Topology B-Trees and Their Applications

Paul Callahan[⋆] Michael T. Goodrich[⋆⋆] Kumar Ramaiyer[⋆⋆⋆]

Dept. of Computer Science, The Johns Hopkins Univ., Baltimore, MD 21218, USA

Abstract. The well-known B-tree data structure provides a mechanism for dynamically maintaining balanced binary trees in external memory. We present an external-memory dynamic data structure for maintaining arbitrary binary trees. Our data structure, which we call the *topology B-tree*, is an external-memory analogue to the internal-memory topology tree data structure of Frederickson. It allows for dynamic expression evaluation and updates as well as various tree searching and evaluation queries. We show how to apply this data structure to a number of external-memory dynamic problems, including approximate nearest-neighbor searching and closest-pair maintenance.

1 Introduction

The B-tree [8, 12, 14, 15] data structure is a very efficient and powerful way for maintaining balanced binary trees in external memory [1, 11, 13, 18, 19, 21, 22, 2]. Indeed, in his well-known survey paper [8], Comer calls B-trees "ubiquitous," for they are found in a host of different applications. Nevertheless, there are many applications that operate on unbalanced binary trees.

In this paper we describe a data structure, which we call the *topology B-tree*, for maintaining unbalanced binary trees in external memory. We allow for dynamic expression updates [7] and we consider a number of tree-search queries on arbitrary binary trees, which in turn can be used to solve a number of dynamic external-memory problems, including approximate nearest-neighbor searching and closest-pair maintenance. The topology B-tree is an external memory analogue to the *topology tree* data structure of Frederickson [10], which is an elegant internal-memory method for maintaining unbalanced binary trees.

Before we describe our results, let us review the model for external memory [1, 11, 13, 18, 19, 21, 22] that we will be assuming throughout this paper.

1.1 The External-Memory Model

We assume that the external-memory device (e.g., a disk) is structured so that seek time is much larger than the time needed to transfer a single record; hence,

[⋆] This research supported by the NSF under Grant CCR-9107293.

[⋆⋆] This research supported by the NSF under Grants CCR-9300079, IRI-9116843 and CCR-9300079.

[⋆⋆⋆] This research supported by the NSF under Grants CCR-9300079, IRI-9116843 and CCR-9300079.

to compensate for this time difference data is transferred between internal and external memory in *blocks* of records. We let B denote the number of records that can be transferred in a single external-memory input or output (i/o), and our measure of efficiency will be in terms of the total number of i/o's needed for a particular computation. Indeed, the model does not at all consider the number of internal computations performed by the CPU (provided this is kept within reason). This is motivated by the large difference in speed between modern CPU and disk technologies, for most computations on modern CPU's are measured in nanoseconds whereas most access times for modern disk drives are measured in milliseconds. As Comer[1] puts it, this is analogous to the difference in speed in sharpening a pencil by using a sharpener on one's desk or by taking an airplane to the other side of the world and using a sharpener on someone else's desk.

In addition to the parameter B, measuring block size, we also use M to denote the number of records that can fit in internal memory, and we use N to denote the total number of records i.e., input size. For the problems we consider, we make the reasonable assumptions that $M < N$, and $1 \leq B \leq M/2$.

1.2 Our Results

As mentioned above, in this paper we give an external-memory analogue to the topology tree data structure, which we call the topology B-tree. We show how use this data structure to dynamically maintain arbitrary binary trees, subject to the operations insertion and deletion of nodes, a generalized deepest-intersection search, and evaluation of arithmetic expressions, which we implement using $O(\log_B N)$ block i/o's. In addition, we show that each of the operations on dynamic expression trees require $\bar{O}(\log_B N)$ block i/o's[2]. Finally, using these primitives, we design optimal external-memory methods for dynamically solving the following geometric problems:

- Approximate nearest neighbor [3, 4]: given a set S of points in $\mathbb{R}^d$, for fixed d, a query point p, a metric L_t, and a parameter ϵ, find a point q in S that is within distance at most $(1+\epsilon)$ times the distance of the actual nearest neighbor of p in S under L_t metric. We support this query under the operations of insertion and deletion of points in S.
- Closest pair [6, 5]: given a set S of points in $\mathbb{R}^d$, for fixed d, find a pair of points in S which are the closest among all pairs of points in S under the Euclidean distance metric. We support this operation under insertion and deletion of points from S.

