Static-to-dynamic transformation for metric indexing structures

Bilegsaikhan Naidan, Magnus Lie Hetland

Department of Computer and Information Science,
Norwegian University of Science and Technology,
Sem Sælands vei 7-9, NO-7491 Trondheim, Norway

Abstract

In this paper, we study the well-known algorithm of Bentley and Saxe in the context of similarity search in metric spaces. We apply the algorithm to existing static metric index structures, obtaining dynamic ones. We show that the overhead of the Bentley-Saxe method is quite low, and because it facilitates the dynamic use of any state-of-the-art static index method, we can achieve results comparable to, or even surpassing, existing dynamic methods. Another important contribution of our approach is that it is very simple—an important practical consideration. Rather than dealing with the complexities of dynamic tree structures, for example, the core index can be built statically, with full knowledge of its data set.

Keywords: similarity search, static and dynamic indexes, Bentley-Saxe algorithm, experiments.

1. Introduction

Many modern applications require efficient similarity retrieval, including applications in multimedia (to find similar images, audio in digital-repositories), pattern recognition (to identify finger-prints, face images in image databases), and string searching (to find words in a dictionary while permitting spelling errors). In such applications, the search problem is often stated in terms of distance-search in a metric space. That is, given a metric d over a universe $\mathbb{U}$, and a data set $\mathbb{D} \subset \mathbb{U}$, find the objects in $\mathbb{D}$ that are closest to some query $q \in \mathbb{D}$ (either all within a search radius r, or the k nearest neighbors, kNN).

Rather than performing a linear scan of the full data set, it is common to preprocess the data set by building an index structure, exploiting the metric axioms (the triangular inequality in particular). Most existing such index structures are *static*.[1] That is, the index is built with access to the full data set, and if an object is to be added or deleted, a full rebuild of the entire index is required. Such rebuilding is, of course, time consuming and computationally intensive. To accommodate insertions and deletions, some special-purpose *dynamic* index structures, supporting additions and deletions at low cost, have been proposed. Maintaining the integrity and performance of a dynamic structure over time, with only incremental information, can be challenging; such structures can be more complicated, as well as less able to utilize global information about the data set.

In this paper, we study the Bentley-Saxe [1] algorithm in the context of similarity search in metric spaces. The Bentley-Saxe method is a tool that allows us to transform a static data structure into a dynamic one for any decomposable search problem (as explained in Section 3). This means that we can still use the state of the art in static indexing, even if we need the functionality of a dynamic indexing method, without losing the ability to globally analyze the data set, and without adding any appreciable complexity. In fact, the Bentley-Saxe method can use the indexing methods as black-box modules, permitting a clean separation of the (static) indexing and the dynamism.

This paper is organized as follows. Section 2 describes some related work. The Bentley-Saxe method is explained in Section 3. Section 4 provides our experimental results. Some concluding

Email addresses: bileg@idi.ntnu.no (Bilegsaikhan Naidan), mlh@idi.ntnu.no (Magnus Lie Hetland)

[1] Based on an analysis of the proceedings of the International Workshop on Similarity Search and Applications.

remarks are given in Section 5.

2. Related Work

In this section we briefly overview some relevant static and dynamic metric indexing structures. For further details, refer to the tutorial by Hetland [8] and the book by Zezula et al. [16]. We consider two well-known static methods (the VP-tree and the SSS-tree) as well as two dynamic ones (EGNAT and the DSA-tree).

The vantage point (VP) tree [15] is a static balanced binary tree. The construction algorithm for the VP-tree first selects a representative object p (a so-called *vantage point*) from the dataset $\mathbb{D}$ and computes the median m of the distances between p and the other objects in the dataset. Then it divides the dataset into two subsets $\mathbb{D}_1$ and $\mathbb{D}_2$ such that $\mathbb{D}_1 = \{x \in \mathbb{D} \mid x \neq p, d(p,x) \leq m\}$ and $\mathbb{D}_2 = \mathbb{D} \setminus (\mathbb{D}_1 \cup \{p\})$. The algorithm recursively builds left and right subtrees for $\mathbb{D}_1$ and $\mathbb{D}_2$, if they are not empty. A range query q with radius r is performed by recursively traversing the tree from the root to leaves. For each visited node, $d(q,p)$ is computed and p is reported if $d(q,p) \leq r$. It is necessary to traverse the left subtree only if $d(q,p) - r \leq m$, and, similarly, the right subtree only if $d(q,p) + r \geq m$.

There exists a dynamic version of the VP-tree [7]. However, it is not at all straightforward to implement correctly, and in some cases it is still unable to avoid periodic reconstruction of subtrees or even of the entire tree.

Brisaboa et al. have proposed a static index structure called the Sparse Spatial Selection (SSS) tree [3], in which the first object in a dataset is selected as the first cluster center and then the rest of the objects become new cluster centers if they are far enough away from all current centers (i.e., the minimum distance between the object and current cluster centers is greater than αM, where α is an user-defined parameter and M is the maximum distance between any two objects); otherwise, they are assigned to the cluster associated with the nearest center. The process is recursively applied to those clusters that have not yet fallen below a given size threshold.

The Geometric Near-neighbor Access Tree (GNAT) [2] is a multiway static tree and is built as follows. First, a set of pivots are selected at random and then the rest of the objects are assigned to a region associated with the closest pivot. Examples of a GNAT are shown in Figure 1a, 1b. For each region, the minimum and maximum distances to the other regions' objects are kept for efficiently filtering out non-promising regions in the search, meaning that a region is discarded if the query ball does not intersects with this distance interval. The subtrees are recursively built for all regions associated with the pivots.

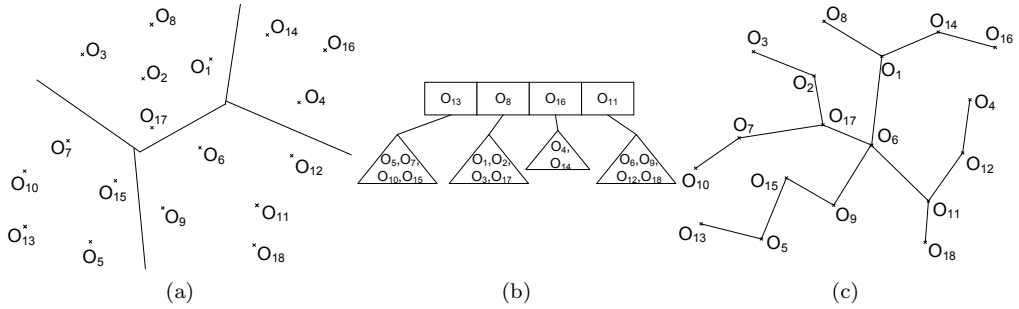
The Evolutionary GNAT (EGNAT) [14] is a dynamic version of GNAT. The root is initially created as a leaf node. The insertion algorithm traverses the index structure by choosing the subtree associated with the closest pivot until a leaf node is reached. If the leaf node has a room for the new object, it is added there. Otherwise, the leaf node is transformed into an internal node by selecting pivots and distributing its objects into new child (leaf) nodes. The leaf nodes also keep information about distances to their parent objects. During the search this information is used to establish lower bounds to the actual distances between the query and objects.

