Dictionaries Using Variable-Length Keys and Data, with Applications *

Daniel K. Blandford
dkb1@cs.cmu.edu

Guy E. Blelloch
blelloch@cs.cmu.edu

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We consider the problem of maintaining a dynamic dictionary in which both the keys and the associated data are variable-length bit-strings. We present a dictionary structure based on hashing that supports constant time lookup and expected amortized constant time insertion and deletion. To store the key-data pairs $(s_1, t_1) \ldots (s_n, t_n)$, our dictionary structure uses $O(m)$ bits where $m = \sum(\max(|s_i| - \log n, 1) + |t_i|)$ and $|s_i|$ is the length of bit string s_i. We assume a word length $w > \log m$.

We present several applications, including representations for semi-dynamic graphs, ordered sets for integers in a bounded range, cardinal trees with varying cardinality, and simplicial meshes of k dimensions. These results either generalize or simplify previous results.

1 Introduction

There has been significant recent interest in data structures that use near optimal space while supporting fast access [19, 23, 10, 8, 24, 17, 27, 15, 4, 28]. In additional to theoretical interest, such structures have significant practical implications. In recent experimental work [5], for example, it was shown that a compact representation of graphs not only requires much less space than standard representations (*e.g.*, adjacency lists), but in many cases it is faster. This is because it requires less data to be loaded into the cache.

The dictionary problem is to maintain an n-element set of keys s_i with associated data t_i. A dictionary is dynamic if it supports insertion and deletion as well as the lookup operation. In this paper we are interested in dynamic dictionaries in which both the keys and data are variable-length bit-strings. Our main motivation is to use such dictionaries as building blocks for various other applications. We describe applications of our dictionary structure to graphs, cardinal trees with nodes of varying cardinality, ordered sets, and simplicial meshes. These applications either generalize or simplify previously known results. We assume the machine has a word length $w > \log |C|$, where $|C|$ is the number of bits used to represent the collection. We assume the size of each string $|s_i| \geq 1$, $|t_i| \geq 1$ for all bit-strings s_i and t_i.

For fixed-length keys the dictionary problem has been well studied. The information-theoretic lower bound for representing n elements from a universe U is $B = \lceil \binom{|U|}{n} \rceil = n(\log |U| - \log n) + O(n)$. Cleary [11] showed how to achieve $(1 + \epsilon)B + O(n)$ bits with $O(1/\epsilon^2)$ expected time for lookup and insertion while allowing satellite data. His structure used the technique of *quotienting* [21], which involves storing only part of each key in a hash bucket; the part not stored can be reconstructed using the index of the bucket containing the key. Brodnik and Munro [8] describe a static structure using $B + o(B)$ bits and requiring $O(1)$ time for lookup; the structure can be dynamized, increasing the space cost to $O(B)$ bits. That structure does not support satellite data. Pagh [25] showed a static dictionary using $B+o(B)$ bits and $O(1)$ query time that supported satellite data, using ideas similar to Cleary's, but that structure could not be easily dynamized.

Recently Raman and Rao [28] described a dynamic dictionary structure using $B + o(B)$ bits that supports lookup in $O(1)$ time and insertion and deletion in $O(1)$ expected amortized time. The structure allows attaching fixed-length ($|t|$-bit) satellite data to elements: in that case the space bound is $B + n|t| + o(B + n|t|)$ bits. None of this considers variable-bit keys or data.

*This work was supported in part by the National Science Foundation as part of the Aladdin Center (www.aladdin.cmu.edu) under grants ACI-0086093, CCR-0085982, and CCR-0122581.

Our variable-bit dictionary structure can store pairs (s_i, t_i) using $O(m)$ space where $m = \sum_i(\max(1, |s_i| - \log n) + |t_i|)$. Note that if $|s_i|$ is constant and $|t_i|$ is zero then $O(m)$ simplifies to $O(B)$. Our dictionaries support lookup in $O(1)$ time and insertion and deletion in $O(1)$ expected amortized time.

Our dictionary makes use of a simpler structure: an "array" structure that supports an array of n locations $(1, \ldots, n)$ with lookup and update operations. We denote the i^{th} element of an array A as a_i. In our case each location will store a bit-string. We present a data-structure that uses $O(m+w)$ space where $m = \sum_{i=1}^{n} |a_i|$ and w is the machine word length. The structure supports lookups in $O(1)$ worst-case time and updates in $O(1)$ expected amortized time. Note that if all bit-strings were the same length then this would be trivial.

Applications. Using our dictionaries we present succinct dynamic representations for several other data structures. For graphs we support adjacency queries, listing the neighbors of a vertex, and deleting and inserting edges. Insertion and deletion run in $O(1)$ expected amortized time, adjacency queries require $O(1)$ worst case time, and listing neighbors requires $O(1)$ time per neighbor. Given an integer labeling of the vertices, the space required is $O(m + n)$ where $m = \sum_{(u,v)\in E} \log |u - v|$, and $n = |V|$. Any graph from a class satisfying an $n^{1-\epsilon}$ edge-separator theorem ($\epsilon > 0$) can be labeled so that $m < kn$ for some constant k, and hence can be coded in $O(n)$ bits. It is well known, for example, that the class of bounded-degree planar-graphs satisfies an $n^{1/2}$ edge-separator theorem. For graphs with bounded degree this extends previous results [29, 20, 18, 4] by permitting insertion and deletion of edges. We say that the graph is partially dynamic since although it allows dynamic insertions and deletions, the space bound relies on m remaining small. As far as we know this is the first compact dynamic graph representation of any kind.