Our query algorithms all use an optimal $O(\log_B N)$ external-memory i/o's.

2 The Topology Tree Data Structure

Before we describe our data structure in detail, however, let us first review the structure of the topology tree [10], and discuss how to implement insertion and

[1] Personal communication.
[2] We use the notation $\bar{O}(.)$ to describe amortized complexity.

deletion of nodes, a generalized deepest-intersection search, and the evaluation of arithmetic expressions.

Given any rooted tree $T = (V, E)$, the topology tree $\mathcal{T}$ is a balanced tree constructed on top of the nodes of T by repeated clustering. The topology tree $\mathcal{T}$ has multiple levels, and at each level there is a tree structure defined on the nodes at that level. Moreover, the nodes at any level define a partition of V. The rules for clustering the nodes are simple, and they enforce certain constraints on the resulting structure, which makes the topology tree balanced. The leaves of the topology tree $\mathcal{T}$ are the nodes V of tree T, and are at level 0. We refer to the tree $T_0 = T$ as the *base* tree of the topology tree. These nodes of T_0 are clustered to form bigger nodes, and result in a new tree structure T_1. The nodes of T_1 are the nodes at level 1 of the topology tree, and from each node v of T_1 there are edges (in $\mathcal{T}$) to the leaves of the topology tree which were combined to form v. Now clustering is done on nodes of T_1 to obtain the nodes for level 2 of the topology tree, and so on. We refer to the tree T_i as the *level i tree*, for its nodes are all at level i in the topology tree $\mathcal{T}$ (numbering up from the leaves). Eventually the clustering results in a single node which forms the root of the topology tree. The clustering is done according to the following simple rules:

1. Each cluster of degree 3 is of cardinality 1.
2. Each cluster of degree less than 3 is of cardinality at most 2.
3. No two adjacent clusters can be combined and still satisfy the above.

The first two rules guide the clustering operation, and the last one specifies the maximality property of clustering at each level. Based upon these clustering rules, it is fairly straightforward to show that the number of levels in a topology tree is $O(\log N)$, where N is the number of nodes in the base tree T. Frederickson [9] proves the following (stronger) lemma which relates the number of clusters at one level with the previous level:

Lemma 1. *[9] For any level $l > 0$ in a topology tree, the number of clusters at level l is at most $5/6$ of the number of clusters at level $l - 1$.*

2.1 Implementation of Primitives on Topology Tree

In this section, we discuss how to implement the dynamic operations on the topology tree. Our methods are very similar to those of Frederickson [10], but simpler, since we consider here only a subset of the operations he considers.

We consider the following operations on an arbitrary rooted binary tree T:

insert(T, v, w, pos): Insert the node v in the tree T as the *pos (left or right)* child of node w.

delete(T, v): Delete node v from the tree T.

swap($\mathbf{T}, \mathbf{T_V}, \mathbf{w}$): Given a tree T_v rooted at node v, replace the subtree rooted at w in T with T_v.

2.2 Augmenting the Tree for Generalized Searching

We augment the topology tree to perform a generalized searching computation and evaluate arithmetic expressions, under the dynamic operations outlined above. Additional operations we implement are as follows:

intersect(T, x, v): Suppose each node w of the tree T stores an $O(1)$-sized description of a set Q_w, such that the set stored at a node w always contains the sets stored at its children. This operation tests if an object x intersects a set Q_v for a given node v in T.

deepest-intersect(T, x): Suppose again that each node w of the tree T stores an $O(1)$-sized description of a set Q_w, such that the set stored at a node w always contains the sets stored at its children. This operation identifies each node v in the tree T in which x intersects Q_v, but x does not intersect the set associated with any of v's children (or v is a leaf).

eval-expression(T, v): Suppose the leaves of the tree T store values from a semiring $(S, +, *, 0, 1)$, and the internal nodes store the operations $+$ or $*$. This operation evaluates the arithmetic expression represented by the subtree T_v rooted at v in T.

Our implementation of the insert, delete, and swap operations is similar to Frederickson [10]. We implement these operations using constant number of *reclustering* operations i.e., removal of clusters along a root-to-leaf path in $\mathcal{T}$, and performing the clustering again. The complexity of reclustering operation is $O(\log N)$ as shown in the following lemma:

Lemma 2. *The reclustering operation along a path in a topology tree $\mathcal{T}$ uses a total of $O(\log N)$ pointers from $\mathcal{T}$ and from all the level trees. Moreover at each level, the number of pointers modified in the level tree is constant (at most 2).*

Hence the maintenance of the topology tree after any of the dynamic operations takes $O(\log N)$ time. We also show in the full version how the additional operations can be implemented on a topology tree in $O(\log N)$ time ($O(k{*}\log N)$ for deepest-intersect queries, where k is the size of the output).

