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. Most previous research, however, considers that the available results remain unchanged. In this paper, we focus on achieving content diversity in the case of continuous data delivery, such as in the context of notification services. We define the Continuous k-Diversity Problem along with appropriate constraints that enforce continuity requirements on the diversified results. We propose an efficient solution for the problem that exploits indexing to handle dynamic insertions and deletions. We present experimental results concerning the efficiency and effectiveness of our approach.

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 [12].

Previous approaches to result diversification can be roughly divided into those employing greedy and interchange heuristics for computing solutions. Greedy heuristics (e.g. [26, 16]) build a diverse set incrementally, selecting one item at a time so that some distance function is maximized, whereas interchange-based heuristics (e.g. [25, 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 [23], a Dewey-based tree is used for structural diversity based on attribute priorities and in [17], 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 on 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 [11], we have experimented with greedy heuristics, while recently [20] applied an interchange heuristic to incrementally build a diverse set for a stream of items.

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 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 log in 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-neighborhood search [9]. They were recently used to compute medoids [7] and priority medoids [10] 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 [14].

We provide theoretical results for the accuracy of the solution achieved using cover trees. We also introduce a batch construction 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, while it can also support multiple queries with different values of $k$.

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,
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 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 the diversity exhibited by a set of items. Here, we focus on content diversity, i.e. selecting a subset of the available items having dissimilar content. 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 \( \mathcal{P} = \{ p_1, \ldots, p_n \} \) be a set of \( n \) items and \( k \) a positive integer, \( k \leq n \). Let also \( d : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{R}^+ \) be a distance metric indicating the dissimilarity exhibited by two items in \( \mathcal{P} \). The diversity of a set \( \mathcal{S}, \mathcal{S} \subseteq \mathcal{P} \), is measured by a function \( f : 2^{\mathbb{P} \times \mathbb{P}} \rightarrow \mathbb{R}^+ \) which takes into account the dissimilarity of the items in \( \mathcal{S} \) 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 \( \mathcal{P} \) that exhibit the largest diversity among all possible combinations of items. Formally:

**Definition 1** (\( k \)-DIVERSITY PROBLEM). Let \( \mathcal{P} \) 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 \( \mathcal{S}^* \) of \( \mathcal{P} \) such that:

\[
\mathcal{S}^* = \arg \max_{\mathcal{S} \subseteq \mathcal{P}} f(\mathcal{S}, d) \quad \text{subject to} \quad |\mathcal{S}| = k.
\]

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 \( f_{\text{MAXMIN}} \) and \( f_{\text{MAXSUM}} \) respectively:

\[
f_{\text{MAXMIN}}(S, d) = \min_{p_i, p_j \in S} d(p_i, p_j)
\]

and

\[
f_{\text{MAXSUM}}(S, d) = \sum_{p_i, p_j \in S} d(p_i, p_j)
\]

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 \( \mathcal{P} \).

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

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

The MAXMIN Greedy Heuristic.

The \( k \)-DIVERSITY PROBLEM has been shown to be NP-hard [14]. Various heuristics have been proposed, among which a natural Greedy heuristic (Algorithm 1) has been shown experimentally to outperform the others in most cases [11, 15]. The algorithm starts by selecting the two items from \( \mathcal{P} \) 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 points \( \mathcal{S} \) is defined as:

\[
d(p, \mathcal{S}) = \min_{p_i \in \mathcal{S}} d(p, p_i)
\]

It has been shown (e.g. in [22]) that the solution provided by the Greedy heuristic is a \( \frac{1}{2} \)-approximation of the optimal solution.

