nodes.

Algorithm 5 includes two heuristics for improving the quality of the resulting cover tree. One is the usual *nearest parent* heuristic shown in line 13: we assign each child of p to the closest among the candidate parents. The other heuristic refers to the order in which the children of p are examined in line 6. We examine them in a greedy manner starting from the one furthest apart from the items in level ℓ' and continue to process them in decreasing order of their distance to the items currently in ℓ'. These heuristics also do not affect the correctness of the algorithm.

THEOREM 4. *Algorithm 5 with input an item p and the root level C_∞ at level ∞ of a cover tree T of base b for P returns a cover tree of base b for $P \setminus \{p\}$.*

PROOF. The item p is removed from all levels that include it, thus the nesting invariant is maintained. For each child q of p, we move up the tree, until a parent for q is located, inserting q in all intermediate levels ℓ' to ensure that the nesting invariant is not violated. Such a parent is guaranteed to be found (at least at the level of the root). Adding q under its new parent does not violate the separation invariant in any of the intermediate levels since $d(q, q') > b^{\ell'}$, for all q' in $Q_{\ell'}$. The covering constraint also holds for the parent of q. □

As in the case of insertions, we also adjust ℓ_{max} for the tree after each deletion (ℓ_{min} does not require adjustment in case the explicit representation is stored). Whenever the root node is deleted, we must select a new root. Note that, it is possible that none of the children of the old root are able to cover all of its siblings. In this case, we promote those of the siblings that continue to be separated from each other in a new (higher) level and continue to do so until we reach a level that is high enough so that a single root node can be selected.

Table 2: Characteristics of Datasets.

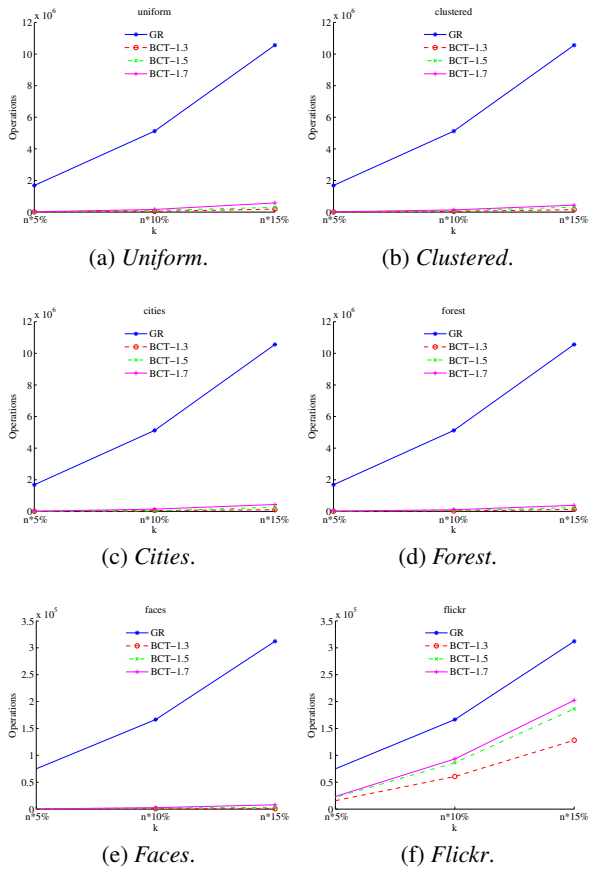
Dataset	Cardinality	Dimensions	Distance metric
Uniform	10000	2	Euclidean
Clustered	10000	2	Euclidean
Cities	5922	2	Euclidean
Forest	5000	10	Cosine
Faces	300	256	Cosine
Flickr	18245	-	Jaccard

5. EVALUATION

In this section, we experimentally evaluate the performance of the cover tree when employed for selecting diverse results. This evaluation is performed both in terms of the achieved diversity of the selected items, as well as, the computational cost of maintaining a cover tree in the case of dynamic insertions and deletions. The second aspect of our evaluation is itself interesting, since, to the best of our knowledge, there is no evaluation of the behavior of the cover tree in the case of dynamic data changes.