3 B-Lists

In this section, we divert our attention and consider a method that is probably part of the folklore in external-memory algorithms, but which is applicable in the construction of our final data structure.

Suppose we are given a doubly-linked list of N weighted nodes, and a parameter B, where each node u in the list is of weight $w_u \leq B$. Let $W = \sum_{i=1}^{N} w_i$. The problem is to appropriately group the nodes in the list into *blocks* so that the weight of each resulting block is less than or equal B, and the total number of blocks is $O(W/B)$. Also, we require that the structure supports the operations of insertion and deletion of nodes in the list.

We solve this problem by simply grouping contiguous nodes into blocks, and maintain the following *weight invariant*: the sum of the weights of any two adjacent blocks is at least B, and the weight of each block is less than or equal to B. We refer to the resulting structure as a *B-list.*

We can easily show that insertion and deletion operations on B-lists manipulates only $O(1)$ blocks. Also, it is easy to show that the operations of weight updates of nodes can also be done similarly by changing only $O(1)$ blocks.

4 Hierarchical B-lists

In this section, we show how to build a structure using the B-list structures, which we call the *hierarchical B-list.* This structure is motivated by the skip list structure of Pugh [20].

The hierarchical B-list consists of a hierarchy of B-lists in which only the blocks of adjacent B-lists are connected. We assign a level to each B-list, and the level numbers increase from bottom to the top of the hierarchy. The pointers are directed, and we refer to them as *down* or *up* pointers based on the direction. We require the blocks in the underlying B-lists satisfy the following connectivity constraint: each block has at most B *down* pointers, and has at most constant number of *up* pointers (if present). We define the hierarchical B-lists to be *governed* if all the inter-level pointers are *down* pointers, and *ungoverned* otherwise.

The blocking on the individual B-lists are done independently as discussed in the previous section. When we split (merge) blocks the pointers get added (removed) to (from) the blocks in the next B-list which may require further spliting (merging). We can do update operations on hierarchical B-lists as in *hysterical B-trees* [15]. We give the details in full version.

In ungoverned hierarchical B-lists, during spliting and merging we may need to perform $O(B)$ pointer changes, since we need to add new *up* pointers to parent blocks. But using a result of hysterical B-trees [15], we can prove that the number of pointer changes during a split or merge is $\bar{O}(1)$.

We now consider hierarchical B-lists consisting of $O(\log_B N)$ B-lists, which we can easily show as requiring a storage of $O(N/B)$ i/o blocks. We can also show the following lemma for update operations on ungoverned hierarchical B-lists.

Lemma 3. *We can perform the operations of search and updates on a ungoverned hierarchical B-list requiring a storage of $O(N/B)$ i/o blocks, using $\bar{O}(\log_B N)$ i/o's.*

For governed hierarchical B-lists, however, we can show the following:

Lemma 4. *We can perform the operations of top-down search and updates on a governed hierarchical B-list requiring a storage of $O(N/B)$ i/o blocks, by performing only $O(\log_B N)$ i/o's.*

5 The Topology B-Tree

In this section we give details of the construction of our external-memory data structure, the *topology B-tree*, and we also discuss some of its properties.

Given a topology tree $\mathcal{T}$, we group the clusters of $\mathcal{T}$ to form a tree of bigger nodes, which we call *super clusters*. This process increases the degree of each node in the resulting tree. We prove bounds on the degree, the size of the new clusters, number of nodes, and the depth of the resulting tree. We also show how to implement dynamic operations on this topology tree of super clusters. We refer to this method of grouping of clusters into super clusters as *stratification*, and we call the resulting tree the *stratified topology tree*.