**Algorithm 1** MAXMIN Greedy Heuristic

**Input:** A set of items \( \mathcal{P} \), an integer \( k \).

**Output:** A set \( \mathcal{S} \) with \( k \) most diverse items of \( \mathcal{P} \).

1: \( p^*, q^* \leftarrow \arg \max_{p, q \in \mathcal{P}} d(p, q) \)
2: \( \mathcal{S} \leftarrow \{p^*, q^*\} \)
3: while \( |\mathcal{S}| < k \) do
4: \( p^* \leftarrow \arg \max_{p \in \mathcal{P} \setminus \mathcal{S}} d(p, \mathcal{S}) \)
5: \( \mathcal{S} \leftarrow \mathcal{S} \cup \{p^*\} \)
6: end while
7: return \( \mathcal{S} \)

2.2 The Continuous \( k \)-Diversity Problem

We consider applications where new items are generated in a continuous manner and, thus, the set \( \mathcal{P} \) of available items changes
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 \textit{durability} requirement.

Second, we want the order in which items are chosen to be diverse to follow the order they appear in the stream. We call this the \textit{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.

Based on the above observations, we formally define the \textit{Constrained Continuous \(k\)-Diversity Problem} as follows:

\textbf{Definition 2. (Constrained Continuous \(k\)-Diversity Problem).} Let \(P\) be a stream of items and \(P_{t-1}\), \(P_t\) be two consequent 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 \textit{Constrained Continuous \(k\)-Diversity Problem} is to select a subset \(S_t^*\) of \(P_t\), for each \(P_t\), such that:

\[
S_t^* = \arg\max_{S_t \subseteq P_t} f(S_t, d) \\
|S_t| = k
\]

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

(i) \(\forall p_j \in S_{t-1}^* \cap P_t \Rightarrow p_j \in S_t^*\) (durability requirement),

(ii) Let \(p_t\) be the newest item in \(S_{t-1}^*\). Then, \(\exists p_j \in P_t \setminus S_{t-1}^*\)

with \(j < t\), such that, \(p_j \in S_t^*\) (freshness requirement).

Note that, the \(k\) most diverse items for a user are completely renewed after the generation of \(w\) new items.

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 and guide us through the selection of diverse subsets of items. Our approach is based on the cover tree, defined next.

3.1 The Cover Tree

The Cover Tree (CT) [9] 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 \(\ell \in (\infty, \infty)\), \(\ell\) 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 are far apart from each other. 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 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 \(C_{\ell}\) be the set of nodes at level \(\ell\) of a cover tree. The cover tree of base \(b\), \(b > 1\), obeys the following invariants for all \(\ell\):

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

2. \textbf{Covering:} For every \(p_t \in C_{\ell-1}\), 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. \textbf{Separation:} For all distinct \(p_i, p_j \in C_{\ell}\), it holds that, \(d(p_i, p_j) > b^\ell\).

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 \(\ell\) of the tree, then \(p\) is a child of itself in all levels below \(\ell\).
However, such 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). While the 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, the explicit representation of a cover tree for a set of 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, the explicit representation of a cover tree for a set of points in the 2-dimensional space.

We make the following observation which is important for diversification: items indexed at sibling nodes of higher levels of a cover tree are further apart from each other than those indexed 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 $\ell$ be the highest level that 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 $\ell$, 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 shows the $k$ most diverse items selection procedure. We start from the root node of the tree and proceed to select nodes (i.e., items) in the order they are found in the tree. Due to the nesting invariant, this can be done efficiently. Since $C_1 \subseteq C_{\ell-1}$, we descend the tree until we reach the first level $\ell$ for which $|C_1| \leq k$ and $|C_{\ell-1}| > k$. All items of $C_1$ are added to the solution. The remaining $k - |C_1|$ items are selected from $C_{\ell-1}$ in a greedy fashion (lines 7-11).

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

The two requirements of Definition 2 can be easily implemented using the cover tree. For the durability requirement, items that are selected as diverse are marked so and remain part of the diversity set until they expire. Let $m$ be the number of such items. Algorithm 2 just selects $k - m$ 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 in any of the diverse sets.

#### 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 diverse algorithm on a cover tree for $P$ (Algorithm 2). It holds that $d^{CT}(P, k) \geq d^{OPT}(P, k)$, where $\alpha = \frac{1}{2}\sqrt{\pi^2}$.