5.1 Setup

In our evaluation, we use a variety of datasets, both real and synthetic. Our synthetic datasets consist of 10000 multi-dimensional items in the Euclidean space, where each dimension takes values in $[0, 1]$. Items are either uniformly distributed ("*Uniform*") or form (hyper)spherical clusters of different sizes ("*Clustered*"). We also employ a number of real datasets. The first one is a collection of 2-dimensional points representing geographical information about 5922 cities and villages of Greece ("*Cities*") [5]. Due to the geography of Greece, which includes a large number of islands, this dataset provides us with both dense and sparse areas of points which makes it suitable for evaluating diversification methods. The second real dataset ("*Forest*") contains forest cover information for areas in the United States, such as elevation and distance to hydrology [4]. 10 features are present for 5000 locations. The third dataset ("*Faces*") consists of 256 features extracted from each of 300 human face images with the eigenfaces method [1]. Finally, for our last real dataset ("*Flickr*"), we used data from [3] which consists of tags assigned by users to photographs uploaded to the Flickr photo service [2] from January 2004 to December 2005. Since the available descriptions span over an extremely large space due to the great variety of available photographs, we concentrated on a subset of them by extracting all tags for photographs that were tagged with



(a) *Uniform*. (b) *Clustered*.

(c) *Cities*. (d) *Forest*.

(e) *Faces*. (f) *Flickr*.

Figure 4: Total number of operations for retrieving the k most diverse items for GR and BCTs with base values of 1.3, 1.5 and 1.7.

a specific keyword, namely the keyword "holiday". Throughout our evaluation, we used different distance metrics for our datasets. We used the Euclidean distance for "*Uniform*", "*Clustered*" and "*Cities*", while for "*Faces*" and "*Forest*", we employed the Cosine distance. Finally, for "*Flickr*", we used the Jaccard distance among the sets of tags describing each photograph. The characteristics of our datasets are summarized in Table 2.

We first compare the performance of the Batch Cover Tree (BCT) and the Incremental Cover Tree (ICT) against the greedy heuristic (GR). For comparison, we also report results of randomly selecting k of the available items (RA). Then, we concentrate on the behavior of ICT in a streaming scenario and also report on the quality of the achieved solutions for the CONSTRAINED CONTINUOUS k-DIVERSITY PROBLEM.

All approaches are implemented in Java using JDK 1.6. Our experiments were executed on an Intel Pentium Core2 2.4GHz PC with 2GB of RAM.

5.2 Comparison of BCT, ICT and GR

First, we compare BCT against GR. Diversity-wise, as shown in Corollary 1, the solutions provided by BCT are identical to those of GR, which is generally one of the best performing heuristics for computing diverse results [10]. Due to the NP-hardness of the k-DIVERSITY PROBLEM, it is not possible to compute optimal solutions in reasonable time even for moderate values of n and k, and therefore, such values are not reported here. However, we know

Table 3: Height of produced BCTs and ICTs (in parentheses).

Maximum level						
b	***Uniform***	***Clustered***	***Cities***	***Forest***	***Faces***	***Flickr***
1.3	2 (1)	7 (6)	-6 (-7)	-1 (-2)	0	0
1.5	1	4	-4	-1	0	0
1.7	1	4 (3)	-3	0 (-1)	0	0
Minimum level						
b	**uniform**	**clustered**	**cities**	**forest**	**faces**	**flickr**
1.3	-26	-25	-35	-45	-45	-8
1.5	-17	-16	-23	-29	-29	-6
1.7	-13	-12	-18	-22	-22	-4

Table 4: Computational cost of ICT as compared to BCT.

b	***Uniform* (1000 items)**		***Clustered* (1000 items)**		***Cities* (1000 items)**	
	ICT - np	**ICT**	**ICT - np**	**ICT**	**ICT - np**	**ICT**
1.3	0.16%	0.16%	0.16%	0.16%	0.15%	0.15%
1.5	0.08%	0.08%	0.08%	0.08%	0.07%	0.07%
1.7	0.05%	0.05%	0.06%	0.06%	0.05%	0.05%
b	***Forest* (1000 items)**		***Faces* (300 items)**		***Flickr* (400 items)**	
	ICT - np	**ICT**	**ICT - np**	**ICT**	**ICT - np**	**ICT**
1.3	0.13%	0.13%	0.78%	0.79%	3.83%	3.84%
1.5	0.07%	0.07%	0.41%	0.41%	2.10%	2.10%
1.7	0.05%	0.05%	0.28%	0.28%	1.36%	1.36%