The spatial approximation (SA) tree [9] is based on an approach that is, at least superficially, quite different from the hierarchical space decomposition of the other trees. First, an arbitrary object is selected as the root of the tree and a set of its neighbors is selected as follows. An object is inserted in the neighbor list if it is closer to the root than all current neighbors. Otherwise, the object is assigned to a subset associated with its closest neighbor. Then, for each subset the procedure is applied recursively. Figure 1c shows an example of a SA-tree. The search algorithm uses a best-first branch-and-bound approach, similar to that used by most metric tree structures.

Navarro et al. [11] have shown that the SA-tree can be built dynamically, and they call the resulting structure the dynamic SA (DSA) tree. They manage to preserve the semantics of the SA-tree by introducing a *time-stamp* for every object. These time-stamps are then used during search, to ensure that only distance relationships that were known at the time of insertion are used when filtering out objects, to avoid false dismissals.

3. The Bentley and Saxe algorithm

We call a search problem *decomposable* if, for any pair of data sets $\mathbb{D}_1$ and $\mathbb{D} \setminus \mathbb{D}_1$, the answer to a query over $\mathbb{D}$ can be computed efficiently from the



(a) (b) (c)

Figure 1: Examples of (a) a GNAT space decomposition with hyperplanes between O_8, O_{11}, O_{13} and O_{16}, (b) the corresponding GNAT tree, and (c) a SA-tree with the root O_6.

answers to queries for each of $\mathbb{D}_1$ and $\mathbb{D} \setminus \mathbb{D}_1$. The Bentley-Saxe algorithm (BS) exploits this sort of decomposition to reduce the size of the structures that need to be rebuilt, on average (i.e., amortized), when inserting or deleting objects.

The main data structure of BS is a set of $m = \lfloor \log_2 n \rfloor + 1$ buckets[2] $B_0, B_1, \ldots, B_{m-1}$ and each bucket B_i is either empty or a static data structure that contains a collection of 2^i objects. To insert a new object into the index, the algorithm follows the same principle that is used for incrementing a binary counter, where the ith bit denotes the absence or presence of a static index structure in the bucket B_i. The search is performed by accessing non-empty buckets and combining the results. Pseudo-code for the transformation is given in Algorithm 1. Note that the search starts from B_{m-1} and proceeds to B_0. This may affect the efficiency of kNN search by shrinking the covering radius of the current kNN candidate set as much as possible early on.

Let us consider an example where we insert a new object into the existing data structure. The example is illustrated in Figure 2.

Let the buckets $B_0, B_1, \ldots, B_{k+2}$ be non-empty. Thus, the first empty bucket is B_{k+3}. We build an index structure for bucket B_{k+3} containing the new object and all the objects stored in buckets $B_0, B_1, \ldots, B_{k+2}$. After building this structure, buckets $B_0, B_1, \ldots, B_{k+2}$ are nulled. Buckets B_{k+4} and upward are unchanged.

Now consider the asymptotic running time and space requirements of this approach. Let T be a

Algorithm 1 Static to dynamic transformation

```
1: function Init():
2:     B_0 ← null; m = 0

3: function Insert(x):
4:     D ← {x}
5:     Find minimum k such that B_k = null
6:     for i ← 0 to k − 1:
7:         D ← D ∪ Unbuild(B_i)
8:         B_i ← null
9:     B_k ← Build(D)
10:    if k = m:
11:        B_{m+1} ← null; m ← m + 1

12: function Query(q):
13:    ans ← ∅
14:    for i ← m − 1 downto 0:
15:        if B_i ≠ null:
16:            Search using q in B_i and update ans with results
17:    return ans
```

static metric index structure with size $S_T(n)$ that can be built in time $C_T(n)$ and perform a query in time $Q_T(n)$. BS gives us a dynamic metric index structure T' based on T that requires the storage $S_{T'}(n) \in \mathcal{O}(S_T(n))$ and bulk construction time $C_{T'}(n) \in \mathcal{O}(C_T(n))$ (assuming that both storage and construction requirements are at least linear), and, because each object is inserted in $\log n$ buckets, an amortized insertion time for n elements of $I_{T'}(n) \in \mathcal{O}(\log n \cdot C_T(n))$. In fact, if $C_T(n) \in \Omega(n^{1+\epsilon})$, for some $\epsilon > 0$, we have $I_{T'}(n) \in \mathcal{O}(C_T(n))$, that is, there is no asymptotic overhead.[3] We can, in general, guarantee a query time of $\mathcal{O}(\log n \cdot Q_T(n))$. Moreover, if $Q_T(n) \in \Omega(n^\alpha)$ for some $\alpha > 0$, which is generally assumed [10], the query time is $Q_{T'}(n) \in \mathcal{O}(Q_T(n))$.

The original version of BS method was not de-

[2]For a dynamic index, the number of buckets is, of course, unknown at the outset. The problem size n is the number of objects added *so far*.

[3]This can also be made to hold in the worst case, using lazy rebuilding techniques that we have not studied in this paper.

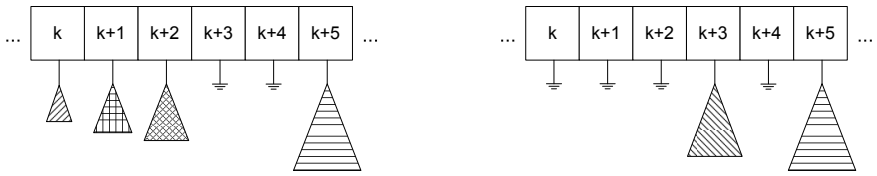


Figure 2: Illustrations of an index structure before the insertion of a new object (left) and after the insertion (right).

signed to handle deletions efficiently. Consider, for example, the scenario where we have a single non-empty bucket B_k, containing 2^k objects. To delete an object now, we have to split B_k into $B_0, B_1, \ldots, B_{k-1}$. This entails building k index structures, which might be prohibitively expensive. To address this, Overmars et al. [12] weakened the condition of the BS method so that every bucket B_k can be either empty or a static data structure which stores at least 2^{k-2} and at most 2^k objects. With this new condition, our deletion would affect only B_{k-2}, B_{k-1} and B_k. The approach of Overmars et al. is shown in Algorithm 2.

