Chapter 22
Approximate Data Structures with Applications

Yossi Matias* Jeffrey Scott Vitter† Neal E. Young‡

Abstract

In this paper we introduce the notion of *approximate data structures*, in which a small amount of error is tolerated in the output. Approximate data structures trade error of approximation for faster operation, leading to theoretical and practical speedups for a wide variety of algorithms. We give approximate variants of the van Emde Boas data structure, which support the same dynamic operations as the standard van Emde Boas data structure [28, 20], except that answers to queries are approximate. The variants support all operations in constant time provided the error of approximation is $1/\text{polylog}(n)$, and in $O(\log\log n)$ time provided the error is $1/\text{polynomial}(n)$, for n elements in the data structure.

We consider the tolerance of prototypical algorithms to approximate data structures. We study in particular Prim's minimum spanning tree algorithm, Dijkstra's single-source shortest paths algorithm, and an on-line variant of Graham's convex hull algorithm. To obtain output which approximates the desired output with the error of approximation tending to zero, Prim's algorithm requires only linear time, Dijkstra's algorithm requires $O(m \log\log n)$ time, and the on-line variant of Graham's algorithm requires constant amortized time per operation.

1 Introduction

The van Emde Boas data structure (VEB) [28, 20] represents an ordered multiset of integers. The data structure supports query operations for the current minimum and maximum element, the predecessor and successor of a given element, and the element closest to a given number, as well as the operations of insertion and deletion. Each operation requires $O(\log\log U)$ time, where the elements are taken from a universe $\{0, ..., U\}$.

We give variants of the VEB data structure that are faster than the original VEB, but only guarantee approximately correct answers. The notion of approximation is the following: the operations are guaranteed to be consistent with the behavior of the corresponding exact data structure that operates on the elements after they are mapped by a fixed function f. For the multiplicatively approximate variant, the function f preserves the order of any two elements differing by at least a factor of some $1 + \epsilon$. For the additively approximate variant, the function f preserves the order of any two elements differing additively by at least some Δ.

Let the elements be taken from a universe $[1, U]$. On an arithmetic RAM with b-bit words, the times required per operation in our approximate data structures are as follows:

	multiplicative approx. $(1+\epsilon)$	additive approx. Δ
time	$O\left(\log\log_b \frac{\log U}{\epsilon}\right)$	$O\left(\log\log_b \frac{U}{\Delta}\right)$

Under the standard assumption that $b = \Omega(\log U + \log n)$, where n is the measure of input size, the time required is as follows:

$\epsilon, \Delta/U$	$1/\text{polylog}(nU)$	$1/\exp(\text{polylog}(n))$
time	$O(1)$	$O(\log\log n)$

The space requirements of our data structures are $O(\log(U)/\epsilon)$ and $O(U/\Delta)$, respectively. The space can be reduced to close to linear in the number of elements by using dynamic hashing. Specifically, the space

*AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. Email: matias@research.att.com.

†Department of Computer Science, Duke University, Box 90129, Durham, N.C. 27708–0129. Part of this research was done while the author was at Brown University. This research was supported in part by National Science Foundation grant CCR–9007851 and by Army Research Office grant DAAL03–91–G–0035. Email: jsv@cs.duke.edu.

‡Computer Science Department, Princeton University. Part of this research was done while the author was at UMIACS, University of Maryland, College Park, MD 20742 and was partially supported by NSF grants CCR–8906949 and CCR–9111348. Email: ney@cs.princeton.edu.

needed is $O(|S| + |f(S)| \cdot t)$, where S is the set of elements, f is the fixed function mapping the elements of S (hence, $|f(S)|$ is the number of distinct elements under the mapping), and t is the time required per operation. The overhead incurred by using dynamic hashing is constant per memory access with high probability [6, 5]. Thus, if the data structures are implemented to use nearly linear space, the times given per operation hold only with high probability.

1.1 Description of the data structure. The approach is simple to explain, and we illustrate it for the multiplicative variant with $\epsilon = 1$ and $b = 1 + \lfloor \log U \rfloor$. Let $f(i) = \lfloor \log_2 i \rfloor$ (the index of i's most significant bit). The mapping preserves the order of any two elements differing by more than a factor of two and effectively reduces the universe size to $U' = 1 + \lfloor \log U \rfloor$. On an arithmetic RAM with b-size words, a bit-vector for the mapped elements fits in a single word, so that successor and predecessor queries can be computed with a few bitwise and arithmetic operations. The only additional structures are a linked list of the elements and a dictionary mapping bit indices to list elements.