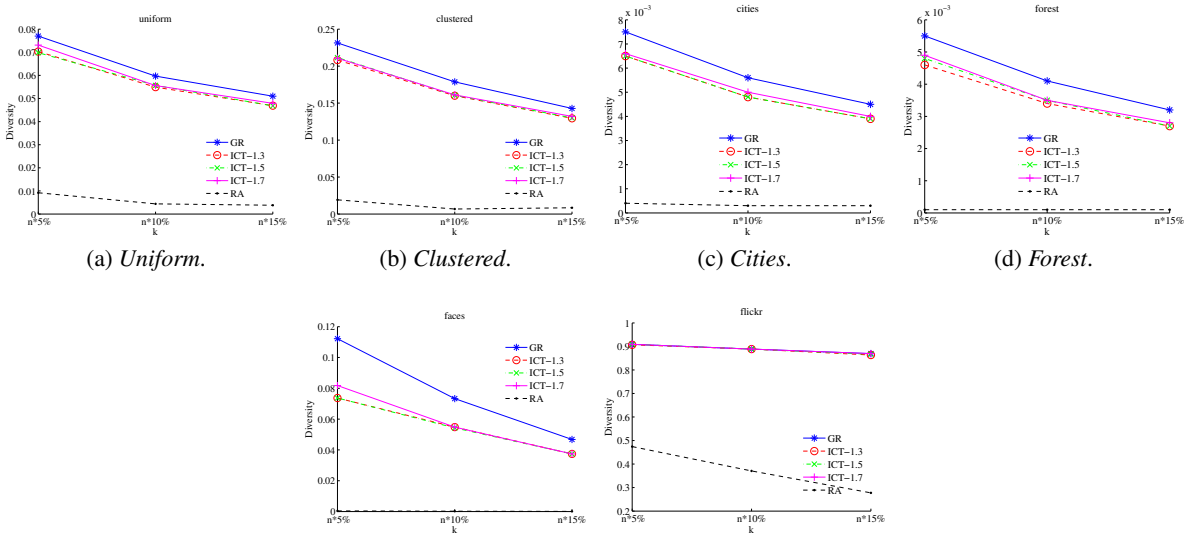
that GR, and thus BCT, provide a $\frac{1}{2}$-approximation of the optimal solution.

We focus our evaluation on the computational cost of building a BCT compared to that of employing GR. We measure this cost in terms of *operations*, i.e., distance computations required to be calculated, instead of other measures, such as time. The reason for this is that the cost of a single distance computation varies a lot depending on the distance metric used. For example, computing Jaccard distances between sets of items is more expensive than computing Euclidean distances. To evaluate this computational cost, we use 1000 random items for all datasets except from "*Faces*" where there are only 300 available items and "*Flickr*" where the distance computations are much more time-consuming.

The cost of building a BCT consists of: (i) performing operations at the lowest (leaf) level among the n available items in order to build the first non-leaf level and then (ii) performing a number of operations to assign nodes to the most suitable parent as the upper levels of the tree are constructed. We measure the operations performed at steps (i) and (ii) separately. Operations required by step (i) are also the operations required by GR. Therefore, the actual extra cost of building a BCT is reflected by the operations of step (ii). The amount of these operations differs depending on whether our nearest-parent heuristic (denoted "np") is employed or not.

Table 1 shows the required operations to build BCTs for different values of base b. For clarity, we report the absolute number of operations required by step (i) and the additional operations required by step (ii) as a percentage of the operations required by step (i). For example, for $b = 1.3$ and the "*Uniform*" dataset, step (ii) of BCT with the nearest-parent heuristic requires only an additional 0.57% of the operations of step (i). When we just need to compute the k most diverse items only once, employing GR would be preferable. However, building a BCT comes at little extra cost and can be used to answer queries for multiple values of k and for the case of dynamic insertions and deletions of items.

Next, we show the cost of retrieving the k most diverse items from a BCT as opposed to retrieving them employing GR with var-



(a) *Uniform*. (b) *Clustered*. (c) *Cities*. (d) *Forest*.

(e) *Faces*. (f) *Flickr*.

Figure 5: Achieved diversity of ICT and GR.

ious values of k expressed as a percentage of the total number of available items n (Figure 4). We see that retrieving diverse items from an already constructed BCT has much smaller cost than executing GR from scratch, even for small values of k. The cost is higher for the "*Flickr*" dataset because the constructed tree is short and wide, which results in more operations performed by lines 8-12 of Algorithm 2. The heights of the produced BCTs for our different datasets can be seen in Table 3 where we report the ℓ values for the highest and lowest levels. The BCTs for the "*Flickr*" dataset are considerably shorter because the average distance among the items of this dataset is larger than those of the other datasets. Another interesting observation is that the trees constructed for the "*Faces*" and "*Forest*" datasets are almost of the same height, even though the size of the "*Faces*" dataset is around one third of the size of the "*Forest*" dataset.