Algorithm 2 Overmars and Leeuwen

```
1: function Insert(x):
2:     Replace line 9 of Insert function of Algorithm 1
       with the following
        if |D| > 2^{k-1}: B_k ← Build(D)
        else: B_{k-1} ← Build(D)                    ▷ |D| > 2^{k-2}

3: function Remove(o):
4:     Perform a range search in B_k to find k such that o ∈ B_k
                                          ▷ k from m − 1 downto 0
5:     if not found o:
6:         return false
7:     Delete o from B_k                  ▷ |B_k| is decremented by 1
8:     if |B_k| > 2^{k-2}:
9:         return true
10:    elif |B_k| = 2^{k-2} and k ≥ 2:
11:        if B_{k-1} ≠ null:
12:            D ← Unbuild(B_k) ∪ Unbuild(B_{k-1})
13:            if |B_{k-1}| > 2^{k-2}:
14:                B_{k-1} ← null
15:                B_k ← Build(D)
16:            else:
17:                B_k ← null
18:                B_{k-1} ← Build(D)
19:        elif B_{k-1} = null and B_{k-2} ≠ null:
20:            D ← Unbuild(B_k) ∪ Unbuild(B_{k-2})
                                    ▷ |B_k| + |B_{k-2}| > 2^{k-2}
21:            B_k ← null; B_{k-2} ← null
22:            B_{k-1} ← Build(D)
23:        elif B_{k-1} = null and B_{k-2} = null:
24:            D ← Unbuild(B_k); B_k ← null
25:            B_{k-2} ← Build(D)
26:        return true
```

In line 7, we mark o as deleted in B_k. The bucket B_k might not be rebuilt until its total number of objects becomes 2^{k-2}. That would, of course, affect the search performance. In order to decrease this effect, we introduce a parameter tuning option between lines 8 and 9. There are many possibilities for the parameter tuning. For instance, the bucket B_k can be rebuilt each time when 2^{k-3} objects have been deleted from that bucket. This is the strategy that is tested in our experiments.

4. Experiments

In this section we present our experimental evaluation of two new dynamic trees based on BS, comparing them against two existing dynamic trees. As the performance measure we used the number of distance computations required to construct index structures and to answer similarity queries. We have also investigated the overhead of the BS method, by comparing the build and search times of the static indexes to those of their transformed, dynamic counterparts. We have provided performance comparisons of range and kNN queries, as well as deletion costs per object and search performance after deletions.

4.1. The testbed

We performed experiments using both synthetic data sets, generated by us, and real-world datasets obtained from the SISAP metric space library [6]. For all vectors we use the Euclidean distance.

- Uniform 10: Synthetic. 100 000 uniformly generated 10-dimensional vectors.[4]
- Clusters 10: Synthetic. 100 000 clustered 10-dimensional vectors with 10 cluster centers. The centers were randomly chosen from a uniform distribution and objects in the clusters

[4]We also have 8 192 000 uniformly generated 10-dimensional vectors for the complexity analysis of the BS index.

were generated from the multivariate normal distribution around each of the cluster centers with a variance of 0.1.

- Uniform 20: Synthetic. 100 000 uniformly generated 20-dimensional vectors.
- Clusters 20: Synthetic. 100 000 clustered 20-dimensional vectors with 100 cluster centers. We followed the same procedure as in Clusters 10 to generate the cluster centers.
- NASA: 40 150 feature vectors with 20 dimensions extracted from NASA images.
- Dictionary: a dictionary of 69 069 English words. We use the edit distance (or Levenstein distance), that is, the minimum number of insertions, deletions, and substitutions needed to transform one string into another.
- Histogram: a collection of 112 682 color histograms (112-dimensional vectors) from an image database.

Table 1 shows the intrinsic dimensionalities (*idim*s) [4] of the datasets. The distance histograms of the data sets are shown in Figure 3.

4.2. Experiment settings

We have applied the BS method to VP- and SSS-trees and call the resulting dynamic structures the BS-VP-tree and BS-SSS-tree, respectively. We have compared their performances to two dynamic metric index structures, the DSA-tree and EGNAT. We set the maximum node fanout of the BS-SSS-tree to 5, 10, 20, 40 and 80. The parameter α was 0.45 for the 20-dimensional and 0.40 for the remaining of the datasets. The value of M is estimated before every (re)construction of a bucket as follows. An arbitrary object in the bucket is selected as the boundary object. Then, the distances between the boundary and all objects in the bucket are computed 10 times by maximizing the value of M and renewing the boundary object from current one. The cost of this estimation is also included in the construction and deletion costs. We used the SISAP implementation [6] of DSA-tree with time-stamping and bounded arity. The original authors [11, § 5.8] suggested this version of DSA-tree that would give the best results in terms of construction cost and search efficiency. The maximum arities of DSA-tree were set to 2, 4, 8, 16 and 32, as in their experiments. For EGNAT, we set the parameters by trial and error. We used internal node sizes of 4, 8, 12, 16 and 20 and maximum leaf node arities of 5, 10, 20, 40 and 80. In total, we performed 18 (5+4+3*3) runs (with several queries).[5]

We randomly shuffled the order of all objects in each dataset 10 times, obtaining 10 versions of the dataset and the results were averaged over 10 runs using these versions. For each run, a query set consists of 1000 queries which were selected from the respective dataset and the remaining objects in the dataset used for indexing. We selected search radii for range queries so that we capture on average 0.01 %, 0.1 % and 1 % of the vectors. The search radii were in the range from 1 to 4 for the dictionary, capturing on average 0.003 %, 0.042 %, 0.361 %, and 1.946 % of the dataset, respectively. For kNN search, we compared the search efficiency of four structures by varying the result size thresholds, using the values 1, 5, 10, 20, 40 and 80. We report only best results in terms of search efficiency from the results obtained with different parameters use on every query set.

For deletions, the cost includes both locating the object to be deleted (usually with a range search with radius 0) and rebuilding buckets. First, we constructed the BS-VP-tree and BS-SSS-tree on the datasets. Then, we deleted every 10 % of the corresponding datasets from BS indexes and obtained the ratio between the number of distance computations required to answer a query set in the *deleted* BS indexes and in *newly built* BS indexes for the remaining objects in the datasets after deletions.[6]

We have implemented our experimental framework in C++, which was compiled in gcc 4.6.2 with the option -O3. All experiments are performed on a PC with a 3.3 GHz Intel Core i5-2500 processor and 8 GiB RAM. We did not use any caching of distances during the construction of index and query processing.