In general, each of the approximate problems with universe size U reduces to the exact problem with a smaller universe size U': For the case of multiplicative approximation we have size

$$U' = 2\log_2(U)/\epsilon = O(\log_{1+\epsilon} U),$$

and for the case of additive approximation

$$U' = U/\Delta.$$

Each reduction is effectively reversible, yielding an equivalence between each approximate problem and the exact problem with a smaller universe. The equivalence holds generally for any numeric data type whose semantics depend only on the ordering of the elements. The equivalence has an alternate interpretation: each approximate problem is equivalent to the exact problem on a machine with larger words. Thus, it precludes faster approximate variants that don't take advantage of fast operations on words.

For universe sizes bigger than the number of bits in a word, we apply the recursive divide-and-conquer approach from the original VEB data structure. Each operation on a universe of size U' reduces to a single operation on a universe of size $\sqrt{U'}$ plus a few constant time operations. When the universe size is b, only a small constant number of arithmetic and bitwise operations are required. This gives a running time of $O(\log\log_b U')$, where U' is the effective universe size after applying the universe reduction from the approximate to the exact problem.

1.2 Outline. In the next section we motivate our development of approximate VEB data structures by demonstrating how they can be used in three well-known algorithms: Prim's algorithm for minimum spanning trees, Dijkstra's shortest paths algorithm, and an on-line version of the Graham scan for finding convex hulls. Related work is discussed in Section 3. Our model of computation is defined in Section 4. In Section 5, we show how to construct our approximate VEB data structures and we analyze their characteristics. We make concluding remarks in Section 6.

2 Applications

We consider three prototypical applications: to minimum spanning trees, to single-source shortest paths, and to semi-dynamic on-line convex hulls. Our approximate minimum spanning tree algorithm runs in linear time and is arguably simpler and more practical than the two known linear-time MST algorithms. Our approximate single-source shortest paths algorithm is faster than any known algorithm on sparse graphs. Our on-line convex hull algorithm is also the fastest known in its class; previously known techniques require preprocessing and thus are not suitable for on-line or dynamic problems. The first two applications are obtained by substituting our data structures into standard, well-known algorithms. The third is obtained by a straightforward adaptation of an existing algorithm to the on-line case. These examples are considered mainly as prototypical applications. In general, approximate data structures can be used in place of any exact counterpart.

Our results below assume a RAM with a logarithmic word size as our model of computation, described in more detail in Section 4. The proofs are simple and are given in the full paper.

2.1 Minimum spanning trees. For the minimum spanning tree problem, we show the following result about the performance of Prim's algorithm [16, 25, 7] when our approximate VEB data structure is used to implement the priority queue:

THEOREM 2.1. *Given a graph with edge weights in $\{0, .., U\}$, Prim's algorithm, when implemented with our approximate* VEB *with multiplicative error $(1 + \epsilon)$, finds a $(1 + \epsilon)$-approximate minimum spanning tree in an n-node, m-edge graph in $O((n + m)\log(1 + (\log \frac{1}{\epsilon})/\log\log nU))$ time.*

For $1/\epsilon \leq \text{polylog}(nU)$, Theorem 2.1 gives a linear-time algorithm. This algorithm is arguably simpler and more practical than the two known linear-time MST algorithms. This application is a prototypical

example for which the use of an approximate data structure is equivalent to slightly perturbing the input. Approximate data structures can be "plugged in" to such algorithms without modifying the algorithm.

2.2 Shortest paths. For the single-source shortest paths problem, we get the following result by using an approximate VEB data structure as a priority queue in Dijkstra's algorithm (see, e.g., [27, Thm 7.6]):

THEOREM 2.2. *Given a graph with edge weights in* $\{0, ..., U\}$ *and any* $0 < \epsilon \leq 2$, *Dijkstra's algorithm, when implemented with our approximate* VEB *with multiplicative error* $(1 + \epsilon/(2n))$, *computes single-source shortest path distances within a factor of* $(1 + \epsilon)$ *in* $O((n + m)\log(\log \frac{n}{\epsilon} / \log\log U))$ *time.*

If $\log(1/\epsilon) \leq \text{polylog}(n)\log\log U$, the algorithm runs in $O((n + m)\log\log n)$ time — faster than any known algorithm on sparse graphs — and is simpler than theoretically competitive algorithms. This is a prototypical example of an algorithm for which the error increases by the multiplicative factor at each step. If such an algorithm runs in polynomial time, then $O(\log\log n)$ time per VEB operation can be obtained with insignificant net error. Again, this speed-up can be obtained with no adaptation of the original algorithm.