The lemma 1 shows that the number of clusters in a topology tree decreases in a geometric progression as we move from leaves to the root. This provides us a method for stratification, which we now discuss in detail. We split the levels of a topology tree into groups of contiguous $\log_2 B$ levels. We refer to each level of $\mathcal{T}$ that is a multiple of $\log_2 B$ as a *super level*. We refer to the contiguous group of $\log_2 B$ levels between two super levels, say i and $i+1$, in $\mathcal{T}$ as a *layer* i of $\mathcal{T}$. From every node u in each super level, we construct a super cluster by including all the descendants of u in $\mathcal{T}$ up to but not including the nodes of the next super level. We refer to the resulting tree of layers of super clusters as the *stratified topology tree*. To obtain our final structure, we construct a B-list on the super clusters in each layer of stratified $\mathcal{T}$ (ordered left to right), and then build a hierarchical B-lists structure over the B-lists constructed on all the layers of $\mathcal{T}$. We call the resultant structure the *topology B-tree*.

Now consider a block b at level i in the hierarchical B-lists. The block b contains one or more super clusters from layer i of $\mathcal{T}$. But the total number of nodes of $\mathcal{T}$ in b from these super clusters is at most B. These nodes have descendants which belong to at most B blocks at level $(i-1)$ of the hierarchical B-lists. We make the *down* pointers for the block b point to these blocks in the B-list at level $i-1$. Similarly we make the *up* pointers for a block point to the blocks containing the ancestor super clusters (if the application needs ungoverned hierarchical B-lists). We can easily bound the number of up pointers required for each block by at most 2.

We now prove some properties of the topology B-tree:

Lemma 5. *In a topology B-tree $\mathcal{T}'$ corresponding to a topology tree $\mathcal{T}$ of N nodes,*

1. *the number of blocks in $\mathcal{T}'$ is $O(N/B)$,*
2. *the depth of $\mathcal{T}'$ is $O(\log_B N)$, and*
3. *each super cluster belongs to exactly one block.*

Proof: These follow directly from the above discussions. ■

We begin our discussion of the implementation of our dynamic operations by first proving an important bound on the number of blocks that may be modified in a reclustering operation performed on a topology B-tree due to an insertion or deletion of a node in T. When we access a block, we have information about adjacencies in the topology tree as well as information from all the level tree nodes in that blocks. When we perform the changes required for reclustering within a block, we use the level tree edges which point to other blocks. This operation could potentially access a large number of blocks. But, as we show in the following lemma, the number of blocks that may be accessed during the reclustering operation is not too large.

Lemma 6. *The total number of blocks accessed during a reclustering operation on a topology B-tree is $O(\log_B N)$.*

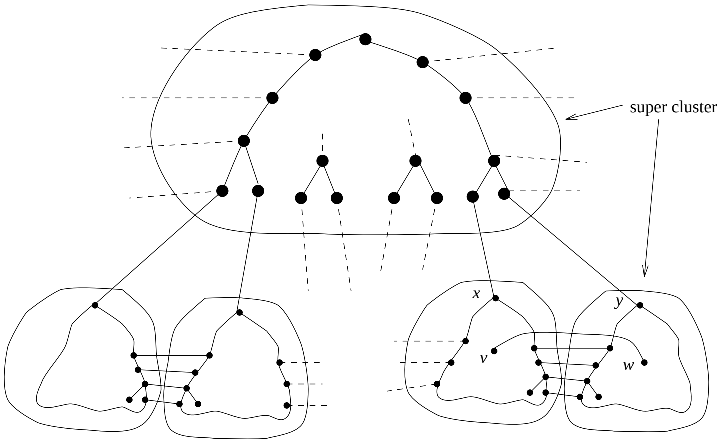

Fig. 1. The Structure of a Topology B-Tree (blocking not shown).

Proof: The structure of a topology B-tree $\mathcal{T}'$ is shown in Figure 1. Consider a super cluster x at layer l in the corresponding stratified topology tree $\mathcal{T}$. It contains part of $\mathcal{T}$, and also information about the level tree nodes contained in it. There are some level tree edges going out from x to other super clusters (see Figure 1). Consider a node v at level i within x. Suppose v has a edge e going to the node w in the super cluster y. There are two cases:

Case 1: The parent of node v at level $i+1$ is v itself in $\mathcal{T}$. In this case, the node v is not part of the clustering at level i. So, the level tree edge is inherited by the parent nodes of v and u, and hence by the corresponding super clusters.

Case 2: The parent of node v at level $i+1$ is some other node, say u in $\mathcal{T}$. In this case, either the nodes v and w become part of the same super cluster or we fall back to Case 1.