4.3. The overhead of the BS index

First, let us consider the *construction cost* overhead of BS-based index structures. We constructed the VP-tree, SSS-tree (with maximum

[5] Note that leaf node size should be greater than or equal to internal node size.

[6] The experiments of deletions with DSA-tree were not performed because of a bug in the SISAP metric space library. We contacted one of the original authors of DSA-tree and it became clear that the bug can not be fixed before the journal's deadline.

Uniform 10	Clusters 10	Uniform 20	Clusters 20	NASA	Dictionary	Histogram
13.36	9.24	27.64	20.44	5.18	8.49	2.74

Table 1: *idim*s of the datasets.

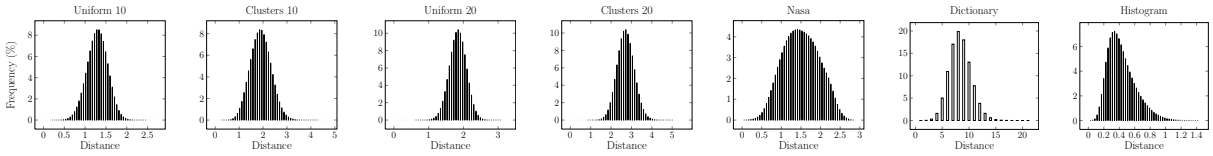


Figure 3: Distance distribution histograms.

node fanout 5), BS-VP-tree and BS-SSS-tree (with maximum node fanout 5) 10 times on every 10 % of several datasets and obtained the ratio between the number of distance computations that required to build the BS index and static index with the same settings, with the BS index built incrementally. The mean of ratio was 3.23 (standard deviation 0.61), with minimum and maximum values of 1.69 and 4.27. So in our experiments, the BS index is at most 4.27 times as costly to build as the corresponding static index. Figure 4 shows the construction cost overhead of the BS-based index structures using the box plots that display the minimum, the 25 % quantile, the median, the 75 % quantile and the maximum value.

The figures show that the average ratio for VP-tree is much higher than the SSS-tree, and the values for VP-tree are distributed almost evenly. The values for SSS-tree are positively skewed in general, i.e., it has relatively few high values; it also performed particularly well on the dictionary.

Now let us consider the ratio for *index construction time* and *query set execution time* with all k and search radii. We followed the same principle previously used for the construction cost ratio to obtain these ratios with $2^m - 1$ objects for each dataset. The motivation for this was to force all the buckets in the BS structure to be non-empty; that is, we intentionally increased the overhead of BS-based index structures, intending to elicit the worst-case search performance. The ratios are shown in Figure 5. For the construction time ratio, the maximum value for BS-VP-tree was 3.62 (with construction time 1.23 s) on the dictionary (Figure 5a) while the maximum value for BS-SSS-tree was 3.80 (with

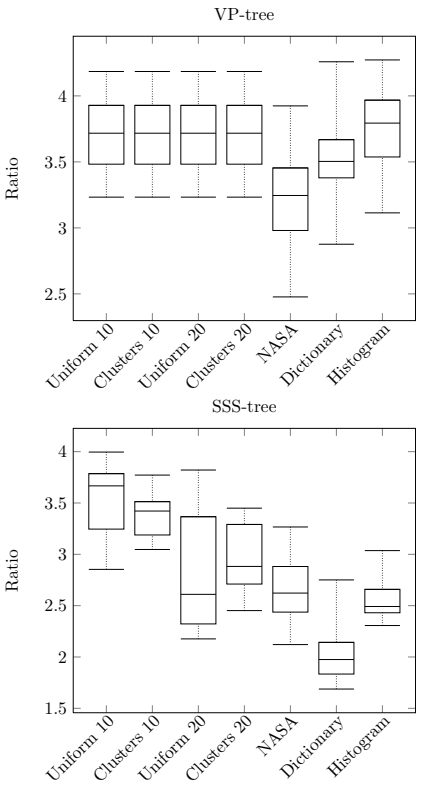


Figure 4: Construction cost ratio of BS index to static index with the same settings.

construction time 11.09 s) on Uniform 20 (Figure 6). In Figure 5b, 5c, 5e and 5f, we see that there is almost no search time difference between static and BS-based index structures on the 20-dimensional synthetic vectors due to high *idim*s. Across all of our experiments, the maximum value of query set execution time for the static index structures was 19.87 s while for the BS-based index structures the maximum value was 20.97 s.

4.4. Comparison of construction costs

All index structures were built in an incremental fashion, i.e., initially all of the index structures were empty and then all objects in the datasets were added into the index one by one. The construction costs are shown in Table 2.

As the *idim* of the synthetic datasets increases we see that the datasets become difficult to index. This increase clearly affects the construction cost of the SSS-tree. This effect may be due to the fact that every object tends to become a cluster center of the tree because all objects are approximately equidistant from each other in high-dimensional spaces. It should also be noted that the clustering cost of the SSS-tree is high also in the static case, so this is not an artifact of our approach.

When considering any overhead in construction, it is important to note that our method is quite amenable to *bulk loading*: If a given data set is available at the outset, or if a large number of objects are added, there is *no need* to build the structure incrementally, by adding individual objects. Instead, which buckets need to be filled can be easily calculated from the total data size, and the objects can be partitioned among these (e.g., randomly), and the static structures built. This means that there would be no need for multiple rebuilds, and the overhead would be much lower. For example, if the data size were a power of 2, there would be *no overhead whatsoever*. The resulting data structure would still retain all its dynamic properties. (The overhead in general will, of course, vary with how close the data size is to a power of 2, either above or below.)

4.5. Empirical complexity analysis of BS index

We have now looked at the overhead of the BS algorithm beyond the corresponding static structures, and we have compared the construction costs of the BS-based structures and custom-designed dynamic ones. We now wish to tentatively examine the asymptotic complexity of the method, both for construction and search. We have some expectations about how the method ought to behave, but these are based on certain assumptions (primarily the running times of the static, underlying structures), which may not hold in practice.