Analysis. The proof of Theorem 2.2 follows the proof of the exact shortest paths algorithm (see, e.g., [27, Thm 7.6]). The crux of the proof is an inductive claim, saying that any vertex w that becomes labeled during or after the scanning of a vertex v also satisfies $dist(w) \geq dist(v)$, where $dist(w)$ is a so-called tentative distance from the source to w. When using a $(1+\epsilon)$-approximate VEB data structure to implement the priority queue, the inductive claim is replaced by

$$dist(w) \geq dist(v)/(1 + \epsilon/(2n))^i ,$$

where vertex v is the ith vertex to be scanned. Thus, the accumulated multiplicative error is bounded by

$$(1 + \epsilon/(2n))^n \leq e^{\epsilon/2} \leq (1 + \epsilon) .$$

We leave the details to the full paper, and only note that it is not difficult to devise an example where the error is actually accumulated exponentially at each iteration.

2.3 On-line convex hull. Finally, we consider the semi-dynamic on-line convex hull problem. In this problem, a set of planar points is processed in sequence. After each point is processed, the convex hull of the points given so far must be computed. Queries of the form "is x in the current hull?" can also be given at any time. For the approximate version, the hull computed and the answers given must be consistent with a $(1+\Delta)$-approximate hull, which is contained within the true convex hull such that the distance of any point on the true hull to the approximate hull is $O(\Delta)$ times the diameter.

We show the following result about the Graham scan algorithm [12] when run using our approximate VEB data structure:

THEOREM 2.3. *The on-line* $(1 + \Delta)$*-approximate convex hull can be computed by a Graham scan in constant amortized time per update if* $\Delta \geq \log^{-c} n$ *for any fixed* $c > 0$, *and in* $O(\log\log n)$ *amortized time per update if* $\Delta \geq n^{-c}$.

This represents the first constant-amortized-time-per-query approximation algorithm for the on-line problem. This example demonstrates the usefulness of approximate data structures for dynamic/on-line problems. Related approximate sorting techniques require preprocessing, which precludes their use for on-line problems.

Analysis. Graham's scan algorithm is based on scanning the points according to an order determined by their polar representation, relative to a point that is in the convex hull, and maintaining the convex hull via local corrections. We adapt Graham's scan to obtain our on-line algorithm, as sketched below. As an invariant, we have a set of points that are in the intermediate convex hull, stored in an approximate VEB according to their angular coordinates. The universe is $[0, 2\pi]$ with a Δ additive error, which can be interpreted as the perturbation error of points in their angular coordinate, without changing their values in the distance coordinates. This results in point displacements of at most $(1 + \Delta)$ times the diameter of the convex hull.

Given a new point, its successor and predecessor in the VEB are found, and the operations required to check the convex hull and, if necessary, to correct it are carried on, as in Graham's algorithm [12]. These operations may include the insertion of the new point into the VEB (if the point is on the convex hull) and the possible deletion of other points. Since each point can only be deleted once from the convex hull, the amortized number of VEB operations per point is constant.

3 Related work

Our work was inspired by and improves upon data structures developed for use in dynamic random variate generation by Matias, Vitter, and Ni [19].

Approximation techniques such as rounding and bucketing have been widely used in algorithm design.

This is the first work we know of that gives a general-purpose approximate data structure.

Finite precision arithmetic. The sensitivity of algorithms to approximate data structures is related in spirit to the challenging problems that arise from various types of error in numeric computations. Such errors has been studied, for example, in the context of computational geometry [8, 9, 13, 14, 21, 22, 23]. We discuss this further in Section 6.

