COMPACT DATA STRUCTURES WITH FAST QUERIES

Daniel K. Blandford
dkb1@cs.cmu.edu

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

I consider the problem of compactly representing data structures which still allow queries to be made of them. Such compact representations can be very useful when dealing with large datasets: they can allow a representation to fit in main memory when it would otherwise need to be swapped out to disk. These representations can improve cache performance as well since they allow more data to fit into the cache.

Compact representations are useful for many real-world applications. Search engines, in particular, may index billions of web pages and must answer queries about those pages in real-time. For such applications it is vitally important to decrease the space used by their index while still allowing efficient queries. Compact representations are also useful for large-scale graphs and meshes.

In my thesis I will describe compact representations of graphs and meshes. I describe a compact index which is suitable for application to the search engine problem; additionally, I show how compact ordered sets can be used to solve that problem, sacrificing some compression for a wider range of operations. Further, I describe generic compressed array and dictionary structures which can contain data elements of variable size (while using space proportional to the sum of the sizes of the elements).

For each representation I present experimental results demonstrating the efficiency of that representation in both space usage and query response time.

1 Introduction

In my thesis I will consider the problem of compactly representing data structures which still allow queries to be made of them. Such compact representations can be very useful when dealing with large datasets: they can allow a representation to fit in main memory when it would otherwise need to be swapped out to disk. These representations can improve cache performance as well since they allow more data to fit into the cache.

The structures I describe make use of difference coding (see section 7) to represent data by its difference from other, previously known, data. For example, I represent the neighbors of a vertex by the difference between the neighbor and the original vertex. For many structures I combine this with a relabeling scheme which ensures that most of the differences I encode are small.

I will describe compact representations of several types of data structures including separable graphs, meshes, text indices, ordered sets, and variable-length arrays. Each data structure will be accompanied by experimentation describing its practical performance.

Separable Graphs. I say that a graph is separable if it and all its subgraphs can be partitioned into two approximately equally sized parts by removing a relatively small number of vertices. (The expected separator size of a random graph is $\Omega(m - n)$.)

In my thesis I will present a data structure for compressing separable graphs to $O(n)$ bits while still permitting adjacency queries in $O(1)$ time and listing neighbors in $O(1)$ time per neighbor. I will present experimental results from compressing several types of real-world graphs. All the graphs tested compress well using this data structure, indicating that real-world graphs do in fact have small separators.

Mesbes. Many applications involve representing very large meshes. For such applications the limiting factor can be the space required to represent the mesh in memory. Standard representations of tetrahedral meshes, for example, can require 300-500 bytes per vertex. In my thesis I will describe a compact representation for 2-
and 3-dimensional meshes which uses an average of 30 bytes per vertex in 2D or 80 bytes per vertex in 3D. This representation supports dynamic queries with speed that is close to that of non-compressed representations.

**Indices.** Memory considerations are a serious concern in the design of search engines. Most of the space used by a search engine is in the representation of an inverted index, a data structure that maps search terms to lists of documents containing those terms. Each entry (or posting list) in an inverted index is a list of the document numbers of documents containing a specific term. When a query on multiple terms is entered, the search engine retrieves the corresponding posting lists from memory, performs some set operations to combine them into a result, sorts the resulting hits based on some priority measure, and reports them to the user.

Posting lists are generally kept compressed using difference coding. Difference coding produces the best compression when the data to be compressed has high locality: when the numbers to be stored in the lists are clustered rather than randomly distributed over the interval \( \{0, \ldots, n\} \). In my thesis I will describe an algorithm for improving the locality of posting lists by renumbering the documents. This improves the available compression by about 14%.

**Ordered Sets.** For many applications it may be useful to represent an ordered set, that is, a set of integers from some fixed range. In particular, the posting list data structure used by an inverted index is represented as an ordered set of document numbers. In my thesis I will describe a representation of ordered sets which supports a wide range of operations while maintaining the data in a compressed form.

More specifically, I describe a technique for modifying existing ordered-set data structures (such as balanced trees) to maintain the data in compressed form while still supporting all operations in the same time bounds. By applying this technique to a functional implementation of treaps I produce a compressed data structure that supports rapid set union and intersection operations. These operations are very useful for posting lists.

**Arrays and Dictionaries.** A common thread in the data structures I have discussed is the need to store variable-length bit strings. For graphs and meshes, the amount of data stored for each vertex is variable; for ordered sets, the size of the difference between each pair of elements is variable. In my thesis I will describe a generic approach to storing variable length bit strings in “arrays” and dictionaries. The structures are optimal in space usage and support a wide range of operations.

Using these variable-bit-length data structures it is possible to implement a wide variety of compressed data structures with fast queries. I will describe compressed representations of graphs, \(d\)-simplicial meshes, ordered sets, and cardinal trees.