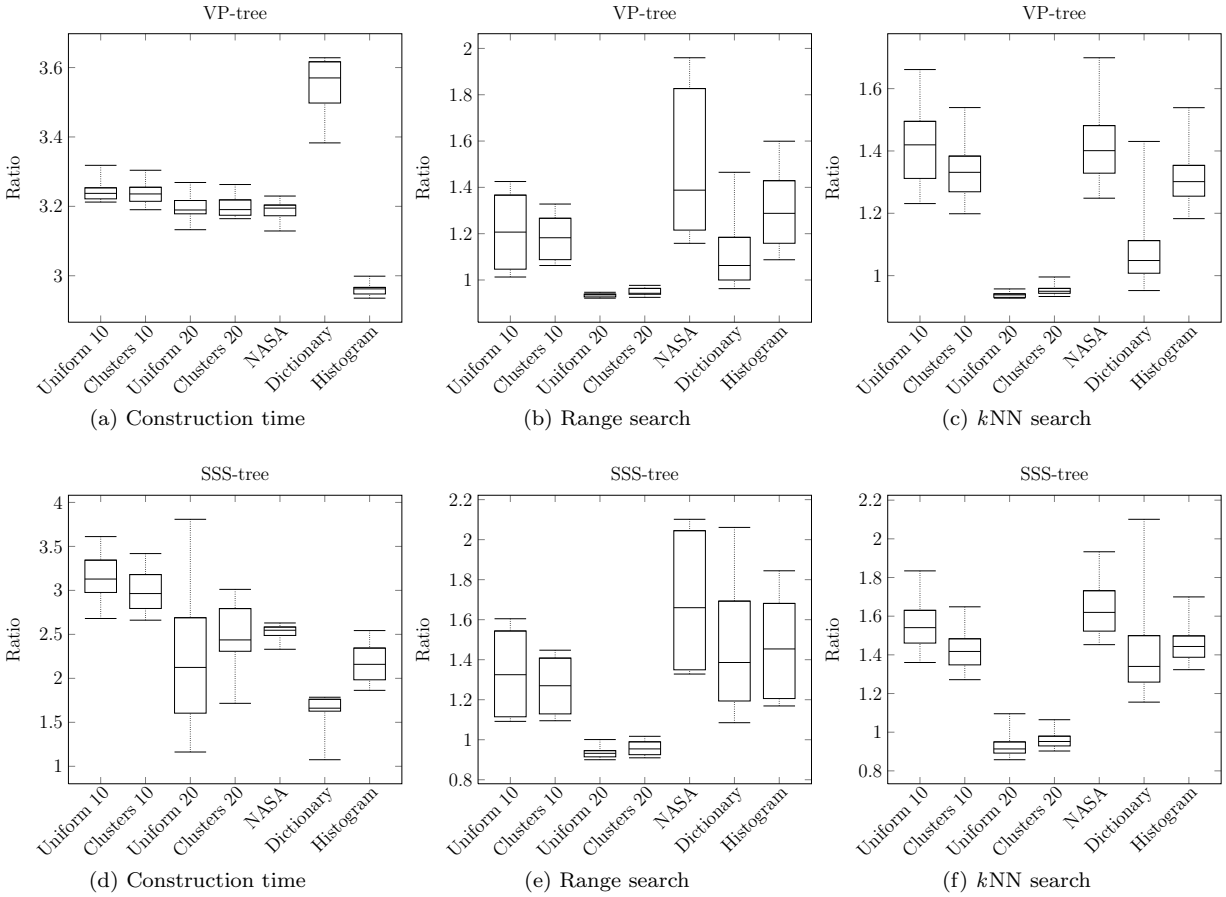
To map out the functional relationship between input size and performance we use a doubling experiment [13]. To get a sufficiently large data set, we generated 8 192 000 uniform 10-dimensional vectors. We then measured performance with problem sizes at various powers of 2, starting at 8000, with each experiment performed 10 times on 10 randomly shuffled versions of the data. In each experiment, we built the VP-tree, SSS-tree, BS-VP-tree and BS-SSS-tree[7] and obtained the ratio between the time required to build the BS indexes and the static indexes.

As discussed in Section 3, if the build time was $\Omega(n^{1+\epsilon})$, for some $\epsilon > 0$, we would expect the ratio between the build-times for the static and BS-based dynamic structures to be a constant (no asymptotic overhead). For the VP-tree, however, the construction cost is $\Theta(n \log n)$ [see, e.g., 5], which means we would expect a log-factor between the static and incremental dynamic construction cost. While an asymptotic analysis is harder to do for the SSS-tree, because the arity varies from node to node, it is not unreasonable to expect a similar complexity. In Figure 6, we have plotted $C_T(n)/n$ for the SSS-tree, with a logarithmic horizontal axis. A straight line would indicate a growth proportional to $n \log n$. The regression line has been included, and, as can be seen, $C_T(n)$ seems to grow more slowly than this (i.e., a sublinear trend in the log-plot), which would mean that we could expect the ratio of the static and BS-based build costs to be logarithmic here as well.

To see whether the ratios indeed *are* logarithmic, we have plotted them in a similar manner in Figure 7 (once again with a regression line), where a straight line would indicate a perfect logarithmic relationship between data size and the index build slowdown due to the BS method. It seems like the trend is, indeed, approximately linear.

We also performed kNN searches on the four structures by using all of the result size thresholds and obtained (1) the time required to answer a query set in the BS-based and static indexes and

[7] The BS indexes were built in an incremental fashion.



(a) Construction time (b) Range search (c) kNN search

(d) Construction time (e) Range search (f) kNN search

Figure 5: Construction time and query set execution time ratio of BS index to static index with same settings.

Dataset	BS-VP-tree	BS-SSS-tree	EGNAT	DSA-tree
Uniform 10	5762240	59199773	3451333	3126671
Clusters 10	5762240	38317600	5327960	3298234
Uniform 20	5762240	208912258	7582301	8631551
Clusters 20	5762240	96468450	8876612	8857394
NASA	1952946	13619821	1646678	1219949
Dictionary	4676487	90720799	4820213	5531384
Histogram	6246315	45558887	9688228	4124772

Table 2: Construction costs of index structures on various datasets.

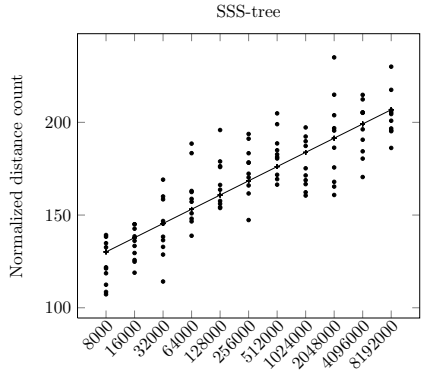


Figure 6: Normalized construction cost ($C_T(n)/n$) vs data set sizes on the synthetic set with SSS-trees.