**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 of any pair of items $p_1, p_2 \in S^{OPT}(P, k)$ appears in the cover tree. Assume that the least common ancestor of any $p_1, p_2 \in S^{OPT}(P, k)$ is at level $l$. This would mean that there are at least $k$ nodes at level $l - 1$ (i.e. the distinct ancestors of the $k$ items in $S^{OPT}(P, k)$), thus, the cover tree algorithm would stop at level $m - 1$ in the worst case.

Let us now compute a bound on $m$. Assume that the least common ancestor 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, $d(p_1, p) + (p_2, p) \geq d(p_1, p_2)$. Since $p_1, p_2 \in S^{OPT}(P, k)$, it holds that $d(p_1, p_2) \geq d^{OPT}(P, k)$. Thus:

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

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

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

This means that the CT algorithm will select items from this or a higher level. From the separation invariant of the cover tree, we have $d^{CT}(P, k) \geq \frac{b^{m-1}}{b^{1/2}}$. Using (2), we get that $d^{CT}(P, k) \geq \frac{b^{1/2}d^{OPT}(P, k)}{2} - 2$ $\Rightarrow$ $d^{CT}(P, k) \geq \frac{b^{1/2}d^{OPT}(P, k)}{2}$, which proves the theorem.

#### 3.4 Batch Construction

Given a set of items $P$, there may be many different cover trees that maintain the three invariants. Next, we present a batch construction of a cover tree for $P$ appropriate for the MAXMIN problem. The tree is built bottom-up. The algorithm proceeds greedily...
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: $\ell \leftarrow \left\lceil \log_b \left( \min_{p,q \in P} d(p,q) \right) \right\rceil$
2: $Q_\ell \leftarrow \emptyset$
3: for all $p \in P$ do
4: \hspace{1em} $Q_\ell \leftarrow Q_\ell \cup \{p\}$
5: end for
6: while $|T.C_\ell| > 1$ do
7: \hspace{1em} $T.C_{\ell+1} \leftarrow \emptyset$
8: \hspace{1em} Candidates $\leftarrow T.C_\ell$
9: \hspace{1em} $p^*, q^* \leftarrow \arg\max_{p,q \in \text{Candidates}} d(p,q)$
10: \hspace{1em} $T.C_{\ell+1} \leftarrow T.C_{\ell+1} \cup \{p^*, q^*\}$
11: \hspace{1em} Candidates $\leftarrow \text{Candidates}\setminus\{p^*, q^*\}$
12: end while
13: \hspace{1em} Candidates $\leftarrow \text{Candidates}\setminus\{p : \exists q \in T.C_{\ell+1} \text{ with } d(p,q) \leq b^{\ell+1}\}$
14: \hspace{1em} $p^* \leftarrow \arg\max_{p \in \text{Candidates}} d(p,T.C_{\ell+1})$
15: \hspace{1em} $T.C_{\ell+1} \leftarrow T.C_{\ell+1} \cup \{p^*\}$
16: \hspace{1em} Candidates $\leftarrow \text{Candidates}\setminus\{p^*\}$
17: end while
18: for all $p \in T.C_\ell$ do
19: \hspace{1em} $q^* \leftarrow \arg\min_{q \in T.C_{\ell+1}} d(p,T.C_{\ell+1})$
20: \hspace{1em} make $q$ parent of $p$
21: end for
22: $T.C_\ell \leftarrow T.C_{\ell+1}$
23: $\ell \leftarrow \ell + 1$
24: end while
25: return $T.C_\ell$

by promoting to higher levels the items with the largest possible distance with 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 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 $C_\ell$ 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 $C_\ell$ 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 the subtrees of each parent node.

We shall prove that the items at each level $\ell$ 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,

$$\forall \ell \leq T. \text{C}_\ell = S^{GR}(P,|\text{C}_\ell|)$$

where $C_\ell$ is the set of items at level $\ell$ of $T$.