Approximate sorting. Bern, Karloff, Raghavan, and Schieber [3] introduced *approximate sorting* and applied it to several geometric problems. Their results include an $O((n \log\log n)/\epsilon)$-time algorithm that finds a $(1+\epsilon)$-approximate Euclidean minimum spanning tree. They also gave an $O(n)$-time algorithm that finds a $(1+\Delta)$-approximate convex hull for any $\Delta \geq 1/\text{polynomial}$.

In a loose sense, approximate VEB data structures generalize approximate sorting. The advantages of an approximate VEB are the following. An approximate VEB bounds the error for each element individually. Thus, an approximate VEB is applicable for problems such as the general minimum spanning tree problem, for which the answer depends on only a subset of the elements. The approximate sort of Bern *et al.* bounds the *net* error, which is not sufficient for such problems. More importantly, a VEB is dynamic, so is applicable to dynamic problems such as on-line convex hull and in algorithms such as Dijkstra's algorithm in which the elements to be ordered are not known in advance. Sorting requires precomputation, so is not applicable to such problems.

Convex hull algorithms. There are several relevant works for the on-line convex hull problem. Shamos (see, e.g., [26]) gave an on-line algorithm for (exact) convex hull that takes $O(\log n)$ amortized time per update step. Preparata [24] gave a *real-time* on-line (exact) convex hull algorithm with $O(\log n)$-time worst-case time per update step. Bentley, Faust, and Preparata [2] give an $O(n + 1/\Delta)$-time algorithm that finds a $(1+\Delta)$-approximate convex hull. Their result was superseded by the result of Bern *et al.* mentioned above. Janardan [15] gave an algorithm maintaining a fully dynamic $(1+\Delta)$-approximate convex hull (allowing deletion of points) in $O(\log(n)/\Delta)$ time per request. Our on-line approximation algorithm is based on Graham's scan algorithm [12] and can be viewed as a combination of the algorithms by Shamos and by Bentley *et al.*, with the replacement of an exact VEB data structure by an approximate variant.

Computation with large words. Kirkpatrick and Reich [17] considered exact sorting with large words, giving upper and lower bounds. Their interest was theoretical, but Lemma 5.1, which in some sense says that maintaining an approximate VEB data structure is equivalent to maintaining an exact counterpart using larger words, suggests that lower bounds on computations with large words are relevant to *approximate* sorting and data structures.

Exploiting the power of RAM. Fredman and Willard have considered a number of data structures taking advantage of arithmetic and bitwise operations on words of size $O(\log U)$. In [10], they presented the *fusion tree* data structure. Briefly, fusion trees implement the VEB data type in time $O(\log n/\log\log n)$. They also presented an *atomic heap* data structure [11] based on their fusion tree and used it to obtain a linear-time minimum spanning tree algorithm and an $O(m + n\log n/\log\log n)$-time single-source shortest paths algorithm. Willard [29] also considered similar applications to related geometric and searching problems. Generally, these works assume a machine model similar to ours and demonstrate remarkable theoretical consequences of the model. On the other hand, they are more complicated and involve larger constants.

Subsequent to our work Klein and Tarjan recently announced a randomized minimum spanning tree algorithm that requires only expected linear time [18]. Arguably, our algorithm is simpler and more practical.

4 Model of computation

The model of computation assumed in this paper is a modernized version of the *random access machine* (RAM). Many RAM models of a similar nature have been defined in the literature, dating back to the early 1960s [1]. Our RAM model is a realistic variant of the logarithmic-cost RAM [1]: the model assumes constant-time exact binary integer arithmetic (+, −, ×, **div**), bitwise operations (**left-shift**, **right-shift**, **bitwise-xor**, **bitwise-and**), and addressing operations on words of size b. Put another way, the word size of the RAM is b. We assume that numbers are of the form $i + j/2^b$, where i and j are integers with $0 \leq i, j < 2^b$, and that the numbers are represented with two words, the first holding i and the second holding j. For simplicity of exposition, we use the "most-significant-bit" function $\text{MSB}(x) = \lfloor \log_2 x \rfloor$; it can be implemented in small constant time via the previously mentioned operations and has lower circuit complexity than, e.g., division.

5 Fast approximate data structures

This section gives the details of our approximate VEB data structure. First we give the relevant semantics

and notations. The operations supported are:

$N \leftarrow$ INSERT(x, d),
DELETE(N),
$N \leftarrow$ SEARCH(x),
$N \leftarrow$ MINIMUM( ),
$N \leftarrow$ MAXIMUM( ),
$N \leftarrow$ PREDECESSOR(N),
$N \leftarrow$ SUCCESSOR(N),
$d \leftarrow$ DATA(N), and
$x \leftarrow$ ELEMENT(N).