We also report the cost of building an ICT as compared to the cost of building a BCT (Table 4). To construct such trees, we used our dynamic version of $Insertion$ to insert all items of the corresponding datasets to the trees. The cost is shown as a percentage of the total cost (step (i) and step (ii)) of constructing the BCT tree for each dataset. Clearly, building ICTs has a much lower cost than their BCT counterparts. Figure 5 reports the quality of the solutions produced by GR, and thus BCT, and ICT for our datasets. ICTs achieve very good solutions, while their cost is very small compared to that of GR and BCTs. In general, the produced ICT for a dataset is different than the BCT for this dataset. However, in most cases they both have the same height. Table 3 reports in parentheses the height of the ICT, for the cases in which, this is different from the height of the corresponding BCT.

To give some intuition concerning actual execution times, performing the greedy step to execute GR, and also construct the lower level of a BCT, requires, with our current implementation, around 7 seconds for our Euclidean datasets and the reported b values. Building the corresponding ICTs is much more efficient, requiring under 0.5 seconds on average.

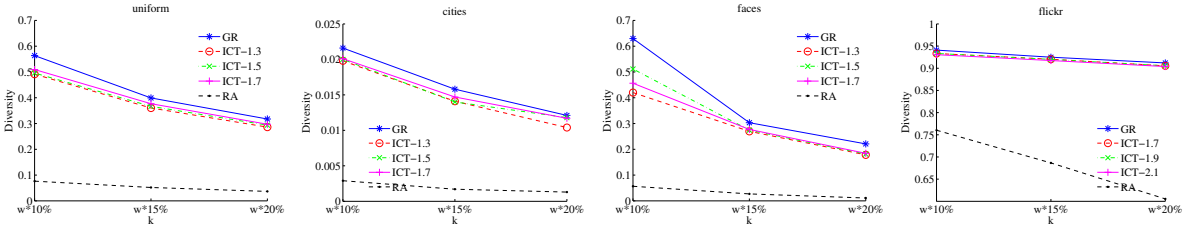
5.3 Continuous Data

Next, we concentrate on how our approach performs with streaming. To simulate the order in which data arrives, we used the photo upload time for our "*Flickr*" dataset and, for the rest of our datasets, we randomly permuted the items so that they enter the stream in a random order. Due to space constraints, in this section, we omit the "*Clustered*" and "*Forest*" datasets. The results are similar to the other cases.

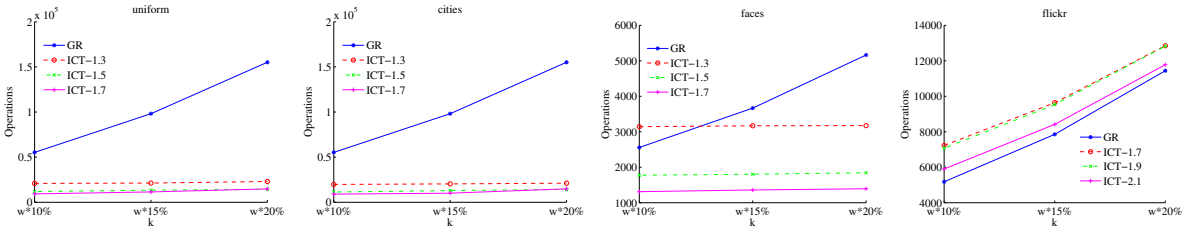
Figures 6, 7 and 8 show results for different values of the number of required items k, the length of the jumping window w and the jumping step of the window h respectively. w is selected as a percentage of the total number of available items n, while k and h are selected as a percentage of w. We report averages over computations performed for all windows while the window slides along the stream of items.

The top rows of Figures 6, 7 and 8 report the achieved diversity of GR (and equivalently BCT) and the dynamically constructed ICTs for different base values. Again, we also report the diversity achieved by RA. We observe that, even though the cover trees are dynamically constructed, and thus may differ from the corresponding BCTs for the same set of items, the achieved diversity is very close to that of GR in all cases. This means that, in practice, the performance of ICTs is better than its theoretical bound of Theorem 1.