I present theoretical results for most of the structures I describe, and I will present experiemiental results for all of them. This is an important property of my thesis: many previous theoretically-optimal structures have been too complex to be useful in practice. My structures are useful in both theory and practice.

## 2 Separable Graphs

Many applications involve representing very large graphs. In such applications space can be as much of an issue as time. For example, there has been recent interest in efficiently representing the link structure of the web for use in various algorithms [10, 1, 33]. This graph consists of over a billion vertices and ten billion edges. Other applications of large graphs include representing the meshes used in finite-element simulations, or 3d models in graphics. For such applications it is useful to be able to represent large graphs compactly (with as few bits as possible), while still supporting dynamic queries efficiently.

In regards to compactness, the information-theoretic lower bound for representing a random graph with \(n\) vertices and \(m\) edges is \(\Theta(n \log \frac{m}{n})\) bits. This bound can be matched by using adjacency lists with difference encoding [39]. Although such a representation is likely to reduce the number of bits used compared to a naive representation, considerably more savings can be achieved by taking advantage of the fact that graphs in practice are typically not random, but have significant structure.

There has been considerable previous work on compressing planar graphs. Turan [36] first showed that \(n\)-vertex planar graphs can be compressed into \(O(n)\) bits. The constant in front of the high order term was improved by Keeler and Westbrook [20], and He, Kao and Lu [15] later describe a technique that is optimal in the first order term. These results generalize to any graph with constant genus [22]. None of these results consider fast queries.

Jacobson [17] first showed how planar graphs can be represented using \(O(n)\) bits while permitting adjacency queries in \(O(\log n)\) time. Munro and Raman [27] improved the time for adjacency queries to \(O(1)\) time.
In my thesis I consider a much more general class of graphs: that of graphs with small separators. I say that a graph has small separators if it and all its subgraphs can be partitioned into two approximately equally sized parts by removing a relatively small number of vertices. (The expected separator size of a random graph is Ω(m − n)). Planar graphs, for example, have O(n^{1/2}) separators [21] and play an important role in any partitioning of 2-dimensional space, such as 2-dimensional triangulated meshes. Even graphs that are not strictly planar because of crossings, such as telephone and power networks, tend to have small separators. More generally, nearly all graphs that are used to represent connections in low dimensional spaces have small separators. For example, most 3-dimensional meshes have O(n^{2/3}) separators [23], as do most nearest neighbor graphs in 3-dimensions. Furthermore many graphs without pre-defined embeddings in low dimensional spaces have small separators. For example, the link structure of the web has small separators, as my experiments will show.

**Separability.** I say that a class of graphs satisfies a \( f(n) \)-vertex separator theorem if, for some constants \( \alpha \) and \( \beta \), for any graph \( G \) in that class, the vertices of \( G \) can be partitioned into sets \( A, B, \) and \( C \), such that \( |A| > \alpha |G| \), \( |C| > \alpha |G| \), \( |B| < \beta f(|G|) \), and no vertex in \( A \) has an edge to any vertex in \( C \). For example, the class of planar graphs satisfies a \( O(n^{1/2}) \)-vertex separator theorem. In this section I will consider only classes of graphs that are closed under the subgraph operation: that is, if \( G \) is in the class, then all subgraphs of \( G \) must be in the class as well.

I say that a class of graphs satisfies a \( f(n) \)-edge separator theorem if any graph can be partitioned into sets \( A, B, \) and \( C \) as above, where \( B \) is a set of edges whose removal separates \( A \) from \( C \). This is a stronger condition than the vertex separator condition: all graphs that satisfy an \( f(n) \)-edge separator theorem satisfy an \( O(f(n)) \)-vertex separator theorem, but the converse is not true.

In my thesis I will prove the following theorem:

**Theorem 2.1** For a class of graphs satisfying an \( n^{\epsilon} \)-vertex separator theorem, any \( n \)-vertex member can be represented in \( O(n) \) bits while supporting adjacency queries and degree queries in \( O(1) \) time and neighborhood queries in \( O(1) \) time per neighbor.

I will present a data structure that supports these operations in the given time and space bounds. Additionally I will present a simplified version of the data structure for the special case in which the class of graphs also satisfies an edge separator theorem. I will present experimental results from using this data structure to compress several varieties of real-world graphs. All of these graphs compress well, demonstrating that most real-world graphs are in fact separable.

### 3 Meshes

Many applications involve representing very large meshes. Meshes are used in air flow simulations, earthquake modeling, finite element analysis, computer graphics, and many other applications involving representation of complex 3-dimensional objects. For such applications the limiting factor can be the space required to represent the mesh in memory. The most efficient standard representations of meshes use 6 pointers per triangle in 2D or 8 pointers per tetrahedron in 3D. This is 48 bytes per vertex in 2D or over 200 bytes per tetrahedron in 3D.