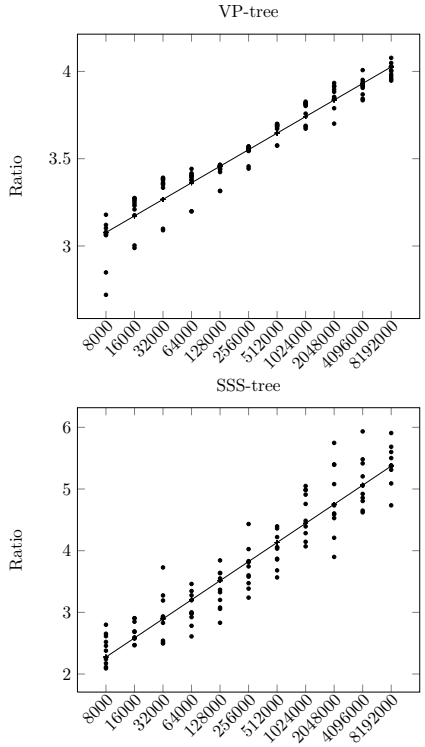


Figure 7: Construction time ratio of BS index to static index with same settings on various data set sizes.

(2) the ratio between the query set execution time in the BS indexes and in the static indexes. The results are given in Figure 8. (Note that both axes are logarithmic.) In this case, the expected ratios would depend on whether the query time is, indeed, $\Omega(n^\alpha)$, for some $\alpha > 0$. If this were the case, we would expect straight lines in subfigures (a) and (c), and we would have a constant performance ratio, which would show as a horizontal line in subfigures (b) and (d). As can be clearly seen, this is not exactly the case, although it may not be too far from the truth. It would seem that the empirical query performance is somewhere between polylogarithmic and polynomial, leading to a ratio that grows, albeit slowly, with problem size.

4.6. Query performance, with and without deletions

Figure 9 shows the search results over the synthetic datasets and the impact of dimensionality. The performance of all of the index structures degrades when the dimensionality increases, especially for EGNAT and BS-VP-tree. In Figure 10, the search results over the real-world datasets are shown. The BS-VP-tree outperforms EGNAT for the real-world datasets and is comparable to DSA-tree for range queries with low selectivity. For the dictionary, the BS-SSS-tree outperforms the DSA-tree, achieving up to twice the search efficiency. In all of the experiments, the BS-SSS-tree outperforms all the other index structures in our experiments, a result almost certainly due to the efficiency of the SSS-tree itself, which comes at the price of a higher building cost. The contribution of our method in this case is that such a tradeoff between build cost and search efficiency can be made in the first place, by providing a dynamic version of the SSS-tree.

Deletions were performed on several datasets. We measured the all distance computation ratios (explained in Section 4.2) for all k and search radii over several data sets. The results are shown in Figure 11. Each point in the figures shows the average of those ratios, while the error bars show the standard error of the mean of the values. The highest ratio of search costs is 1.52, and occurs after deleting 70 % of the Histogram data set. The deletion cost of the BS-VP-tree is quite low in all of our experiments. The highest deletion cost for the BS-SSS-tree was 3129, which resulted from deleting 40 % of the dictionary.

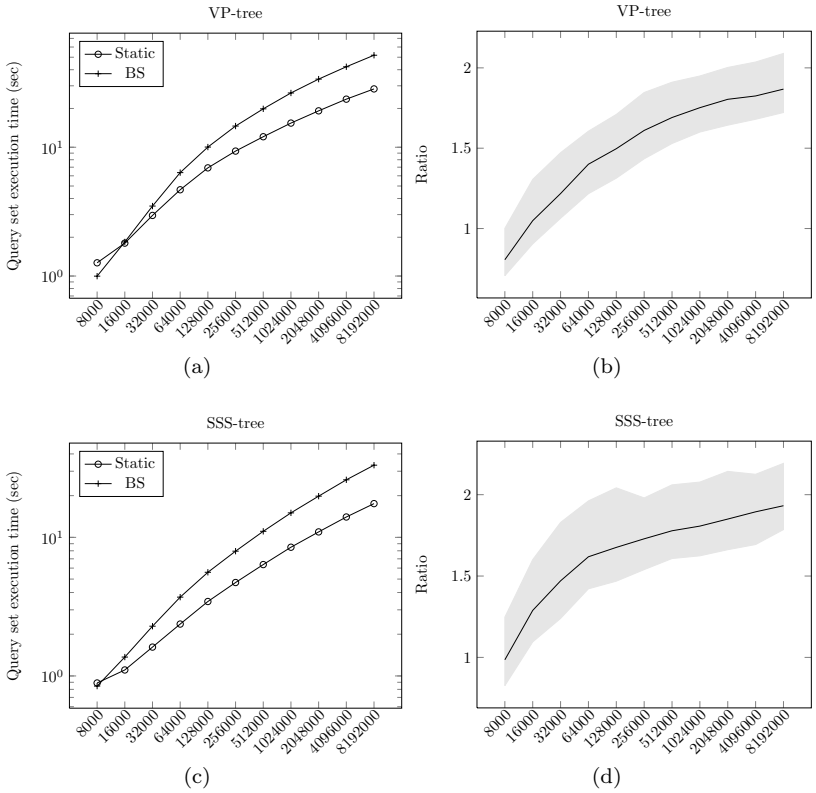


Figure 8: Mean of the query set execution time (a, c) and query set execution time ratio of BS index to static index with same settings (b, d) on various data set sizes. The gray area is filled between the minimum and maximum values, whereas the line represents the mean of the values.

5. Conclusions

We have studied the Bentley-Saxe algorithm for static-to-dynamic data structure transformations and how it can be applied to in similarity search, yielding a simple method for transforming static index structures into dynamic ones. We have also empirically demonstrated that the method has a reasonably low overhead, both in terms of building and search cost. In fact, this overhead is low enough that when adapting a particularly efficient static data structure such as the SSS-tree, we can still achieve search times lower than comparable custom-designed dynamic data structures. In addition to this increased performance, the dynamic structures resulting from using the Bentley-Saxe method can be considerably less complex than other dynamic indexes, given that it is simply an isolated add-on to existing (usually simpler) static indexes.

References

[1] J. L. Bentley and J. B. Saxe. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301 – 358, 1980.

[2] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of 21th International Conference on Very Large Data Bases, VLDB*, pages 574–584, 1995.