For ordered sets $S \subset \{0, \ldots, |U| - 1\}$ we support the same operations in the same bounds as recently reported [3], except that the updates are expected amortized time here and were worst case time bounds there. The structure we describe here, however is simpler and quite different from the previous structure. It also allows for attaching a satellite bit string to each key.

For cardinal trees (aka tries) we support a tree in which each node can have a different cardinality. Queries can request the k^{th} child, or the parent of any vertex. Again we can attach satellite bit-strings to each vertex. Updates can add or delete the k^{th} child. For an integer labeled tree the space bound is $O(m)$ where $m = \sum_{v\in V}(\log c(p(v)) + \log |v - p(v)|)$, and $p(v)$ and $c(v)$ are the parent and cardinality of v, respectively. Using an appropriate labeling of the vertices m reduces to $\sum_{v\in V} \log c(p(v))$, which is asymptotically optimal. This generalizes previous results on cardinal trees [2, 27] to varying cardinality. We do not match the optimal constant in the first order term.

For d-simplicial meshes[1] we support insertion and deletion of simplices of dimension d, and returning the neighbors across all faces of dimension $d - 1$. For example in a 3d tetrahedral mesh we can add and delete tetrahedrons, and ask for the neighboring tetrahedron across any of the four faces, if there is one. Given an integer labeling of the vertices, the space required is $O(m+n)$ where $m = \sum_{(a,b,c)\in F}(\log |a-b| + \log |a-c|)$, $n = |V|$, and F are the faces in the 2−skeleton of the mesh. In the bounded degree case this reduces to $m = \sum_{(u,v)\in E}(\log |u - v|)$ where E are the edges in the 1-skeleton of the mesh. As usual, updates take $O(1)$ amortized expected time and queries take $O(1)$ worst case time. We note that we have actually used a similar data structure as described here to implement triangulated and tetrahedral meshes [6]. The experiments show that the tetrahedral mesh allows fast access and updates and uses only about 7 bytes per tetrahedron (compared to 32 bytes per tetrahedron needed for the most compact traditional representation). In that paper we do not give any theoretical bounds on space.

2 Preliminaries

Processor model. In our data structures we assume that the processor word length is w bits, for some $w > \log |C|$, where $|C|$ is the total number of bits consumed by our data structure. That is, we assume that we can use a w-bit word to point to any memory we allocate.

We assume that the processor supports two special operations, `bitSelect` and `bitRank`, defined as follows. Given a bit string s of length w bits, `bitSelect`(s, i) returns the least position j such that there are i ones in the range $s[0] \ldots s[j]$. `bitRank`(s, j) returns the number of ones in the range $s[0] \ldots s[j]$. These operations mimic the function of the `rank` and `select` data structures of Jacobson [19].

If the processor does not support these operations, we can implement them using table lookup in $1/\epsilon$ time using $O(2^{\epsilon w}\epsilon w \log(\epsilon w))$ bits. By simulating a word size of $\Theta(\log |C|)$ this can be reduced to less than $|C|$, and thus made a low order term, while running in constant time. Note that it is always possible to simulate smaller words with larger words with constant overhead by

[1] By d simplicial mesh we mean a pure simplicial complex of dimension d, which is a manifold, possibly with boundary [13].

packing multiple small words into a larger one.

Memory allocation. Many of our structures do not explicitly support storage of bit strings longer than w bits. To handle these strings we use a separate memory allocation system. This memory system must be capable of allocating or freeing $|s|$ bits of memory in time $|s|/w$, and may use $O(|s|)$ space to keep track of each allocation. It is well known how to do this (*e.g.*, [1]).

Quotienting. For sets of fixed length elements a space bound is already known [24]: to represent n elements, each of size $|s|$ bits, requires $O(n(|s| - \log n))$ bits. A method used to achieve this bound is *quotienting*: every element $s \in U$ is uniquely hashed into two bit strings s', s'' such that s' is a $\log n$-bit index into a hash bucket and s'' contains $|s| - \log n$ bits. Together, s' and s'' contain enough bits to describe s; however to add s to the data structure, it is only necessary to store s'' in the bucket specified by s'. The idea of quotienting was first described by Knuth [21, Section 6.4, exercise 13] and has been used in several contexts [11, 8, 28, 15].

Gamma codes. The gamma code [14] is a variable-length prefix code that represents a positive integer v with $\lfloor \log v \rfloor$ zeroes, followed by the $(\lfloor \log v \rfloor + 1)$-bit binary representation of v, for a total of $2\lfloor \log v \rfloor + 1$ bits.

Given a string s containing a gamma code (of length $\leq w$) followed possibly by other information, it is possible to decode the gamma code in constant time. First, an algorithm uses $\texttt{bitSelect}(s, 1)$ to find the location j of the first one in s. The length of the gamma code is $2j + 1$, so the algorithm uses shifts to extract the first $2j + 1$ bits of s. A gamma code for d is equivalent to a binary code for d with some leading zeroes; thus decoding d is equivalent to reinterpreting it as an integer.

If the integer d to be encoded might be zero or negative, this can be handled by packing a sign bit with the gamma code for d. If the sign bit is a zero, then the gamma code is a code for d; otherwise, the gamma code is actually a code for $1 - d$.