Hence the total number of super clusters accessed during a reclustering operation is proportional to the number of layers or the depth of stratified topology tree $\mathcal{T}$, which is $O(\log_B N)$. Since each super cluster belongs to exactly one block in $\mathcal{T}'$ (see lemma 5), the number of blocks accessed during a reclustering operation on a topology B-tree is also $O(\log_B N)$. ■

Using the above lemmas we can show the following:

Theorem 7. *The dynamic operations insert, delete, and swap on the topology B-tree use $O(\log_B N)$ i/o's. The intersect operation at any node in the base tree uses $O(\log_B N)$ i/o's, and the deepest-intersect query uses $O(k * \log_B N)$ i/o's, where k is the number of nodes identified by the search. The arithmetic expression evaluation, and the maintenance of the dynamic expression tree uses $\bar{O}(\log_B N)$ i/o's.*

Proof: The claim follows from the lemmas 3, 4, 5, and 6, and from the discussions above. The topology tree built on an expression tree does not support search operation for a leaf node, and hence it requires ungoverned hierarchical B-lists for reclustering operation. As a result, we get amortized complexity as shown in lemma 3. However, for implementing other operations we assume the underlying tree supports search operation for locating a leaf. Once the search is done, we can use the path information obtained during reclustering. ■

6 Applications

In this section we consider two fundamental problems in computational geometry, and discuss dynamic solutions for those problems using our topology B-tree. Both the applications involve tree structure which supports top-down search from the root to locate any leaf. Hence, we use a topology B-tree which is built using governed hierarchical B-lists.

6.1 Dynamic Approximate Nearest Neighbor

The problem we consider here is as follows: Given a set S of points in d dimensions (d is a constant), a query point p, a metric L_t, and a parameter ϵ, find a point $q \in S$ such that $\frac{dist(p,q)}{dist(p,p^*)} \leq 1 + \epsilon$, where $dist(.,.)$ represents the distance between two points in L_t metric, and p^* is the actual nearest neighbor of p in S. Our goal is to come up with a data structure to answer this *approximate nearest neighbor* query, and to maintain the data structure under the operations of insertion and deletion of points in S.

Arya and Mount [3] first presented a (static) linear-space data structure for the approximate nearest neighbor problem that answered queries in $O(\log^3 N)$ time, which was later improved by Arya *et al.* [4] to $O(\log N)$ time and also made dynamic. All the previous work on this problem is for the internal memory model.

We give a brief outline of their method here. The set of points in S in d dimensions is partitioned into boxes of "good" aspect ratio. The partitioning is done by repeatedly splitting boxes of points into two boxes using hyperplanes parallel to one of the d dimensions. The boxes are arranged in a binary tree with leaves representing the boxes containing some constant number of points, and the internal nodes containing boxes which are the union of the boxes contained in the children[3]. The tree of boxes can have linear depth in the worst case. But a topology tree is constructed on top of it to obtain a balanced structure.

The algorithm of Arya *et. al.* [4] can be characterized as a method that performs constant number of point location queries on the topology tree constructed, maintaining the closest point found so far. This observation helps us to "externalize" the algorithm. We represent the tree of boxes obtained by partitioning the point space using a topology tree whose nodes are augmented with sets of constant size. The tree of boxes supports searching for a leaf node, and hence we use a topology B-tree built using governed hierarchical B-lists.

Using topology B-tree, we can compute the approximate nearest neighbor for a query point using $O(\log_B N)$ i/o's. We can also update the data structure using $O(\log_B N)$ i/o's, during the operations of insertion and deletion of points. We give the details in the full version.

6.2 Dynamic Closest Pair

Recently, Bespamyatnikh [5] has developed an algorithm to maintain the closest pair of points in a point set S in optimal worst case $O(\log N)$ time for insertions and deletions. The space requirement is linear in N. We adapt his algorithm to the present framework in order to externalize it, using the same box decomposition used in the preceding section for computing approximate nearest neighbor queries.

Callahan and Kosaraju [6] have shown how to maintain such a box decomposition of S under point insertions and deletions using only algebraic operations. In the present framework, this box decomposition corresponds to the tree T_0 in which each node v is labeled by a rectangle $R(v)$. The adapted algorithm can be modified to require $O(\log_B N)$ i/o's in the external-memory model.

Bespamyatnikh maintains a linear-size subset E of the set of distinct pairs of S such that E is guaranteed to contain the closest pair. As in the $O(\log^2 N)$ algorithm of [6], which solves a more general class of dynamic problems, these pairs are then maintained on a heap, resulting in the maintenance of the closest pair. The set E must satisfy certain invariants, which we now recast in the terminology of Callahan and Kosaraju [6].