The INSERT operation and the query operations return the *name* N of the element in question. The name is just a pointer into the data structure allowing constant-time access to the element. Subsequent operations on the element are passed this pointer so they can access the element in constant time. INSERT takes an additional parameter d, an arbitrary auxiliary data item. SEARCH(x), where x is a real number (but not necessarily an element), returns the name of the largest element less than or equal to x. For the approximate variants, the query operations are approximate in that the element returned by the query is within a $(1+\epsilon)$ relative factor or a Δ absolute amount of the correct value. Operations ELEMENT(N) and DATA(N), given an element's name N, return the element and its data item, respectively.

The universe (specified by U) and, for the approximate variants, the error of approximation (ϵ or Δ) are specified when the data structure is instantiated.

5.1 Equivalence of various approximations. The lemma below assumes a logarithmic word-size RAM. The notion of equivalence between data structures is that, given one of the data structures, the other can be simulated with constant-time overhead per operation.

LEMMA 5.1. *The problem of representing a multiplicatively* $(1+\epsilon)$*-approximate* VEB *on universe* $[1, U]$ *is equivalent to the problem of representing an exact* VEB *on universe* $\{0, 1, ..., O(\log_{1+\epsilon} U)\}$.

The problem of representing an additively Δ*-approximate* VEB *on universe* $[0, U]$ *is equivalent to the problem of representing an exact* VEB *on universe* $\{0, 1, ..., O(U/\Delta)\}$.

Proof. Assume we have a data structure for the exact data type on the specified universe. To simulate the multiplicatively approximate data structure, the natural mapping to apply to the elements (as discussed previously) is $x \mapsto \lfloor \log_{1+\epsilon} x \rfloor$. Instead, we map x to approximately $\frac{1}{\ln 2}(\log_{1+\epsilon} x) \approx (\log_2 x)/\epsilon$ and we use a mapping that is faster to compute: Let $k = \lceil \log_2 \frac{1}{\epsilon} \rceil$, let $x = i + j/2^b$, and let $\ell = \text{MSB}(i)$. We use the mapping f that maps x to

$$\begin{aligned} \ell \quad & \textbf{left-shift}(k) \\ & \textbf{bitwise-or}\,(i \ \textbf{right-shift}\,(\ell - k)) \\ & \textbf{bitwise-xor}\,(1 \ \textbf{left-shift}\ k) \\ & \textbf{bitwise-or}\,(j \ \textbf{right-shift}\,(b + \ell - k)). \end{aligned}$$

If $\ell < k$, then to right-shift by $(\ell - k)$ means to left-shift by $(k - \ell)$. Note that in this case the fractional part of x is shifted in.

This mapping effectively maps x to the lexicographically ordered pair $\langle \text{MSB}(x), y \rangle$, where y represents the bits with indices $(\ell - 1)$ though $(\ell - k)$ in x. The first part of the tuple distinguishes between any two x values that differ in their most significant bit. If two x values have $\text{MSB}(x) = \ell$, then it suffices to distinguish them if they differ additively by $2^{\ell-k}$. The second part of the tuple suffices for this.

Note that $f(1) = 0$ and $f(U) < 2^{k+1} \log_2 U = O(\log_{1+\epsilon} U)$. This shows one direction of the first part. The other direction of the first part is easily shown by essentially inverting the above mapping, so that distinct elements map to elements that differ by at least a factor of $1 + \epsilon$. Finally, the second part follows by taking the mapping $(x \mapsto x \ \textbf{div}\ \Delta)$ and its inverse.

5.2 Implementations. Lemma 5.1 reduces the approximate problems to the exact problem with smaller universe size. This section gives an appropriate solution to the exact problem. If an approximate variant is to be implemented, we assume the elements have already been mapped by the constant-time function f in Lemma 5.1. The model of computation is a RAM with b-bit words.