One option for using larger meshes is to maintain the mesh in external memory. To avoid thrashing, this requires designing algorithms that carefully orchestrate how they access the mesh. Although several such external memory algorithms have been designed these algorithms can be much more complicated than their main-memory counterparts, and can be significantly slower.

Another option for using larger meshes is to try to compress the representation within main memory. There has in fact been significant interest in compressing meshes [12, 14, 35, 29, 31, 34, 19, 16, 13]. In three dimensions, for example, these methods can compress a tetrahedral mesh to less than a byte per tetrahedron [34]—about 6 bytes/vertex. These techniques, however, are designed for storing meshes on disk or for reducing transmission time, not for representing a mesh in main memory. They therefore do not support dynamic queries or updates to the mesh while in compressed form.

Here I am interested in compressed representations of meshes that permit dynamic queries and updates to the mesh. The goal is to solve larger problems while using standard main-memory algorithms. In my thesis I will present data structures for representing two and three dimensional simplicial meshes. The representations support standard operations on meshes including traversing among neighboring simplices, inserting and deleting simplices, and the ability to store data on simplices. For a class of well shaped meshes [23] these operations all take constant time. Although these representations are not as compact as those designed for disk storage, they
still save a factor of between 5 and 10 over standard representations.

**Separators.** The representation I present takes advantage of the separator property of well-shaped meshes [23] and uses an encoding similar to that of separable graphs. It uses separators to relabel the vertices so that vertices that share a simplex are likely to have labels that are close in value. (For applications that need to generate new vertices, e.g., Delaunay refinement, I leave extra space in the label space and assign new labels based on the labels of the neighbors. Pointers are then difference encoded using variable length codes. I use this technique to radially store the neighboring vertices around each vertex in 2D and around a subset of the edges in 3D. A query need only decode a single vertex in 2D or vertex and edge in 3D.

**Experimental Results.** An implementation of this structure uses about 5 bytes per triangle in 2D and about 7.5 bytes per tetrahedron in 3D when measured over a range of mesh sizes and point distributions. In my thesis I will present experiments based on using the representation as part of incremental Delaunay algorithms in both 2D and 3D. I demonstrate tests with a variant of the standard Bowyer-Watson algorithm [9, 38] and the exact arithmetic predicates of Shewchuk [32] for all geometric tests. I also present experiments based on a Delaunay refinement algorithm that removes triangles with small angles by adding new points at their circumcenters.

4 Text Indices

Memory considerations are a serious concern in the design of search engines. Some web search engines index over a billion documents, and even this is only a fraction of the total number of pages on the Internet. Most of the space used by a search engine is in the representation of an inverted index, a data structure that maps search terms to lists of documents containing those terms. Each entry (or posting list) in an inverted index is a list of the document numbers of documents containing a specific term. When a query on multiple terms is entered, the search engine retrieves the corresponding posting lists from memory, performs some set operations to combine them into a result, sorts the resulting hits based on some priority measure, and reports them to the user.

A naive posting list data structure would simply list all the document numbers corresponding to each term. This would require $\lceil \log(n) \rceil$ bits per document number, which would not be efficient. To save space, the document numbers are sorted and then compressed using difference coding (see Section 7).

In general, a difference-coding algorithm will get the best compression ratio if most of the differences are very small (but one or two of them are very large). Several authors [25, 7, 8] have noted that this is achieved when the document numbers in each posting list have high locality. These authors have designed methods to explicitly take advantage of this locality. These methods achieve significantly improved compression when the documents within each term have high locality. However, all compression methods thus far have been devoted to passive exploitation of locality that is already present in inverted indices.

In my thesis, I will describe how to improve the compression ratio of difference coding on an inverted index by permuting the document numbers to actively create locality in the individual posting lists. One way to accomplish this is to apply a hierarchical clustering technique to the document set as a whole, using the cosine measure as a basis for document similarity. My algorithm can then traverse the hierarchical clustering tree, applying a numbering to the documents as it encounters them. Documents that share many term lists should be close together in the tree and therefore close together in the numbering.

I have implemented this idea and tested it on indexing data from the TREC-8 ad hoc track [37] (disks 4 and 5, excluding the Congressional Record). I tested a variety of codes in combination with difference coding. My algorithm was able to improve the performance of the best compression technique I found by fourteen percent simply by reordering the document numbers. The improvement offered by my algorithm increases with the size of the index, so I believe the improvement on larger real-world indices would be greater.

5 Ordered Sets

As described above, memory considerations are a serious concern in the design of search engines. Most of the space used by a search engine is in the representation of an inverted index, a data structure that maps search terms to lists of documents containing those terms. Each entry (or posting list) in an inverted index is a list of the document numbers of documents containing a specific term. When a query on multiple terms is entered, the search engine retrieves the corresponding posting lists from memory, performs some set operations to combine them into a result, sorts the resulting hits based on some priority measure, and reports them to the user.