Let $l_{\max}(R)$ denote the longest dimension of some rectangle R, let $p(v)$ denote the parent of v in T_0 (where the point a is used interchangeably with the leaf representing it), and let $d_{\min}$ and $d_{\max}$ denote, respectively the minimum and maximum distances between two point sets. We also introduce constant parameters α and β that must be adjusted to reconcile the new definitions with those of Bespamyatnikh.

First, we define a *rejected pair* to be an ordered pair (a, b) of points from S such that there exists v in the box decomposition T_0 satisfying the following:

1. $a \notin R(v)$,

[3] There are other types of nodes also, representing *doughnut cells*, but we omit the details here.

2. $l_{\max}(R(v)) \leq \alpha l_{\max}(R(p(a)))$,
3. $d_{\min}(R(p(a)), R(p(v))) \leq \beta l_{\max}(R(p(v)))$, and
4. $d_{\max}(a, R(v)) < d(a, b)$.

Then we maintain the invariant that for all $a, b \in S$, $\{a, b\} \in E$ unless (a, b) or (b, a) is a rejected pair. For sufficiently small α and β, the conditions for rejection are stronger than those of Bespamyatnikh, and will therefore suffice to prove correctness.

Second, we define E_p to be the set of all those pairs in E that contain the point p, and let N_d denote some d-dependent constant. We maintain the additional invariant that for all $p \in S$, $|E_p| \leq N_d$. Note that this gives us a linear bound on $|E|$. This requirement is justified by a theorem of Bespamyatnikh deriving a constant N_d such that for any p such that $|E_p| > N_d$, there is some $\{p, q\}$ in $|E_p|$ such that either (p, q) or (q, p) is a rejected pair.

There are two kinds of searches we must perform in order to maintain the set E. First, for any point p, we must be able to retrieve a set E_p containing at most N_d points. This is necessary for restricting the size of these sets under updates when we add new pairs to E. Second, we must be able to retrieve a set $A(v)$ for any v that contains all $p \in S$ such that there exists a (p, q) that is a rejected pair by virtue of the rectangle $R(v)$. This is necessary for determining when new pairs must be added to E to account for the deletion of some point. Bespamyatnikh has shown that the size of $A(v)$ is constant. Intuitively, this follows from conditions 2 and 3, which allow us to apply a packing argument. See [5] for a proof that these two operations are sufficient.

The above searches can be reduced to an invocation of an approximate nearest neighbor search of the preceding section and a search defined in [6] in which we construct a *δ-rendering* of a given d-cube C, denoted $\rho_\delta(C)$. Intuitively, $\rho_\delta(C)$ is an approximate covering of the points in C using a constant number of rectangles from T_0. More formally, $\rho_\delta(C)$ is a set containing all tree nodes w such that $l_{\max}(R(w)) \leq \delta l(C) < l_{\max}(R(p(w)))$, and $R(w)$ intersects C. It is straightforward to show by a packing argument that the size of a δ-rendering is $O(1)$ (with constants dependent on δ and d).

We can implement the construction of a δ-rendering as a search on the topology tree that is much like point location, but in which we may need to descend into both children of a cluster, resulting in a search subtree of the topology tree rather than a single path. The objects for such a "truncated" deepest-intersect query now correspond to rectangle intersection rather than point inclusion. Details of this search will be given in the final version. Because the number of leaves in such a search will be bounded by the size of the δ-rendering, the total number of i/o's will remain bounded by the depth of the search tree, and will therefore have complexity $O(\log_B N)$ in the external memory model.

Computing E_p: Given a point p, we compute E_p as follows. First, we find the approximate nearest neighbor of p using the algorithm of the preceding section. Let r denote the distance to this neighbor. Clearly r is at least the distance to the actual nearest neighbor. Moreover, it can be greater by a factor of at most $(1 + \epsilon)$. The value of ϵ used is not critical, though it may be chosen to optimize efficiency.

We now let C be the cube centered at p with sides of length $2r$, and compute $\rho_\delta(C)$ for δ sufficiently small that the diameter of each $R(w)$ is bounded above by $r(1+\epsilon)^{-1}$. We let E_p consist of all those points q such that $w \in \rho_\delta(C)$ and $S \cap R(w) = \{q\}$. That is, we rule out any points contained in boxes in which there is more than one point. Note that for any $q \in S \cap R(w)$ such that $w \in \rho_\delta(C)$ and $|S \cap R(w)| > 1$, one can construct a v (a descendant of w) such that (q, p) is a rejected pair. Hence, we may maintain the first invariant while rejecting such q. It is even easier to verify that we may reject all points not covered by boxes in $\rho_\delta(C)$. We derive the constant N_d by bounding $|\rho_\delta(C)|$ in terms of d and δ.