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

Consider the construction of level $\ell + 1$. From the induction step, it holds $C_\ell = S^{GR}(P,|C_\ell|)$. Let $p$ be the first item in $C_\ell$ that 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 $C' \subseteq C_\ell$, 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 $p' \in C \setminus C'$ (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 $p' \in C \setminus C'$, $\exists q \in C'$ 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 $C_\ell$ are already covered. Therefore, to construct $C_{\ell+1}$, the items from $C_\ell$ to be included in level $\ell + 1$ are considered in the same order as with 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. $\square$

Note that, here we made an 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 in a deterministic order. We assume that, in case ties are present, these are resolved not arbitrarily but in some specific order that may vary according to the nature of the items.

Regarding the complexity of Algorithm 3, note that most computational steps are shared among levels, since the construction of each level start by the two furthest apart items of the dataset and the rest of the items are selected in a greedy way. Therefore, these computational steps can be performed only once (at the lower level). Then, it suffices to maintain the order at which each item was selected from the lowest level for promotion to the next one. Each level $C_{\ell+1}$ is a subset of $C_\ell$ and, more specifically, consists of the items of $C_\ell$ in the order that were inserted into $C_\ell$ until the first item whose minimum distance from the already selected items of $C_\ell$ at the point of its insertion is smaller than $b^{\ell+1}$.

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.

### 3.5 Changing $k$

The cover tree can be used to compute multiple queries with different $k$. Thus, each user can individually tune the amount of
Algorithm 4 Insert

Input: An item \( p \), a set of nodes \( T.Q_L \) of a cover tree \( T \) at level \( \ell \).

Output: A cover tree \( T \).

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

4. DYNAMIC CONSTRUCTION

In this section, we present insertion and deletion algorithms for a cover tree of base \( b \). We also describe algorithms for supporting constrained continuous diversity, as defined in Section 2.

4.1 Incremental Insertion

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 \( P \) incrementally as new items arrive and old ones expire.

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 [9] and subsequent corrections in [18]. Our implementation of the algorithm takes as input the new item \( p \) and a set of candidate nodes \( Q_\ell \) at level \( \ell \) under which the new item could be inserted.

Next, we prove the correctness of the algorithm for \( b > 1 \).

THEOREM 3. Algorithm 4 with input an item \( p \) and the root level \( C_\infty \) at level \( \ell \) 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 \( \ell \) can range from \(+\infty\) to \(-\infty\), there is always a (sufficiently low) level 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 \( d(p, Q_1) \leq b^\ell \). 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 covering 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. \( d(p, q') > b^{\ell-1} \). Using the triangle inequality, we have that \( d(p, q) > d(p, q') - d(q, q') = d(p, q') - \sum_{i=1}^{\ell-1} b^i = d(p, q') - \left(b^{\ell-1} - b^{\ell'} \right) > b^{\ell-1} + b^{\ell'-1} = b^{\ell'} > b^{\ell-1} \). 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 be able to retrieve better diverse items in the future.

For clarity of presentation, Algorithm 4 assumes that the new item \( p \) can be inserted in some existing level of the tree. In case 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 greater distance from the root node than \( b^{\text{max}} \), where \( b^{\text{max}} \) 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 level lower than \( b^{\text{min}} \), where \( b^{\text{min}} \) is the minimum level of the tree, we copy all nodes of \( C_\text{max} \) to a new level \( C_{b^{\text{max}}-1} \) 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.

4.2 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 can 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 \( \ell' \) and continue to process them in decreasing order of their distance to the items currently in \( \ell' \). These heuristics also do not affect the correctness of the algorithm.

THEOREM 4. Algorithm 5 with input an item \( p \) and the root level \( C_\infty \) at level \( \ell \) of a cover tree \( T \) of base \( b \) for \( P \) returns a cover tree of base \( b \) for \( P - \{ p \} \).

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 \( \ell' \) 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 \( q \).
Algorithm 5 Delete

Input: An item \( p \), sets of nodes \( \{T.Q_0, T.Q_{b_1}, \ldots, T.Q_{b_\ell}\} \) of a cover tree \( T \) at level \( \ell \)

Output: A cover tree \( T \).