Gamma codes are only one of a wide class of variable-length codes. This paper makes use of gamma codes because they require very few operations to decode.

3 Arrays

The variable-bit-string array problem is to maintain bit strings $a_1 \ldots a_n$, supporting `update` and `lookup` operations. Our array representation supports strings of size $1 \leq |a_i| \leq w$. Strings of size more than w must be allocated separately, and w-bit pointers to them can be stored in our structure.

Our structure consists of two parts: a set of blocks B and an index I. The bit-strings in the array are stored in the blocks. The index allows us to quickly locate the block containing a given array element.

Blocks. A *block* B_i is an encoding of a series of bit strings (in increasing order) a_i, a_{i+1}, $\ldots$, a_{i+k}. The block stores the concatenation of the strings $b_i = a_i a_{i+1} \ldots a_{i+k}$, together with information from which the start location of each string can be found. It suffices to store a second bit string b'_i such that b'_i contains a 1 at position j if and only if some bit string a_k ends at position j in b_i.

A block B_i consists of the pair (b_i, b'_i). We denote the size of a block by $|b_i| = \sum_{j=0}^{k} |a_{i+j}|$. We maintain the strings of our array in blocks of size at most w. We maintain the invariant that, if two blocks in our structure are adjacent (meaning, for some i, one block contains a_i and the other contains a_{i+1}), then the sum of their sizes is greater than w.

Index structure. The index I for our array structure consists of a bit array $A[1 \ldots n]$ and a hash table H. The array A is maintained such that $A[i] = 1$ if and only if the string a_i is the first string in some block B_i in our structure. In that case, the hashtable H maps i to B_i.

The hashtable H must use $O(w)$ bits (that is, $O(1)$ words) per block maintained in the hashtable. It must support insertion and deletion in expected amortized $O(1)$ time, and lookup in worst-case $O(1)$ time. Cuckoo hashing [26] or the dynamic version of the FKS perfect hashing scheme [12] have these properties. If expected rather than worst-case lookup bounds are acceptable, then a standard implementation of chained hashing will work as well.

Operations. We begin by observing that no block can contain more than w bit strings (since blocks have maximum size w and each bit string has size at least one bit). Thus, from any position $A[k]$, the distance to the nearest one in either direction is at most w. To find the nearest one on the left, we let $s = A[k - w] \ldots A[k - 1]$ and compute $\texttt{bitSelect}(s, \texttt{bitRank}(s, w - 1))$. To find the nearest one on the right, we let $s = A[k+1] \ldots A[k+w]$ and compute $\texttt{bitSelect}(s, 1)$. These operations take constant time.

To access a string a_k, our structure first searches I for the block B_i containing a_k. This is simply a search on A for the nearest one on the left. The structure performs a hashtable lookup to access the target block B_i. Once the block is located, the structure scans the index string b'_i to find the location of a_k. This can be done using $\texttt{bitSelect}(b'_i, k - i + 1)$.

If a_k is updated, its block B_i is rewritten. If B_i

becomes smaller as a result of an update, it may need to be merged with its left neighbor or its right neighbor (or both). In either case this takes constant time.

If B_i becomes too large as a result of an update to a_k, it is split into at most three blocks. The structure may create a new block at position k, at position $k+1$, or (if the new $|a_k|$ is large) both. To maintain the size invariant, it may then be necessary to join B_i with the block on its left, or to join the rightmost new block with the block on its right.

All of the operations on blocks and on A take $O(1)$ time: shifting and copying takes can be done w bits at a time. Access operations on H take $O(1)$ worst-case time; updates take $O(1)$ expected amortized time.

We define the total length of the bit-strings in the structure to be $m = O(\sum_{i=1}^{n} |a_i|)$. The structure contains n bits in A plus $O(w)$ bits per block; there are $O(m/w+1)$ blocks, so the total space usage is $O(m+w)$. This gives us the following theorem:

Theorem 3.1 *Our variable-bit-string array representation can store bit strings of length $1 \le a_i \le w$ in $O(w + \sum_{i=1}^{n} |a_i|)$ bits while allowing accesses in $O(1)$ worst-case time and updates in $O(1)$ amortized expected time.*

The proof follows from the discussion above.

4 Dictionaries

Using our variable-bit-length array structure we can implement space-efficient variable-bit-length dictionaries. In this section we describe dictionary structures that can store a set of bit strings $s_1 \ldots s_n$, for $1 \le |s_i| \le w+\log n$. (We can handle strings of length greater than $w + \log n$ by allocating memory separately and storing a w-bit pointer in our structure.) Our structures use space $O(m)$ where $m = \sum(\max(|s_i| - \log n, 1) + |t_i|)$.

We will first discuss a straightforward implementation based on chained hashing that permits $O(1)$ expected query time and $O(1)$ expected amortized update time. We will then present an implementation based on the dynamic version [12] of the FKS perfect hashing scheme [16] that improves the query time to $O(1)$ worst-case time. Our structure uses quotienting, as described in Section 2.

For our quotienting scheme to work, we will need the number of hash buckets to be a power of two. We will let q be the number of bits quotiented, and assume there are 2^q hash buckets in the structure. As the number of entries grows or shrinks, we will resize the structure using a standard doubling or halving scheme so that $2^q \approx n$.

Hashing. For purposes of hashing it will be convenient to treat the bit strings s_i as integers. Accordingly we reinterpret, when necessary, each bit string as the binary representation of a number. To distinguish strings with different lengths we prepend a 1 to each s_i before interpreting it as a number. We denote this padded numerical representation of s_i by x_i.