The computational cost for using the cover trees, depicted in the bottom rows of Figures 6, 7 and 8, is the sum of two factors: (i) the cost for computing the k most diverse items at each window and (ii) the cost of removing (resp. inserting) from the cover tree the items that have exited (resp. entered) the window. We see that, in most cases, this cost is substantially smaller than the cost of computing the diverse items using GR at each window. Also, this cost increases as b decreases, since a smaller base value corresponds to taller trees, something that has an impact on the cost of insertions and deletions. In this experiment, we used the nearest-parent variations of $Insertion$ and $Deletion$, as explained in Section 4. We also experimented with the non-nearest-parent alternatives and noticed that the decrease in the quality of the solution was less than 8%. The savings in computational cost were not that important for insertions as for deletions. The cost of insertions remained roughly the same, while deletions required only around 55% of the opera-

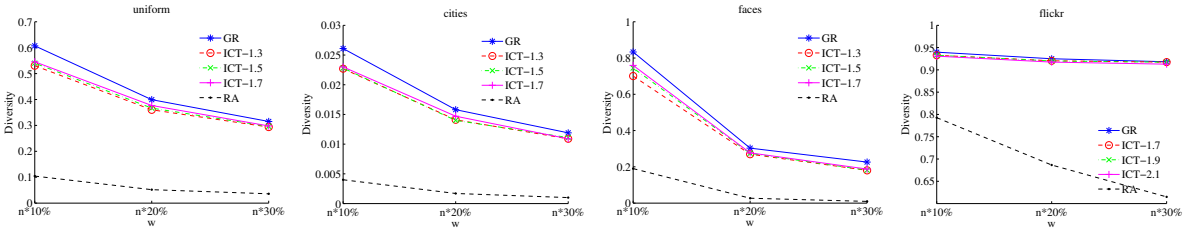


(a) *Uniform.* (b) *Cities.* (c) *Faces.* (d) *Flickr.*

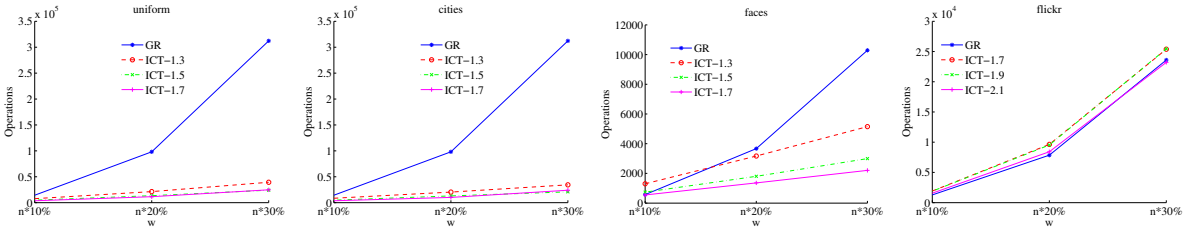


(e) *Uniform.* (f) *Cities.* (g) *Faces.* (h) *Flickr.*

Figure 6: Achieved diversity (top row) and operations (bottom row) for $w = n \cdot 20\%$ and $h = w \cdot 20\%$.