1: \( C \leftarrow \{ \text{children}(q) : q \in T.Q_0 \} \)
2: \( T.Q_{\ell-1} \leftarrow \{ q \in C : d(p, q) \leq \frac{b'}{1+\ell} \} \)
3: Delete\( p, \{T.Q_{\ell-1}, T.Q_0, \ldots, T.Q_{b_\ell}\}, \ell - 1 \)
4: \( \text{if } d(p, C) = 0 \text{ then} \)
5: \( p \leftarrow \text{delete } p \text{ from level } \ell - 1 \text{ and from Children(Parent(p))} \)
6: \( T.Q_{\ell-1} \leftarrow \{ q \in C : d(p, q) \leq \frac{b'}{1+\ell} \} \)
7: \( \ell' \leftarrow \ell - 1 \)
8: \( \text{while } d(q, T.Q_{\ell'}) > b' \text{ do} \)
9: \( \text{add } q \text{ into level } \ell' \)
10: \( T.Q_{\ell'} \leftarrow T.Q_{\ell'} \cup \{q\} \)
11: \( \ell' \leftarrow \ell' + 1 \)
12: \( \text{end while} \)
13: \( q^* \leftarrow \arg\min_{q' \in T.Q_{\ell'}} d(p', q) \)
14: \( \text{make } q \text{ a child of } q^* \)
15: \( \text{end for} \)
16: \( \text{end if} \)

As in the case of insertions, we also adjust \( \ell_{\text{max}} \) for the tree after each deletion (\( \ell_{\text{max}} \) 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 the 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.

### 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. First, we concentrate on the behavior of the Cover Tree for static data and then we examine the dynamic case.

#### 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 the range \([0, 1]\). Items are either uniformly distributed in space (“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 [3]. 10 features are present for 5000 locations. The third one (“Faces”) consists of 256 features extracted from each of 300 human face images with the eigen-faces method [4]. Finally, for our last real dataset (“Flickr”), we used data from [2] which consists of tags assigned by users to photographs uploaded to the Flickr [1] photo service 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 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 distances among the sets of tags describing each photograph. The characteristics of our datasets are summarized in Table 1.

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

#### 5.2 Batch Cover Tree

First, we report findings concerning the construction of the Batch Cover Tree (denoted BCT). Diversity-wise, the solutions produced by the BCT are identical to those of the Greedy heuristic (denoted GR). GR is generally one of the best performing heuristics for computing diverse results [11]. Due to the NP-hardness of the \(k\)-DIVERSITY PROBLEM, it is not possible to compute optimal solutions in a relatively short time even for moderate values of \( n \) and \( k \). However, we know that both GR, and thus, BCT produce \( \frac{1}{2} \)- approximations of the optimal solution. Therefore, we focus our evaluation on the computational cost of building a Batch Cover Tree. We choose to measure this cost in terms of operations, i.e. distance computations required to be calculated, instead of other measures such as time. The cost of a single distance computation may vary a lot depending on the distance metric used. For example, computing Jaccard distances between sets of items is more expensive, and thus time-consuming, than computing Euclidean distances. We use 1000 items for all datasets except from “Faces” where there are only 300 available items and “Flickr” where the distance computations are 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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Cardinality</th>
<th>Dimensions</th>
<th>Distance metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>10000</td>
<td>2</td>
<td>Euclidean</td>
</tr>
<tr>
<td>Clustered</td>
<td>10000</td>
<td>2</td>
<td>Euclidean</td>
</tr>
<tr>
<td>Cities</td>
<td>5922</td>
<td>2</td>
<td>Euclidean</td>
</tr>
<tr>
<td>Forest</td>
<td>5000</td>
<td>10</td>
<td>Cosine</td>
</tr>
<tr>
<td>Faces</td>
<td>300</td>
<td>256</td>
<td>Cosine</td>
</tr>
<tr>
<td>Flickr</td>
<td>18245</td>
<td>-</td>
<td>Jaccard</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3: Height of produced Batch Cover Trees.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum level.</td>
</tr>
<tr>
<td>( b )</td>
</tr>
<tr>
<td>1.3</td>
</tr>
<tr>
<td>1.5</td>
</tr>
<tr>
<td>1.7</td>
</tr>
<tr>
<td>Minimum level.</td>
</tr>
<tr>
<td>( b )</td>
</tr>
<tr>
<td>1.3</td>
</tr>
<tr>
<td>1.5</td>
</tr>
<tr>
<td>1.7</td>
</tr>
</tbody>
</table>
to build the first non-leaf level and then (ii) also 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 required by the GR heuristic. Therefore, the actual extra cost of building a BCT is reflected by the operations of step (ii). The amount of these operations differs when out nearest-parent heuristic (denoted "np") is employed. Table 2 shows the required operations to build the most diverse items once, employing GR would be preferable. However, building a BCT comes at little extra cost and can later be used for dynamic insertions and deletions of items, as we will see in the next section.