We say a family H of hash functions onto 2^q elements is *k-universal* if for random $h \in H$, $\Pr(h(x_1) = h(x_2)) \le k/2^q$ [9], and is *k-pairwise independent* if for random $h \in H$, $\Pr(h(x_1) = y_1 \wedge h(x_2) = y_2) \le k/2^{2q}$ for any $x_1 \ne x_2$ in the domain, and y_1, y_2 in the range.

We wish to construct hash functions h', h''. The function h' must be a hash function $h' : \{0,1\}^{w+q+1} \to \{0,1\}^q$. The binary representation of $h''(x_i)$ must contain q fewer bits than the binary representation of x_i. Finally, it must be possible to reconstruct x_i given $h'(x_i)$ and $h''(x_i)$.

Note that others, such as [21, 24, 27], have described quotienting functions in the past. Previous authors, however, were not concerned with variable length keys, so their h'' functions do not have the length properties we need.

For clarity we break x_i into two words, one containing the low-order q bits of x_i, the other containing the remaining high-order bits. The hash functions we use are:

$$\overline{x}_i = x_i \operatorname{div} 2^q \qquad \underline{x}_i = x_i \bmod 2^q$$
$$h''(x_i) = \overline{x}_i \qquad h'(x_i) = (h_0(\overline{x}_i)) \oplus \underline{x}_i$$

where h_0 is any 2-pairwise independent hash function with range 2^q. For example, we can use:

$$h_0(x_i) = ((ax_i + b) \bmod p) \bmod 2^q$$

where $p > 2^q$ is prime and a, b are randomly chosen from $1 \ldots p$. Given h' and h'', these functions can be inverted in a straightforward manner:

$$\overline{x}_i = h'' \qquad \underline{x}_i = h_0(h'') \oplus h'$$

We can show that the family from which h' are drawn is 2-universal as follows. Given $x_1 \ne x_2$, we have

$$\begin{aligned} \Pr(h'(x_1) = h'(x_2)) &= \Pr(h_0(\overline{x}_1) \oplus \underline{x}_1 = h_0(\overline{x}_2) \oplus \underline{x}_2) \\ &= \Pr(h_0(\overline{x}_1) \oplus h_0(\overline{x}_2) = \underline{x}_1 \oplus \underline{x}_2) \end{aligned}$$

The probability is zero if $\overline{x}_1 = \overline{x}_2$, and otherwise it is $< 2/2^{2q}$ (by the 2-pairwise independence of h_0). Thus $\Pr(h'(x_1) = h'(x_2)) \le 2/2^{2q}$.

Note also that selecting a function from H requires $O(\log n)$ random bits.

Dictionaries. Our dictionary data structure is a hash table consisting of a variable-bit-length array A and a hash function h', h''. To insert (s_i, t_i) into the structure, we compute s'_i and s''_i and insert s''_i and t_i into bucket s'_i.

It is necessary to handle the possibility that multiple strings hash to the same bucket. To handle this we prepend to each string s''_i or t_i a gamma code indicating its length. (This increases the length of the strings by at most a constant factor.) We concatenate together all the strings in a bucket and store the result in the appropriate array slot.

If the concatenation of all the strings in a bucket is of size greater than w, we allocate that memory separately and store a w-bit pointer in the array slot instead.

It takes $O(1)$ time to decode any element in the bucket (since the gamma code for the length of an element can be read in constant time with a `bitSelect` function and shifts). Each bucket has expected size $O(1)$ elements (since our hash function is universal), so lookups for any element can be accomplished in expected $O(1)$ time, and insertions and deletions can be accomplished in expected amortized $O(1)$ time.

The bit string stored for each s_i has size $O(\max(|s_i| - q, 1))$; the bit string for t_i has size $O(|t_i|)$. Our variable-bit-length array increases the space by at most a constant factor, so the total space used by our variable dictionary structure is $O(m)$ for $m = \sum(\max(|s_i| - \log n, 1)) + |t_i|)$.

Perfect Hashing. We can also use our variable-bit-length arrays to implement a dynamized version of the FKS perfect hashing scheme. We use the same hash functions h', h'' as above, except that h' maps to $\{0,1\}^{\log n+1}$ rather than $\{0,1\}^{\log n}$. We maintain a variable-bit-length array of $2n$ buckets, and as before we store each pair (s''_i, t_i) in the bucket indicated by s'_i.

If multiple strings collide within a bucket, and their total length is w bits or less, then we store the concatenation of the strings in the bucket, as we did with chained hashing above. However, if the length is greater than w bits, we allocate a separate variable-bit-length array to store the elements. If the bucket contained k bits then the new array has about k^2 slots—we maintain the size and hash function of that array as described by Dietzfelbinger et. al. [12].

In the primary array we store a w-bit pointer to the secondary array for that bucket. We charge the cost of this pointer, and the $O(w)$-bit overhead for the array and hash function, to the cost of the w bits that were stored in that bucket. The space bounds for our structure follow from the bounds proved in [12]: the structure allocates only $O(n)$ array slots, and our structure requires only $O(1)$ bits per unused slot. Thus the space requirement of our structure is dominated by the $O(m)$ bits required to store the elements of the set.