Computing $\boldsymbol{A(v)}$: For this computation, we recall conditions 2 and 3, and consider the set of all $a \in A(v)$. One can easily verify from condition 3 that $R(p(a))$ must intersect a cube C of length $(2\beta + 1)l_{\max}(R(p(v)))$ on a side. For sufficiently small δ, it suffices to consider those a such that there exists $w \in \rho_\delta(C)$ with $S \cap R(w) = \{a\}$. Once again, we may rule out any w such that $R(w)$ contains multiple points. In this case, the reason is that $R(p(a))$ would then violate condition 2.

Recall that the number of i/o's needed to perform an approximate nearest neighbor query and to construct $\rho_\delta(C)$ is $O(\log_B N)$. It follows that the above operations can be performed in $O(\log_B N)$ i/o's. We use standard B-trees to implement heap maintenance in the same i/o complexity. Combining this with the result of [5], we obtain an algorithm for closest-pair maintenance using $O(\log_B N)$ i/o's.

7 Conclusions

We give an efficient method for maintaining arbitrary rooted binary trees in external memory in a dynamic fashion. We show how to perform the dynamic expression tree updates and how these can be applied to solve some interesting dynamic computational geometry problems in external memory. We believe there are other applications, as well, such as approximate range searching [17].

Acknowledgements

We would like to thank David Mount for several helpful discussions concerning the topics of this paper.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. Lars Arge. The buffer tree: A new technique for optimal i/o algorithms. In *Proc. on Fourth Workshop on Algorithms and Data Structures*, 1995.
3. S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.

4. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582, 1994.
5. Sergei N. Bespamyatnikh. An optimal algorithm for closest pair maintenance. In *Proceedings 11th Annual Symposium on Computational Geometry*, 1995.
6. P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *Proc. 6th ACM-SIAM Symp. on Discrete Algorithms*, pages 263–272, 1995.
7. R. F. Cohen and R. Tamassia. Dynamic expression trees and their applications. In *Proc. 2nd ACM-SIAM Sympos. Discrete Algorithms*, pages 52–61, 1991.
8. D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11:121–137, 1979.
9. G. N. Frederickson. Ambivalent data structures for dynamic 2-edge connectivity and k-smallest spanning trees. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 632–641, 1991.
10. G. N. Frederickson. A data structure for dynamically maintaining rooted trees. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 175–184, 1993.
11. Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS 93)*, pages 714–723, 1993.
12. O. Gunther and H.-J. Schek. Advances in spatial databases. In *Proc. 2nd Symposium, SSD '91*, volume 525 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
13. P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. 12th ACM SIGACT-SIGMOD-SIGART Conf. Princ. Database Sys.*, pages 233–243, 1993.
14. Robert Laurini and Derek Thompson. *Fundamentals of Spatial Information Systems*. A.P.I.C. Series. Academic Press, 1992.
15. D. Maier and S. C. Salveter. Hysterical B-trees. *Information Processing Letters*, 12(4):199–202, 1981.
16. J. Matoušek. Reporting points in halfspaces. *Comput. Geom. Theory Appl.*, 2(3):169–186, 1992.
17. D. Mount and S. Arya. Approximate range searching. In *Proc. 11th ACM Symp. on Computational Geometry*, 1995.
18. M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. In *Proceedings of the 12th Annual ACM Symposium on Principles of Database Systems (PODS '93)*, pages 222–232, 1993.
19. M. H. Overmars, M. H. M. Smid, M. T. de Berg, and M. J. van Kreveld. Maintaining range trees in secondary memory, part I: partitions. *Acta Inform.*, 27:423–452, 1990.
20. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.
21. M. H. M. Smid and M. H. Overmars. Maintaining range trees in secondary memory, part II: lower bounds. *Acta Inform.*, 27:453–480, 1990.
22. J. S. Vitter. Efficient memory access in large-scale computation. In *1991 Symposium on Theoretical Aspects of Computer Science (STACS), Lecture Notes in Computer Science,*, Hamburg, 1991. Springer-Verlag.