A *dictionary* data structure supports update operations SET$(key, value)$ and UNSET(key) and query operation LOOK-UP(key) (returning the value, if any, associated with the key). It is well known how to implement a dictionary by hashing in space proportional to the number of elements in the dictionary or in an array of size proportional to the key space. In either case, all dictionary operations require only constant time. In the former case, the time is constant with high probability [6, 5]; in the latter case, a well-known trick is required to instantiate the dictionary in constant time.

Each instance of our data structure will have a doubly-linked list of element/datum pairs. The list is ordered by the ordering induced by the elements. The name of each element is a pointer to its record in this list.

If the set to be stored is a multiset, as will generally be the case in simulating an approximate variant, then

the elements will be replaced by buckets, which are doubly-linked lists holding the multiple occurrences of an element. Each occurrence holds a pointer to its bucket. In this case the name of each element is a pointer to its record within its bucket.

Each instance will also have a dictionary mapping each element in the set to its name. If the set is a multiset, it will map each element to its bucket. In general, the universe, determined when the data structure is instantiated, is of the form $\{L, ..., U\}$. Each instance records the appropriate L and U values and subtracts L from each element, so that the effective universe is $\{0, ..., U - L\}$.

The ordered list and the dictionary suffice to support constant-time PREDECESSOR, SUCCESSOR, MINIMUM, and MAXIMUM operations. The other operations use the list and dictionary as follows. INSERT(i) finds the predecessor-to-be of i by calling SEARCH(i), inserts i into the list after the predecessor, and updates the dictionary. If S is a multiset, i is inserted instead into its bucket and the dictionary is updated only if the bucket didn't previously exist. DELETE(N) deletes the element from the list (or from its bucket) and updates the dictionary appropriately.

How SEARCH works depends on the size of the universe. The remainder of this section describes SEARCH queries and how INSERT and DELETE maintain the additional structure needed to support SEARCH queries.

5.3 Bit-vectors. For a universe of size b, the additional structure required is a single b-bit word w. As described in Section 1.1, the word represents a bit vector; the ith bit is 1 iff the dictionary contains an element i. INSERT sets this bit; DELETE unsets it if no occurrences of i remain in the set. Setting or unsetting bits can be done with a few constant time operations.

The SEARCH(i) operation is implemented as follows. If the list is empty or i is less than the minimum element, return **nil**. Otherwise, let

$$j \leftarrow \text{MSB}(w \textbf{ bitwise-and } ((1 \textbf{ left-shift } i) - 1)),$$

i.e., let j be the index of the most significant 1-bit in w that is at most as significant as the ith bit. Return j's name from the dictionary.

Analysis. On universes of size b, all operations require only a few constant-time operations. If hashing is used to implement the dictionary, the total space (number of words) required at any time is proportional to the number of elements currently in the set.

5.4 Intermediate data structure. The fully recursive data structure is a straightforward modification of the original van Emde Boas data structure. For those not familiar with the original data structure, we first give an intermediate data structure that is conceptually simpler as a stepping stone. The additional data structures to support SEARCH(i) for a universe $\{0, 1, ..., b^j - 1\}$ are as follows.

Divide the problem into $b + 1$ subproblems: if the current set of elements is S, let S_k denote the set $\{i \in S : i \textbf{ div } b^{j-1} = k\}$. Inductively maintain a VEB data structure for each non-empty set S_k. Note that the universe size for each S_k is b^{j-1}. Each S_k can be a multiset only if S is.

Let T denote the set $\{k : S_k \text{ not empty }\}$. Inductively maintain a VEB data structure for the set T. The datum for each element k is the data structure for S_k. Note that the universe size for T is b. Note also that T need not support multi-elements.

Implement SEARCH(i) as follows. If i is in the dictionary, return i's name. Otherwise, determine k such that i would be in S_k if i were in S. Recursively search in T for the largest element k' less than or equal to k. If $k' < k$ or i is less than the minimum element of S_k, return the maximum element of $S_{k'}$. Otherwise, recursively search for the largest element less than or equal to i in S_k and return it.

INSERT and DELETE maintain the additional data structures as follows. INSERT(i) inserts i recursively into the appropriate S_k. If S_k was previously empty, it creates the data structure for S_k and recursively inserts k into T. DELETE(N) recursively deletes the element from the appropriate S_k. If S_k becomes empty, it deletes k from T.