Access to elements stored in secondary arrays takes worst-case constant time. Access to elements stored in the primary array is more problematic, as the potentially w bits stored in a bucket might contain $O(w)$ strings, and to meet a worst-case bound it is necessary to find the correct string in constant time.

We can solve this problem using table lookup. The table needed would range over $\{0,1\}^{\epsilon w} * \{0,1\}^{\epsilon w}$; and would allow searching in a string a of gamma codes for a target string b. Each entry would contain the index in a of b, or the index of the last gamma code in a if b was not present. The total space used would be $2^{2\epsilon w}\log(\epsilon w)$; the time needed for a query would be $O(1/\epsilon)$.

By selecting ϵ and w appropriately we can made the table require $o(|C|)$ space.

This gives us the following theorem:

Theorem 4.1 *Our variable-bit-string dictionary representation can store bit strings of any size using $O(m)$ where $m = \sum(\max(|s_i| - \log n, 1) + t_i)$ bits while allowing updates in $O(1)$ amortized expected time and accesses in $O(1)$ worst-case time.*

5 Graphs

Using our variable-bit-length dictionary structure we can implement space-efficient representations of unlabeled graphs. We will begin by describing a general data structure for representing integer labeled n-vertex graphs. We will then describe how this structure can be efficiently compressed by assigning labels appropriately.

Operations. We wish to support the following operations:

ADJACENT(u, v)**:** true iff u and v are adjacent in G

FIRSTEDGE(v)**:** return the first neighbor of v in G

NEXTEDGE(u, v)**:** given a vertex u and neighbor v in G, return the next neighbor of u

ADDEDGE(u, v)**:** add the edge (u, v) to G

DELETEEDGE(u, v)**:** delete the edge (u, v) from G.

The query operations will take $O(1)$ worst-case time, while the update operations (**addEdge** and **deleteEdge**) will take $O(1)$ amortized expected time.

The **adjacent** operation allows us to support adjacency queries, while the combination of **firstNeighbor** and **nextNeighbor** allow us to support neighbor listing in $O(1)$ time per neighbor. The interface can be

supported by using doubly linked adjacency lists along with a hash table using $O((|E|+|V|)\log|V|)$ bits. The hash table can be used for the adjacency and **deleteEdge** operations. We would like to improve on the space bounds.

Our structure can represent any graph but it will give good compression results only on a certain class of graphs: those with *k-compact labelings*. Given an integer labeling for the vertices of a graph, we define the length of an edge $|e|, e = (u, v)$ to be the distance between its vertices $|u - v|$. We say that a k-compact labeling for a graph is one for which $\sum_{e\in E} \log|e| < k|V|$. We define a graph to be *k-compact* if it has a k-compact labeling.

Blandford et. al. showed that for any class of graphs satisfying an $O(n^{1-\epsilon})$-edge separator theorem, $\epsilon > 0$, all members are $O(1)$-compact [4]. This includes bounded-degree planar graphs, which satisfy an $O(n^{\frac{1}{2}})$-edge separator theorem, and certain well-shaped meshes [22] of fixed dimension. The labeling can be found using separator trees. Additionally, many graphs in practice have been found to be k-compact for much smaller k than would be expected for random graphs [5] (*e.g.*, web link graphs, VLSI circuits, and internet connectivity graphs).

Theorem 5.1 *All n-vertex graphs with a k-compact labeling can be stored in $O(k|V|)$ bits while allowing updates in $O(1)$ amortized expected time and queries in $O(1)$ worst-case time.*

Proof. We begin by describing our graph structure in an uncompressed form, and then describe how it is compressed.

Our structure represents a graph as a dictionary of edges. The edges incident on each vertex are cross linked into a doubly linked list. Consider a vertex u and some ordering on its neighboring vertices $v_1, \ldots, v_d$. We represent each edge $(u, v_i), 1 \le i \le d$ using the dictionary entry $(u, v_i; v_{i-1}, v_{i+1})$. (That is, (u, v_i) is the key, and (v_{i-1}, v_{i+1}) is the associated data.) We define $v_0, v_{d+1} = u$ and for each vertex we include an entry $(u, u; v_d, v_1)$.

Given this representation we can support all of the above operations using functions of the dictionary. Pseudocode for these operations is shown in Figure 1.

In its uncompressed form this dictionary consumes $d + 1$ entries for each vertex of degree d. The total number of entries is therefore $|V| + |E|$. The space used is $O((|E| + |V|)w)$.

Compression. To compress this structure we make use of *difference coding*: we simply store each dictionary entry using differences with respect to u. That is to say, rather than store an entry $(u, v_i; v_{i-1}, v_{i+1})$ in the dictionary, we instead store $(u, v_i - u; v_{i-1} - u, v_{i+1} - u)$.

We use our variable-bit-length dictionary to store the entries. The encoding of u in each entry requires $\log|V|$ bits; the dictionary absorbs this cost using quotienting. The space used, then, is proportional to the cost of encoding $v_i - u$, $v_{i-1} - u$, and $v_{i+1} - u$, for each edge (u, v_i) in the dictionary. We compress these differences by representing them with gamma codes (with sign bits). The cost to encode each edge e with a logarithmic code is $O(\log|e|)$. Each edge appears $O(1)$ times in the structure, so the total cost to encode all the edges is $\sum_{e\in E} \log|e|$.

For a k-compact labeling, $\sum_{e\in E} \log|e|$ is $O(kn)$.