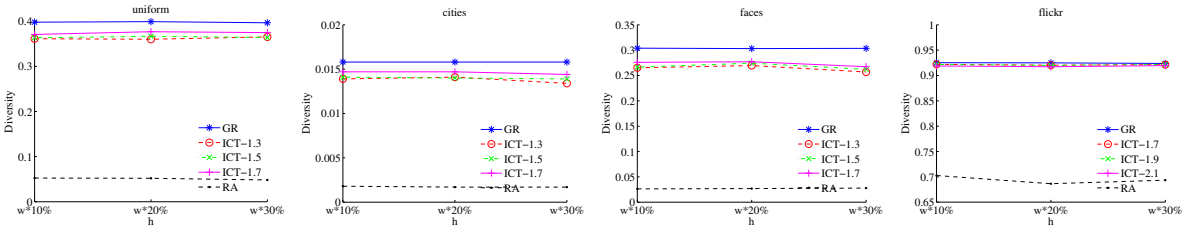


(a) *Uniform.* (b) *Cities.* (c) *Faces.* (d) *Flickr.*

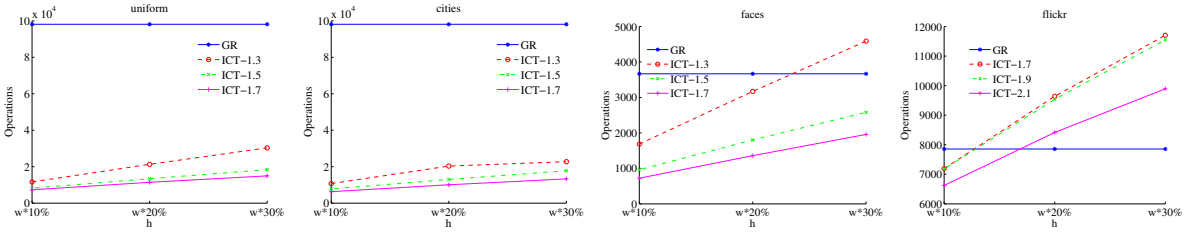


(e) *Uniform.* (f) *Cities.* (g) *Faces.* (h) *Flickr.*

Figure 7: Achieved diversity (top row) and operations (bottom row) for $k = w \cdot 15\%$ and $h = w \cdot 20\%$.

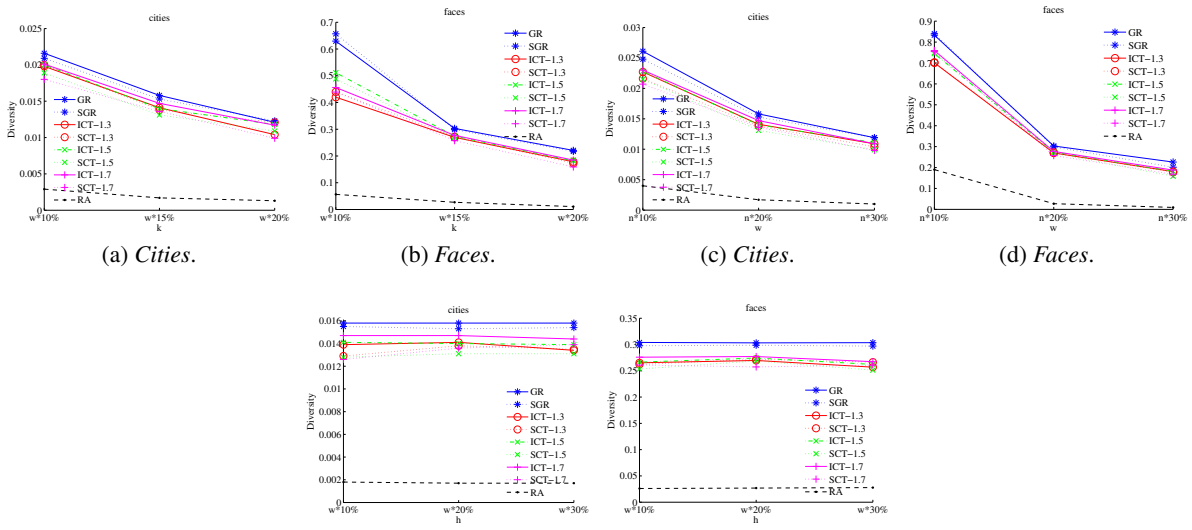


(a) *Uniform.* (b) *Cities.* (c) *Faces.* (d) *Flickr.*



(e) *Uniform.* (f) *Cities.* (g) *Faces.* (h) *Flickr.*

Figure 8: Achieved diversity (top row) and operations (bottom row) for $k = w \cdot 15\%$ and $w = n \cdot 20\%$.



(a) *Cities*. (b) *Faces*. (c) *Cities*. (d) *Faces*.

(e) *Cities*. (f) *Faces*.

Figure 9: Achieved diversity for $w = n \cdot 20\%$ and $h = w \cdot 20\%$ ((a), (b)), $k = w \cdot 15\%$ and $h = w \cdot 20\%$ ((c), (d)) and $k = w \cdot 15\%$ and $w = n \cdot 20\%$ ((e), (f)).

tions required by the nearest-parent variations. This makes the use of the simple alternatives attractive when we are willing to trade a small reduction in quality for smaller computational cost.

Concerning real time, for our Euclidean datasets and the reported b values, insertions require less than 1 millisecond for trees with 1000 items, around 2 milliseconds for trees with 5000 items and up to 2.6 milliseconds for trees with 10000 items. The corresponding times for deletions are around 1, 4 and 10 milliseconds respectively.