Analysis. Because the universe of the set T is of size b, all operations maintaining T take constant time. Thus, each SEARCH, INSERT, and DELETE for a set with universe of size $U = b^j$ requires a few constant-time operations and possibly one recursive call on a universe of size b^{j-1}. Thus, each such operation requires $O(j) = O(\log_b U)$ time.

To analyze the space requirement, note that the size of the data structure depends only on the elements in the current set. Assuming hashing is used to implement the dictionaries, the space required is proportional to the number of elements in the current set plus the space that would have been required if the *distinct* elements of the current set had simply been inserted into the data structure. The latter space would be at worst proportional to the time taken for the insertions. Thus, the total space required is proportional to the number of elements plus $O(\log_b U)$ times the number of distinct elements.

5.5 Full recursion. We exponentially decrease the above time by balancing the subdivision of the problem exactly as is done in the original van Emde Boas data structure.

The first modification is to balance the universe sizes of the set T and the sets $\{S_k\}$. Assume the universe size is b^{2^j}. Note that $b^{2^j} = b^{2^{j-1}} \times b^{2^{j-1}}$. Define $S_k = \{i \in S : i \text{ div } b^{2^{j-1}} = k\}$ and define $T = \{k : S_k \text{ is not empty}\}$. Note that the universe size of each S_k *and* of T is $b^{2^{j-1}}$.

With this modification, SEARCH, INSERT, and DELETE are still well defined. Inspection of SEARCH shows that if SEARCH finds k in T, it does so in constant time, and otherwise it does not search recursively in S_k. Thus, only one non-constant-time recursion is required, into a universe of size $b^{2^{j-1}}$. Thus, SEARCH requires $O(j)$ time.

INSERT and DELETE, however, do not quite have this nice property. In the event that S_k was previously empty, INSERT descends recursively into both S_k and T. Similarly, when S_k becomes empty, DELETE descends recursively into both S_k and T.

The following modification to the data structure fixes this problem, just as in the original van Emde Boas data structure. Note that INSERT only updates T when an element is inserted into an empty S_k. Similarly, DELETE only updates T when the last element is deleted from the set S_k. Modify the data structure (and all recursive data structures) so that the recursive data structures exist only when $|S| \geq 2$. When $|S| = 1$, the single element is simply held in the list. Thus, insertion into an empty set and deletion from a set of one element require only constant time. This insures that if INSERT or DELETE spends more than constant time in T, it will require only constant time in S_k.

This modification requires that when S has one element and a new element is inserted, INSERT instantiates the recursive data structures and inserts both elements appropriately. The first element inserted will bring both T and some S_k to size one; this requires constant time. If the second element is inserted into the same set S_k as the first element, T is unchanged. Otherwise, the insertion into its newly created set $S_{k'}$ requires only constant time. In either case, only one non-constant-time recursion is required.

Similarly, when S has two elements and one of them is deleted, after the appropriate recursive deletions, DELETE destroys the recursive data structures and leaves the data structure holding just the single remaining element. If the two elements were in the same set S_k, then T was already of size one, so only the deletion from S_k requires more than constant time. Otherwise, each set S_k and $S_{k'}$ was already of size one, so only the deletion of the second element from T took more than constant time.

Analysis. With the two modifications, each SEARCH, INSERT, and DELETE for a universe of size $U = b^{2^j}$ requires at most one non-constant-time recursive call, on a set with universe size $b^{2^{j-1}}$. Thus, the time required for each operation is $O(j) = O(\log \log_b U)$. As for the intermediate data structure, the total space is at worst proportional to the number of elements, plus the time per operation (now $O(\log \log_b U)$) times the number of distinct elements.

6 Conclusions

The approximate data structures described in this paper are simple and efficient. No large constants are hidden in the asymptotic notations—in fact, a "back of the envelope" calculation indicates significant speed-up in comparison to the standard van Emde Boas data structure. The degree of speed-up in practice will depend upon the machines on which they are implemented. Machines on which binary arithmetic and bitwise operations on words are nearly as fast as, say, comparison between two words will obtain the most speed-up. Practically, our results encourage the development of machines which support fast binary arithmetic and bitwise operations on large words. Theoretically, our results suggest the need for a model of computation that more accurately measures the cost of operations that are considered to require constant time in traditional models.