6 Ordered Sets

We would like to represent ordered sets S of integers in the range $(0, \ldots, m - 1)$. In addition to lookup operations, an ordered set needs to efficiently support queries that depend on the order. Here we consider findNext and finger searching. *findNext* on a key k_1 finds $\min\{k_2 \in S | k_2 > k_1\}$; *finger searching* on a finger key $k_1 \in S$ and a key k_2 finds $\min\{k_3 \in S | k_3 > k_2\}$, and returns a finger to k_3. Finger searching takes $O(\log l)$ time, where $l = |\{k \in S | k_1 \le k \le k_2\}|$.

To represent the set we use a red-black tree on the elements. We will refer to vertices of the tree by the value of the element stored at the vertex, use n to refer to the size of the set, and without loss of generality we assume $n < m/2$. For each element v we denote the parent, left-child, right child, and red-black flag as $p(v)$, $l(v)$, $r(v)$, and $q(v)$ respectively.

We represent the tree as a dictionary containing entries of the form $(v; l(v) - v, r(v) - v, q(v))$. (We could also add parent pointers $p(v) - v$ without violating the space bound, but in this case they are unnecessary.) It is straightforward to traverse the tree from top to bottom in the standard way. It is also straightforward to implement a rotation by inserting and deleting a constant number of dictionary elements. Assuming dictionary queries take $O(1)$ time, *findNext* can be implemented in $O(\log n)$ time. Using a hand data structure [7], finger searching can be implemented in $O(\log l)$ time with an additional $O(\log^2 n)$ space. Membership takes $O(1)$ time. Insertion and deletion take $O(\log n)$ expected amortized time. We call this data structure a *dictionary red-black tree*.

It remains to show the space bound for the structure.

Lemma 6.1 *If a set of integers $S \subset \{0, \ldots, m-1\}$ of size n is arranged in-order in a red-black tree T then $\sum_{v\in T}(\log|p(v) - v|) \in O(n \log(m/n))$.*

```
ADJACENT(u, v)
    return (LOOKUP((u, v)) ≠ null)

FIRSTEDGE(u)
    (v_p, v_n) ←LOOKUP((u, u))
    return v_n

NEXTEDGE(u, v)
    (v_p, v_n) ←LOOKUP((u, v))
    return v_n

ADDEDGE(u, v)
    (v_p, v_n) ←LOOKUP((u, u))
    INSERT((u, u), (v_p, v))
    INSERT((u, v), (u, v_n))

DELETEEDGE(u, v)
    (v_p, v_n) ←LOOKUP((u, v))
    (v_pp, v) ←LOOKUP((u, v_p))
    (v, v_nn) ←LOOKUP((u, v_n))
    INSERT((u, v_p), (v_pp, v_n))
    INSERT((u, v_n), (v_p, v_nn))
    DELETE((u, v))
```

Figure 1: Pseudocode to support our graph operations.

Proof. Consider the elements of a set $S \subset \{0, \ldots, m-1\}$ organized in a set of levels $L(S) = \{L_1, \ldots, L_l\}$, $L_i \subset S$. If $|L_i| \leq \alpha |L_{i+1}|, 1 \leq i < l, \alpha > 1$, we say such an organization is a *proper level covering* of the set.

We first consider the sum of the log-differences of cross pointers within each level, and then count the pointers in the red-black trees against these pointers. For any set $S \subset \{0, \ldots, m-1\}$ we define $next(e, S) = \min\{e' \in S \cup \{m\} | e' > e\}$, and $M(S) = \sum_{j \in S} \log(next(j, S) - j)$. Since logarithms are concave the sum is maximized when the elements are evenly spaced and hence $M(S) \leq |S| \log(m/|S|))$. For any proper level covering L of a set S this gives:

$$\begin{aligned}
\sum_{L_i \in L(S)} M(L_i) &\leq \sum_{L_i \in L} |L_i| \log(m/|L_i|) \\
&\leq \sum_{i=0}^{i<l} \alpha^{-i} |S| \log(\alpha^i m/|S|)) \\
&\leq 2 + \frac{\alpha}{(\alpha - 1)} |S| \log(m/|S|) \\
&\in O(|S| \log(m/|S|))
\end{aligned}$$

This represents the total log-difference when summed across all "next" pointers. The same analysis bounds similarly defined "previous" pointers. Together we call these *cross pointers*.

We now account for each pointer in the red-black tree against one of the cross pointers. First partition the red-black tree into levels based on number of black nodes in the path from the root to the node. This gives a proper level covering with $\alpha = 2$. Now for each node i, the distance to each of its two children is at most the distance to the previous or next element in its level. Therefore we can account for the cost of the left child against the previous pointer and the right child against next pointer. The sum of the log-differences of the child pointers is therefore at most the sum of the log-differences of the next and previous cross pointers. This gives the desired bound.

Theorem 6.1 *A set of integers $S \subset \{0, \ldots, m-1\}$ of size n represented as a dictionary red-black tree and using a compressed dictionary uses $O(n \log((n+m)/n)$ bits and supports find-next queries in $O(\log n)$ time, finger-search queries in $O(\log l)$ time, and insertion and deletion in $O(\log n)$ expected amortized time.*

Proof. (outline) Recall that the space for a compressed dictionary is bounded by $O(m)$ where $m = \sum_{(s,t) \in D} (\max(1, |s| - \log |D|) + |t|)$. The keys use $\log |D|$ bits each, and the size of the data stored in the dictionary is bounded by Lemma 6.1. This gives the desired bounds.