Finally, we perform a series of experiments to see the relation between the solutions of the UNCONSTRAINED CONTINUOUS k-DIVERSITY PROBLEM and the CONSTRAINED CONTINUOUS k-DIVERSITY PROBLEM. To this end, we adapt GR following our previous work in [10]. After each window jump, we initialize the solution for the new window with the remaining items from the previous window that were determined to be diverse, thus enforcing the durability requirement. Let S be this set of items. Then, we use GR to select $k - |S|$ other items from the new window, ignoring items that have been generated previously than the newest item in S, according to the freshness requirement. We denote this variation as Streaming Greedy (SGR) heuristic. We also modify our diverse item retrieval from the Cover Tree as discussed in Section 3 (denoted SCT).

Figure 9 shows the behavior of SGR and SCT for different base values when we vary k, w and h for "*Cities*" and "*Faces*". We omit the respective figures for the remaining datasets due to space restrictions. We notice that enforcing the durability and freshness requirements does not have a considerable impact on the achieved diversity. Both SGR and SCT achieve lower diversity than their unconstrained streaming counterparts, however, this is something we expect since the durability requirement enforces the inclusion of specific items in the solution. Concerning time, to provide some insight, retrieving the top-100 most diverse results from a cover tree with $1.3 \leq b \leq 1.7$ containing from 1000 up to 10000 items requires around 16 milliseconds, while executing GR in this case requires up to 3.2 seconds for 5000 items and more than 15 seconds for 10000 items.

6. RELATED WORK

Due to the NP-hard complexity of the k-DIVERSITY PROBLEM, a number of heuristics have been proposed in the related literature for selecting diverse subsets of items. There are plenty different flavors of diversification techniques (e.g., [23, 7, 24]). However, most works consider that the available items do not change over time and, thus, a diverse subset of items is computed once and does not evolve. Recently, a couple of works ([10, 20]) considered continuous flows of items and proposed initial approaches on incremental diversification. The authors of [22] also propose an index-based approach to diversification, which imposes certain restrictions on the form of the data and the function f.

In this section, we first present an overview of the most widely used heuristics for the static version of the problem, then existing approaches for the continuous case and, finally, existing index-based approaches.

Static Data: A number of heuristics have been proposed in the literature for solving the k-DIVERSITY PROBLEM [11], ranging from applying exhaustive algorithms to adapting traditional optimization techniques. Most heuristics that locate good solutions at a reasonable time fall in one of the following two families: *greedy* heuristics and *interchange* (or *swap*) heuristics.

Greedy heuristics make use of two sets: the initial set P of n available items and a set S which will eventually contain the selected items. Items are iteratively moved from P to S and vice versa until $|S| = k$ and $|P| = n - k$. In the most widely used variation, first, the two furthest apart items of P are added to S. Then, at each iteration, one more item is added to S. The item that is added is the one that has the maximum distance from S. The complexity in terms of computed distances is $O(n^2)$.

Interchange heuristics are initialized with a random solution S of size k and then iteratively attempt to improve that solution by interchanging an item in the solution with another item that is not in the solution. The item that is eliminated from the solution at each iteration is one of the two closest items in it. There are two main variations: either perform at each iteration the first interchange that improves the solution or consider all possible interchanges and perform the one that improves the solution the most. None of the two variations clearly outperforms the other, while their worst case complexity is $O(n^k)$. Even though there is no clear winner in terms

of complexity, the first variation usually locates better solutions [14]. Recently, [23] introduced randomization steps in interchange heuristics by allowing interchanges that mat not improve the solution at the current step but may lead to allowing better interchanges in the future.

Continuous Data: In our previous work [10], we considered adapting the greedy heuristic to the continuous case. After each window jump, we initialize the solution with any valid items that were selected as diverse in the previous window and then we allow the greedy heuristic to select the remaining items to "fill in" the new solution. As other greedy approaches, this heuristic still requires many distances to be computed in order to locate a solution for the problem.

Recently, the authors of [20] considered an incremental solution based on interchange heuristics. Upon the arrival of a new item p, all possible interchanges between p and the items in the current diverse subset are performed. If there exists some interchange that increases diversity, then the corresponding two items are swapped and p enters the diverse result. A similar technique was also proposed in [12]. Two possible drawbacks of this approach is that old items may never leave the diverse results in case no swaps are performed and also that a new object can enter the diverse set only upon its arrival and not later in time.