The applicability of approximate data structures to specific algorithms depends on the robustness of such algorithms to inaccurate intermediate computations. In this sense, the use of approximate data structures has an effect similar to computational errors that arise from the use of finite precision arithmetic. In recent years there has been an increasing interest in studying the effect of such errors on algorithms. Of particular interest were algorithms in computational geometry. Frameworks such as the "epsilon geometry" of Guibas, Salesin and Stolfi [14] may be therefore relevant in our context. The "robust algorithms" described by Fortune and Milenkovic [8, 9, 21, 22, 23] are natural candidates for approximate data structures.

Expanding the range of applications of approximate data structures is a fruitful area for further research. Other possible candidates include algorithms in computational geometry that use the well-known sweeping technique, provided that they are appropriately robust. For instance, in the sweeping algorithm for the line arrangement problem with approximate arithmetic, pre-

sented by Fortune and Milenkovic [9], the priority queue can be replaced by an approximate priority queue with minor adjustments, to obtain an output with similar accuracy. If the sweeping algorithm of Chew and Fortune [4] can be shown to be appropriately robust then the use of the van Emde Boas priority queue there can be replaced by an approximate variant; an improved running time may imply better performance for algorithms described in [3].

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1974.

[2] J. L. Bentley, M. G. Faust, and F. P. Preparata. Approximation algorithms for convex hulls. *Communications of the ACM*, 25(1):64–68, 1982.

[3] M. W. Bern, H. J. Karloff, P. Raghavan, and B. Schieber. Fast geometric approximation techniques and geometric embedding problems. *Theoretical Computer Science*, 106:265–281, 1992.

[4] L. P. Chew and S. Fortune. Sorting helps for Voronoi diagrams. In *19th Symp. on Mathematical Programming, Japan*, 1988.

[5] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial Hash Functions are Reliable. *Proc. of 19th International Colloquium on Automata Languages and Programming*, Springer LNCS 623, 235–246, 1992.

[6] M. Dietzfelbinger and F. Meyer auf der Heide. A New Universal Class of Hash Functions and Dynamic Hashing in Real Time, In *Proc. of 17th International Colloquium on Automata Languages and Programming*, Springer LNCS 443: 6–19, 1990.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] S. Fortune. Stable maintenance of point-set triangulation in two dimensions. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, 1989.

[9] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. of the 7th Annual Symposium on Computational Geometry*, pages 334–341, 1991.

[10] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proc. of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 1–7, 1990.

[11] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, pages 719–725, 1990.

[12] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[13] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. *Proc. of the 27th IEEE Annual Symp. on Foundation of Computer Science*, pages 143–152, 1986.

[14] L. I. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: Building robust algorithms from imprecise computations. In *Proc. of the 5th Annual Symposium on Computational Geometry*, pages 208–217, 1989.

[15] R. Janardan. On maintaining the width and diameter of a planar point-set online. In *Proc. 2nd International Symposium on Algorithms*, volume 557 of *Lecture Notes in Computer Science*, pages 137–149. Springer-Verlag, 1991. To appear in International Journal of Computational Geometry & Applications.

[16] V. Jarník. O jistém problému minimálmním. *Práca Moravské Prírodovedecké Spolecnosti*, 6:57–63, 1930. (In Czech).

[17] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28:263–276, 1984.

[18] P. N. Klein and R. E. Tarjan. A linear-time algorithm for minimum spanning tree. *Personal communication, August, 1993.*

[19] Y. Matias, J. S. Vitter, and W.-C. Ni. Dynamic generation of random variates. In *Proc. of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 361–370, 1993.

[20] K. Mehlhorn. *Data Structures and Algorithms.* Springer-Verlag, Berlin, Heidelberg, 1984.

[21] V. Milenkovic. *Verifiable Implementations of Geometric Algorithms using Finite Precision Arithmetic.* PhD thesis, Carnegie-Mellon University, 1988.

[22] V. Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In *Proc. of the 5th Annual Symposium on Computational Geometry*, pages 197–207, 1989.

[23] V. Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, 1989.

[24] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Communications of the ACM*, 22(7):402–405, 1979.

[25] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Tech. J.*, 36:1389–1401, 1957.

[26] F. P. Preparata and M. I. Shamos. *Computational Geometry*, Springer-Verlag, New York, 1985.

[27] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, 1983.

[28] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[29] D. E. Willard. Applications of the fusion tree method to computational geometry and searching. In *Proc. of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, 1992.