7 Cardinal Trees

A cardinal tree (aka trie) is a rooted tree in which every node has c slots for children any of which can be filled. We generalize the standard definition of cardinal trees to allow each node v to have a different c, denoted as $c(v)$. For a node v we want to support returning the parent $p(v)$ and the i^{th} child $v[i]$, if any. We also want to support deleting or inserting a leaf node. As with graphs, we consider these semi dynamic operations since the updates might require relabeling of the vertices to maintain the space bounds.

Lemma 7.1 *Integer labeled cardinal trees with $m = \sum_{v \in V} (\log c(p(v)) + \log |v - p(v)|)$, can be stored in $O(m)$ bits and support parent and child queries in $O(1)$ time and insertion and deletion of leaves in $O(1)$ expected amortized time.*

Proof. (outline) For child queries we can just store a dictionary entry for each vertex v that is keyed on $(p(v), i)$ and stores $p(v) - v$ as the data. In the cost for dictionaries given by $m = \sum_{(s,t)\in D}(|t| + \max(1, |s| - \log|D|))$ the $p(v)$ can be counted against the $\log|D|$, the i against the $\log c(p(v))$ and the $p(v) - v$ against the $\log|v - p(v)|$. Parent queries can be supported by a dictionary from v to $p(v)$.

Any tree T can be separated into a set of trees of size at most $1/2n$ by removing a single node. Recursively applying such a separator on the cardinal tree defines a separator tree T_s over the nodes. An integer labeling can then be given to the nodes of T based on the inorder traveral of T_s. We call such a labeling a *tree-separator labeling*.

Lemma 7.2 *For all tree-separator labelings of trees $T = (V, E)$ of size n, $\sum_{(u,v)\in E}(\log|u - v|) < O(n) + 2\sum_{(u,v)\in E}\log(\max(d(u), d(v)))$.*

Proof. Consider the separator tree $T_s = (V, E_s)$ on which the labeling is based. For each node v we denote the degree of v by $d(v)$. We let $T_s(v)$ denote the subtree of T_s that is rooted at v. Thus $|T_s(v)|$ is the size of the piece of T for which v was chosen as a separator.

There is a one-to-one correspondence between the edges E and edges E_s. In particular consider an edge $(v, v') \in E_s$ between a vertex v and a child v'. This corresponds to an edge $(v, v'') \in T$, such that $v'' \in T_s(v')$. We need to account for the log-difference $\log|v - v''|$. We have $|v - v''| < |T_s(v)|$ since all labels in any subtree are given sequentially. We partition the edges into two classes and calculate the cost for edges in each class.

First, if $d(v) > \sqrt{|T_s(v)|}$ we have for each edge (v, v''), $\log|v - v''| < \log|T_s(v)| < 2\log d(v) < 2\log\max(d(v), d(v''))$.

Second, if $d(v) \leq \sqrt{|T_s(v)|}$ we charge each edge (v, v'') to the node v. The most that can be charged to a node is $\sqrt{|T_s(v)|}\log|T_s(v)|$ (one pointer to each child). Note that for any tree in which for every node v (A) $|T_s(v)| < 1/2|T_s(p(v))|$, and (B) $\text{cost}(v) \in O(|T_s(v)|^c)$ for some $c < 1$, we have $\sum_{v\in V}\text{cost}(v) \in O(n)$. Therefore the total charge is $O(n)$.

Summing the two classes of edges gives $O(|T|) + 2\sum_{(u,v)\in E}\log(\max(d(u), d(v)))$.

Theorem 7.1 *Cardinal trees with a tree-separator labeling, with $m = \sum_{v\in V}(1 + \log(1 + c(p(v))))$ can be stored in $O(m)$ bits.*

Proof. We are interested in the edge cost $E_c(T) = \sum_{v\in V}(\log|v - p(v)|)$. Substituting $p(v)$ for u in Lemma 7.2 gives:

$$\begin{aligned} E_c(T) &< O(n) + 2\sum_{v\in V}\log(\max(d(v), d(p(v)))) \\ &< O(n) + 2\sum_{v\in V} d(v) + \log d(p(v))) \\ &= O(n) + 4n + 2\sum_{v\in V}\log(d(p(v))) \\ &< O(n) + 2\sum_{v\in V}\log(1 + c(p(v))) \end{aligned}$$

With Lemma 7.1 this gives the required bounds.

8 Simplicial Meshes

Using our variable-bit-length dictionary structure we can implement space-efficient representations of d dimensional simplicial meshes. By a d simplicial mesh we mean a pure simplicial complex of dimension d, which is a manifold, possibly with boundary [13].

We will describe the structure for $d = 3$ but note that this can be generalized to d dimensions. Our structure supports the following operations:

`find`(a, b, c)**:** finds all vertices d such that (a, b, c, d) form a tetrahedron in M (at most two since the mesh is a manifold).

`insert`(a, b, c, d)**:** adds the tetrahedron (a, b, c, d) to M

`delete`(a, b, c, d)**:** delete the tetrahedron (a, b, c, d) from M

We represent a simplicial mesh as a dictionary of simplices. Each face (a, b, c) in the mesh may belong to two tetrahedra, (a, b, c, d) and (a, b, c, e). For each face in the mesh we store the entry $(a, b, c; d, e)$. If a face is not part of two tetrahedra then we store the special character 0 in that slot. The operations can then be implemented as shown in Figure 2.