Indices for Diversification: The only existing works to the best of our knowledge that make use of indices to assist result diversification are [22] and [18]. [22] aims at selecting diverse tuples of a *structured* relation, where the attributes of the relation follow a *total order* of importance concerning diversity. That means that two tuples that differ in a highly important attribute are considered very different from each other, even if they share common values in other less important attributes. This distance measure allows the exploitation of a Dewey encoding of the tuples that enables them to be organized in a tree structure which is later exploited to select the k most diverse of them. We also employ tree structures in our approach. However, our definition of diversity is more general and does not demand structured data or a specific ordering of some features. In [18], the authors employ cover trees for solving the k-medoids problem. While selecting k representative medoids is a form of diversification, that work focuses on the clustering of data rather than their diversification. Recently, cover trees were also employed in [9] for computing priority medoids, i.e., medoids that are associated with some high relevance factor. Our work differs in the aspect that priority medoid computation cannot be employed in dynamic environments, since it depends on the order of item insertions in the trees.

7. SUMMARY

Recently, result diversification has attracted considerable attention. However, most current research addresses the static version of the problem. In this paper, we have studied the diversification problem in a dynamic setting where the items to be diversified change over time. We have proposed an index-based approach that allows the incremental evaluation of the diversified sets to reflect item updates. Our solution is based on cover trees. We have provided theoretical and experimental results regarding the quality of our solution.

Acknowledgments

The first author is supported by the research program "HRAKLEITOS II", co-funded by the European Union and the Hellenic Ministry of Education, Life Long Learning and Religious Affairs.

We would like to thank the authors of [23] for providing us with their version of the "*Faces*" dataset.

8. REFERENCES

[1] Faces dataset. http://www.informedia.cs.cmu.edu.
[2] Flickr. http://www.flickr.com/.
[3] Flickr dataset. Available at http://www.tagora-project.eu.
[4] Forest cover dataset. Available at http://kdd.ics.uci.edu.
[5] Greek cities dataset. Available at http://www.rtreeportal.org.
[6] Twitter. http://www.twitter.com.
[7] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *WSDM*, 2009.
[8] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML*, 2006.
[9] R. Boim, T. Milo, and S. Novgorodov. Diversification and refinement in collaborative filtering recommender. In *CIKM*, 2011.
[10] M. Drosou and E. Pitoura. Diversity over continuous data. *IEEE Data Eng. Bull.*, 32(4):49–56, 2009.
[11] M. Drosou and E. Pitoura. Search result diversification. *SIGMOD Record*, 39(1):41–47, 2010.
[12] M. Drosou, K. Stefanidis, and E. Pitoura. Preference-aware publish/subscribe delivery with diversity. In *DEBS*, 2009.
[13] E. Erkut. The discrete *p*-dispersion problem. *European Journal of Operational Research*, 46(1), 1990.
[14] E. Erkut, Y. Ülküsal, and O. Yeniçerioglu. A comparison of *p*-dispersion heuristics. *Computers & OR*, 21(10), 1994.
[15] S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, 2009.
[16] J. R. Haritsa. The kndn problem: A quest for unity in diversity. *IEEE Data Eng. Bull.*, 32(4):15–22, 2009.
[17] T. Kollar. Fast Nearest Neighbors. http://nicksgroup.csail.mit.edu/TK/Technical_Reports/covertrees.pdf.
[18] B. Liu and H. V. Jagadish. Using trees to depict a forest. *PVLDB*, 2(1):133–144, 2009.
[19] Z. Liu, P. Sun, and Y. Chen. Structured search result differentiation. *PVLDB*, 2(1), 2009.
[20] E. Minack, W. Siberski, and W. Nejdl. Incremental diversification for very large sets: a streaming-based approach. In *SIGIR*, 2011.
[21] A. Tamir. Obnoxious facility location on graphs. *SIAM J. Discrete Math.*, 4(4):550–567, 1991.
[22] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia. Efficient computation of diverse query results. In *ICDE*, 2008.
[23] M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. T. Jr., and V. J. Tsotras. On query result diversification. In *ICDE*, 2011.
[24] C. Yu, L. V. S. Lakshmanan, and S. Amer-Yahia. It takes variety to make a world: diversification in recommender systems. In *EDBT*, 2009.
[25] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving recommendation lists through topic diversification. In *WWW*, 2005.