[3] N. Brisaboa, O. Pedreira, D. Seco, R. Solar, and R. Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In *Proceedings of SOFSEM'08*, number 4910 in LNCS, pages 186–197, 2008.

[4] E. Chávez and G. Navarro. A probabilistic spell for the curse of dimensionality. In *Revised Papers from ALENEX'01*, pages 147–160, 2001.

[5] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[6] K. Figueroa, G. Navarro, and E. Chavez. Metric spaces library, 2010. Downloaded November 15th, 2011 from `http://www.sisap.org/Metric_Space_Library.html`.

[7] A. W.-C. Fu, P. M.-S. Chan, Y.-L. Cheung, and Y. S. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *The VLDB Journal*, 9(2):154–173, 2000.

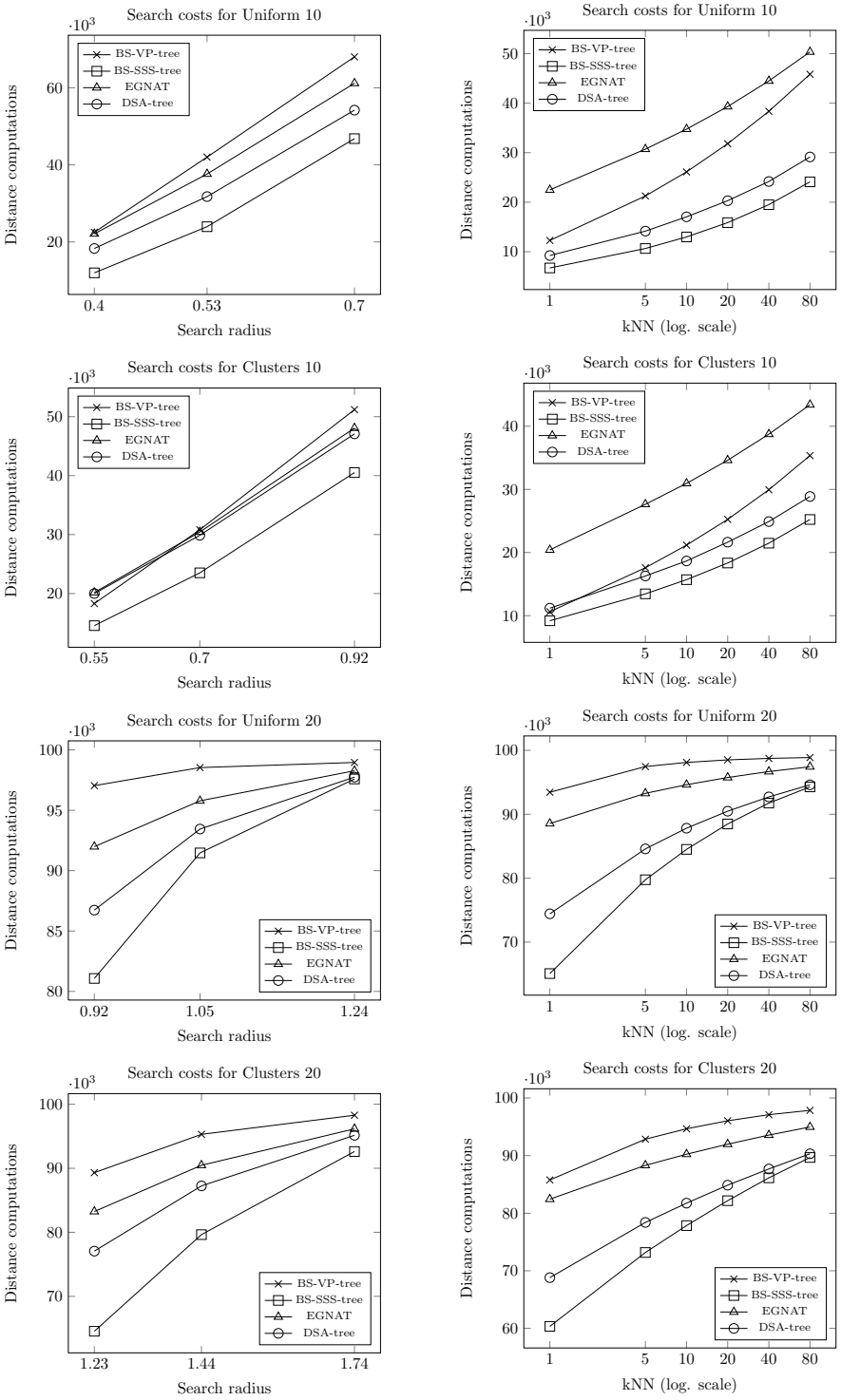


Figure 9: Performance evaluations on the synthetic datasets.