Next, we concentrate on how our approach performs when items from a BCT as opposed to retrieving them employing the GR heuristic. The corresponding numbers are shown in Figure 4 for varying \( k \), where \( k \) is chosen as a percentage of the total number of available items \( n \). We see that retrieving diverse items from an already constructed BCT has much smaller cost that 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 \( \ell \) 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 “Faces” is around one third the size of “Forest”.

Finally, we also compare the diversity achieved by a non-Batch Cover Tree (denoted CT) to that of GR, and thus BCT as well (Figure 5). To construct such trees, we used our dynamic versions of Insert and Delete to insert all items of the datasets to the trees. For comparison reasons, we also include the diversity of a random subset of \( k \) items (denoted RA). We see that even non-Batch Cover Trees achieve very good solutions, while requiring only a small percentage of the operations in Table 2 to be built.

### 5.3 Dynamic Construction of the Cover Tree

Next, we concentrate on how our approach performs when items are dynamically inserted and deleted from the Cover Tree. Fig-

---

<table>
<thead>
<tr>
<th>[ b ]</th>
<th>uniform (1000 items)</th>
<th>clustered (1000 items)</th>
<th>cities (1000 items)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ step (i) ]</td>
<td>[ step (ii) ]</td>
<td>[ step (ii) - np ]</td>
<td>[ step (i) ]</td>
</tr>
<tr>
<td>[ 1.5 ]</td>
<td>166347120</td>
<td>908946</td>
<td>66138</td>
</tr>
<tr>
<td>[ 1.3 ]</td>
<td>166347120</td>
<td>908946</td>
<td>66138</td>
</tr>
<tr>
<td>[ 1.1 ]</td>
<td>166347120</td>
<td>908946</td>
<td>66138</td>
</tr>
</tbody>
</table>

Table 2: Operations required to build the Batch Cover Tree.

**Figure 4:** Total number of operations for retrieving the \( k \) most diverse items for GR and BCT with different base values.
We are willing to trade a small reduction in quality for smaller computational cost.

5.4 Continuous top-$k$ Selection

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 the GR heuristic following our previous work in [11]: After each window jump, we initialize the solution for the new window with the remaining items from the previous windows 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, not taking into consideration 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 form the Cover Tree as discussed in Section 4 (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 non-streaming counterparts, however, this is something we expect since the durability requirement enforces the inclusion of specific items in the solution.

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. [24, 8, 25]). 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 over time. Recently, a couple of works ([11, 21]) considered continuous flows of items and proposed initial approaches on incremental diversification. The authors of [23] also propose an index-based approach to diversification, which imposes certain
Figure 6: Achieved diversity (top row) and operations (bottom row) for $w = n \cdot 20\%$ and $h = w \cdot 20\%$.

Figure 7: Achieved diversity (top row) and operations (bottom row) for $k = w \cdot 15\%$ and $h = w \cdot 20\%$. 
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 [12], 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 [15]. Recently, [24] 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 [11], 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 Greedy to select the remaining items to “fill in” the new solution. We call this heuristic Streaming Greedy. 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 [21] 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 [13]. 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 [23] and [7]. [23] 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 [7], 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 [10] 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.

8. REFERENCES