As we did with our graph implementation, we can compress this structure by encoding b, c, d, and e relative to a. That is, in our variable-bit-length dictionary we store tuples of the form $(a, b-a, c-a; d-a, e-a)$. To account for the space stored, we charge the cost of storing $b - a$ and $c - a$ to the face (a, b, c); we charge the cost of $d - a$ to the face (a, b, d) and of $e - a$ to the face (a, b, e). The dictionary absorbs the $\log|V|$-bit cost of representing a using quotienting.

Each face is charged at most $O(1)$ times, and each time the charge is $O(\log|b-a| + \log|c-a|)$. This gives:

Theorem 8.1 *Our simplicial mesh representation using our variable-bit dictionary uses*

```
FIND((a, b, c))
    return LOOKUP((a, b, c), T)

INSERT(S)
    for each rotation (a, b, c, d) of S
        (e, 0) ← LOOKUP((a, b, c), T)
        INSERT((a, b, c), (d, e))
DELETE(S)
    for each rotation (a, b, c, d) of S
        (d, e) ← LOOKUP((a, b, c), T)
        if e = 0 then DELETE((a, b, c), T)
        else INSERT((a, b, c), (e, 0))
```

Figure 2: Pseudocode to support simplicial mesh operations.

$O(\sum_{(a,b,c)\in F}(\log|a-b|+\log|a-c|))$ bits where F is the $2-$skeleton of the mesh. It supports `find` *in worst-case constant time and* `add` *and* `delete` *in expected amortized constant time.*

If the 2-skeleton of the mesh (that is, the graph induced by the faces) has a k-compact labeling, then the representation of the mesh will use $O(n)$ bits. We note that well-shaped meshes with bounded degree have small separators [22] and are therefore k-compact for fixed dimension.

If the mesh has bounded degree, then the bound $O(\sum_{(a,b,c)\in F}(\log|a-b|+\log|a-c|))$ simplifies to $O(\sum_{(u,v)\in E}(\log|u-v|))$, where E is the $1-$skeleton of the mesh.

References

[1] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[2] D. Benoit, E. D. Demaine, J. I. Munro, and V. Raman. Representing trees of higher degree. In *WADS*, pages 169–180, 1999.

[3] D. Blandford and G. Blelloch. Compact representations of ordered sets. In *SODA*, pages 11–19, 2004.

[4] D. Blandford, G. Blelloch, and I. Kash. Compact representations of separable graphs. In *SODA*, pages 342–351, 2003.

[5] D. Blandford, G. Blelloch, and I. Kash. An experimental analysis of a compact graph representation. In *ALENEX04*, 2004.

[6] D. K. Blandford, G. E. Blelloch, D. E. Cardoze, and C. Kadow. Compact representations of simplicial meshes in two and three dimensions. In *Proc. International Meshing Roundtable (IMR)*, Sept. 2003.

[7] G. E. Blelloch, B. Maggs, and M. Woo. Space-efficient finger search on degree-balanced search trees. In *SODA*, pages 374–383, 2003.

[8] A. Brodnick and J. I. Munro. Membership in constant time and almost-minimum space. *Siam Journal of Computing*, 28(5):1627–1640, 1999.

[9] L. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, pages 143–154, 1979.

[10] R. C.-N. Chuang, A. Garg, X. He, M.-Y. Kao, and H.-I. Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. *Lecture Notes in Computer Science*, 1443:118–129, 1998.

[11] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Comput*, 9:828–834, 1984.

[12] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[13] H. Edelsbrunner. *Geometry and Topology of Mesh Generation*. Cambridge Univ. Press, England, 2001.

[14] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, March 1975.

[15] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. Space efficient hash tables with worst case constant access time. In *STACS*, 2003.

[16] M. L. Fredman, J. Komlos, and E. Szemerdi. Storing a sparse table with 0(1) worst case access time. *JACM*, 31(3):538–544, 1984.

[17] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *FOCS*, pages 397–406, 2000.

[18] X. He, M.-Y. Kao, and H.-I. Lu. A fast general methodology for information-theoretically optimal encodings of graphs. *SIAM J. Computing*, 30(3):838–846, 2000.

[19] G. Jacobson. Space-efficient static trees and graphs. In *30th FOCS*, pages 549–554, 1989.

[20] K. Keeler and J. Westbrook. Short encodings of planar graphs and maps. *Discrete Applied Mathematics*, 58:239–252, 1995.

[21] D. E. Knuth. *The Art of Computer Programming/Sorting and Searching, Volumes 3*. Addison Wesley, 1973.

[22] G. L. Miller, S.-H. Teng, W. P. Thurston, and S. A. Vavasis. Separators for sphere-packings and nearest neighbor graphs. *Journal of the ACM*, 44:1–29, 1997.

[23] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th FOCS*, pages 118–126, 1997.

[24] R. Pagh. Low redundancy in static dictionaries with o(1) worst case lookup. In *ICALP*, pages 595–604, 1999.

[25] R. Pagh. Low redundancy in static dictionaries with constant query time. *Siam Journal of Computing*, 31(2):353–363, 2001.

[26] R. Pagh and F. F. Rodler. Cuckoo hashing. In *ESA*,

2001.

[27] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, 2002.

[28] R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *ICALP*, pages 357–36, 2003.

[29] G. Turán. Succinct representations of graphs. *Discrete Applied Mathematics*, 8:289–294, 1984.