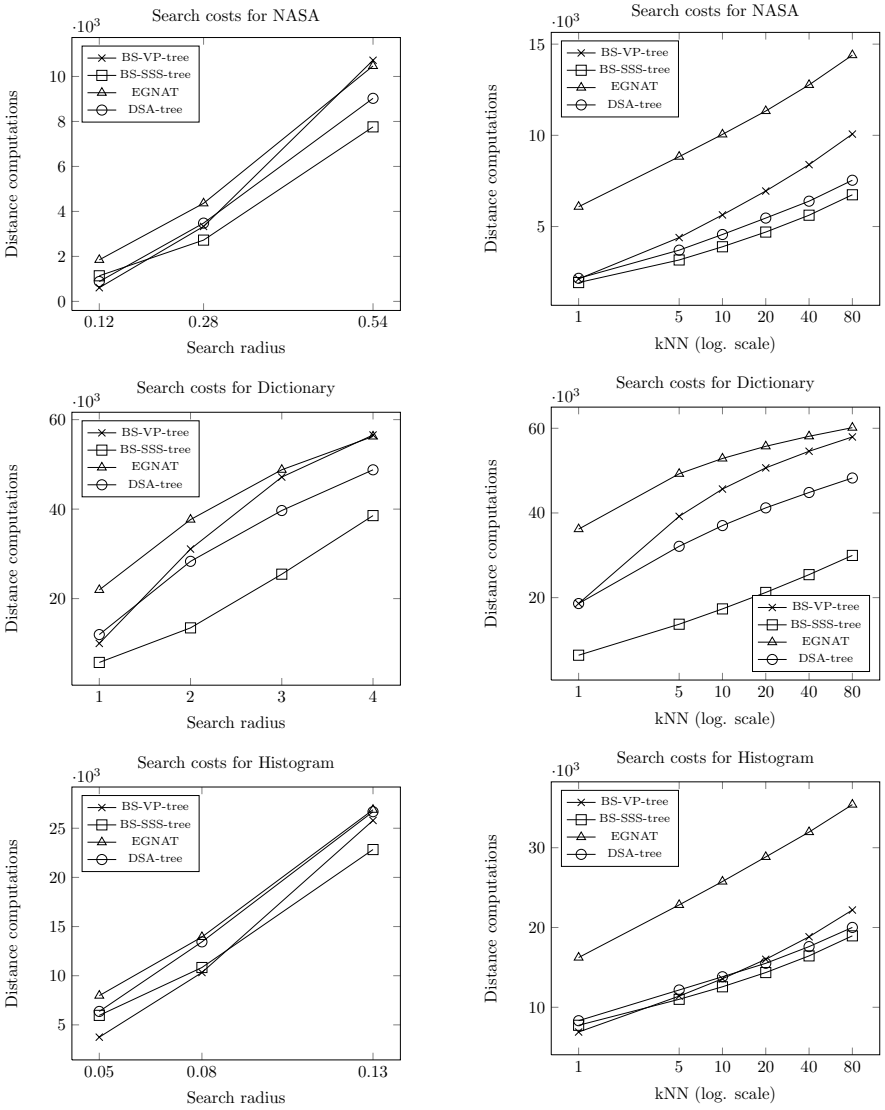


Figure 10: Performance evaluations on the real-world datasets.

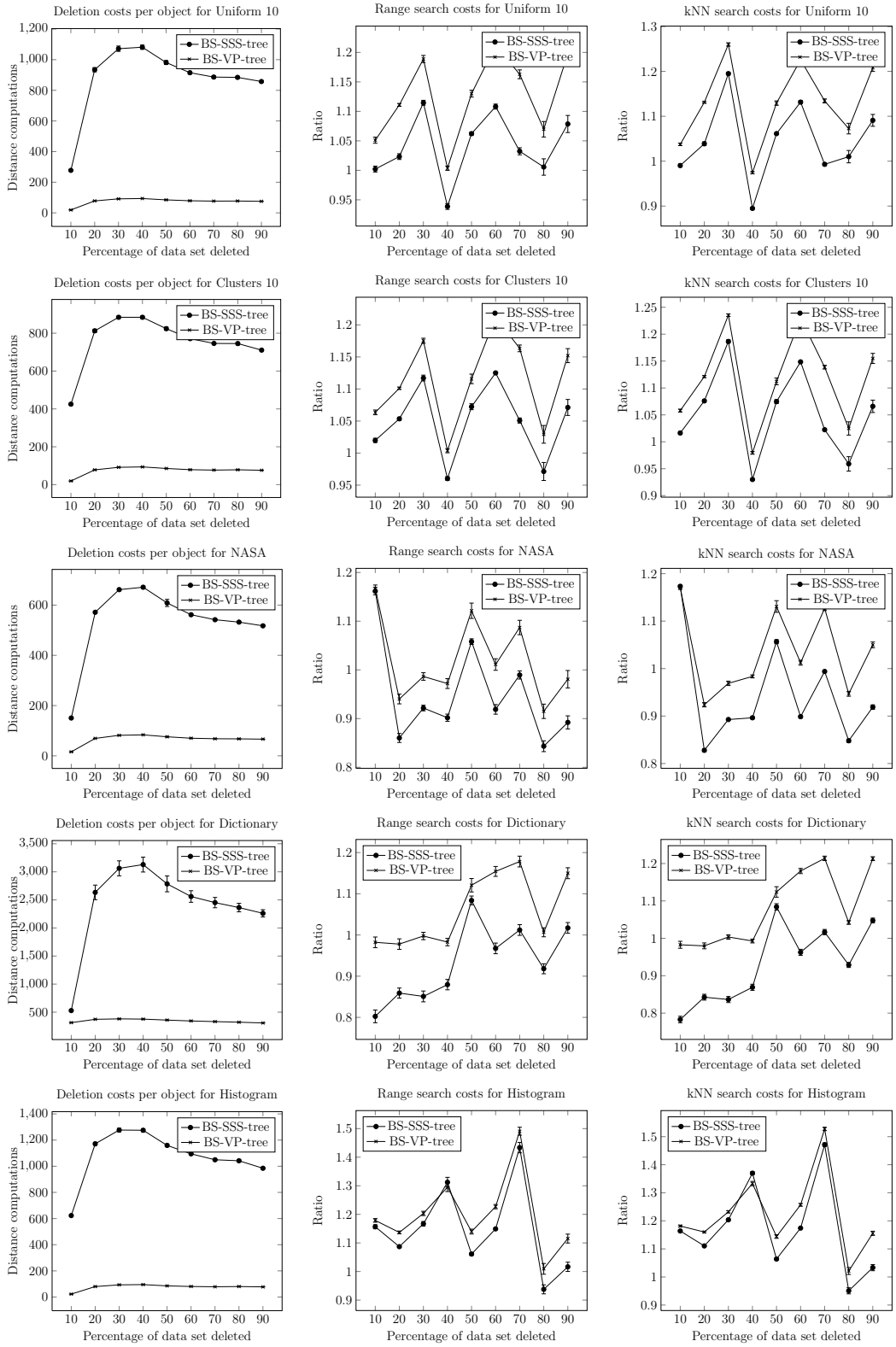


Figure 11: Deletion costs and distance computation ratios of *deleted* BS index to *newly built* BS index.

[8] M. L. Hetland. The basic principles of metric indexing. In *Swarm Intelligence for Multi-objective Problems in Data Mining*, volume 242 of *Studies in Computational Intelligence*. 2009.

[9] G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28–46, August 2002.

[10] G. Navarro. Analyzing metric space indexes: What for? In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, SISAP '09, 2009.

[11] G. Navarro and N. Reyes. Dynamic spatial approximation trees. *Journal of Experimental Algorithmics (JEA)*, 12:1.5:1–1.5:68, 2008.

[12] M. Overmars and J. Leeuwen. Two general methods for dynamizing decomposable searching problems. *Computing*, 26:155–166, 1981.

[13] R. Sedgewick. *Algorithms in Java, Third Edition*. Addison-Wesley, 2003.

[14] R. Uribe, G. Navarro, R. J. Barrientos, and M. Marín. An index data structure for searching in metric space databases. In *Proceedings of the 6th International Conference of Computational Science, ICCS*, volume 3991, pages 611–617, 2006.

[15] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual Symposium on Discrete algorithms*, pages 311–321, 1993.

[16] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search : The Metric Space Approach*. Springer, 2006.