In this section I describe a data structure to compactly represent an individual posting list, represented as an ordered set $S = \{s_1, s_2, \ldots, s_n\}, s_i < s_{i+1}$, from a universe $U = \{0, \ldots, m-1\}$. This data structure should
support dynamic operations including set union and intersection, and it should operate in a purely functional setting [18] since it is desirable to reuse the original sets for multiple queries. In a purely functional setting data cannot be overwritten. This means that all data is fully persistent.

Related Work. The set union and intersection problems are special cases of the list merging problem, which deals with combining two sorted lists into one. Moffat, Petersson, and Wormald [24] show that list merging can be done in $O(k \log \frac{|S_1| + |S_2|}{k})$ time by any structure that supports fast split and join operations. (Here, $k$ is the Block Metric bound: the number of contiguous blocks of $S_1$ and $S_2$ that appear in their union.) A split operation is one that, given an ordered set $S$ and a value $v$, splits the set into sets $S_1$ containing values less than $v$ and $S_2$ containing values greater than $v$. A join operation is one that, given sets $S_1$ and $S_2$, with all values $S_1$ less then the least value in $S_2$, joins them into one set. These operations are said to be fast if they run in $O(\log(\min(|S_1|, |S_2|)))$ time. In fact, the actual algorithm requires only that the split and join operations run in $O(\log(|S|))$ time.

There has been significant research on succinct representation of sets taken from $U$. I consider the following operations:

- **Search** (Search$^+$): Given $x$, return the greatest (least) element of $S$ that is less than or equal (greater than or equal) to $x$.
- **Insert**: Given $x$, return the set $S' = S \cup \{x\}$.
- **Delete**: Given $x$, return the set $S' = S \setminus \{x\}$.
- **FingerSearch** (FingerSearch$^+$): Given a handle (or “finger”) for an element $y$ in $S$, perform **Search** (Search$^+$) for $x$ in $O(\log d)$ time where $d = |\{s \in S \mid y < s < x \lor x < s < y\}|$.
- **First, Last**: Return the least (or greatest) element in $S$.
- **Split**: Given an element $x$, return two sets $S' = \{y \in S \mid y < x\}$ and $S'' = \{y \in S \mid y > x\}$, plus $x$ if it was in $S$.
- **Join**: Given sets $S'$, $S''$ such that $\forall x \in S', \forall y \in S'', x < y$, return $S = S' \cup S''$.
- **(Weighted)Rank**: Given an element $y$ and weight function $w$ on $S$, find $r = \sum_{x \in S, x < y} w(x)$. In the unweighted variant, all weights are considered to be 1.
- **(Weighted)Select**: Given $r$ and a weight function $w$, find the greatest $y$ such that $\sum_{x \in S, x < y} w(x) \leq r$. Return both $y$ and the associated sum. In the unweighted variant, all weights are considered to be 1.

Given any ordered dictionary structure $D$ supporting certain subsets of these operations and using $O(n \log m)$ bits to store $n$ values from $U = \{0, \ldots, m-1\}$, I demonstrate a simple blocking technique that produces an ordered set structure using $O(n \log \frac{m+n}{n})$ bits. This is within a constant factor of the information-theoretic lower bound. The technique requires that the target machine have a word size of $\Omega(\log m)$. This is reasonable since log $m$ bits are required to distinguish the elements of $U$. The technique also makes use of a lookup table of size $O(m^\alpha \log^\alpha m)$ for any $\alpha > 0$.

In my thesis I will prove the following theorem:

**Lemma 5.1** For an ordered universe $U = \{0, \ldots, m-1\}$, given an ordered dictionary structure (or comparison-based ordered set structure) $D$ that uses $O(n \log m)$ bits to store $n$ values, my blocking technique produces a structure that uses $O(n \log \frac{n+m}{n})$ bits.
1. If $D$ supports $\text{Search}^-$, $\text{Search}^+$, $\text{Insert}$, and $\text{Delete}$, the blocked set structure supports those operations using $O(1)$ instructions and $O(1)$ calls to operations of $D$.

2. If $D$ supports $\text{FingerSearch}$, the blocked set structure supports $\text{FingerSearch}$ in $O(1)$ instructions and one call to $\text{FingerSearch}$. If $D$ supports $\text{Insert}$ and $\text{Delete}$ at a finger, then the blocked set structure supports those operations using $O(1)$ instructions and $O(1)$ calls to $\text{Insert}$ and $\text{Delete}$ at a finger.

3. If $D$ supports the $\text{First}$, $\text{Last}$, $\text{Split}$, and $\text{Join}$ operations, then the blocked set structure supports those operations using $O(1)$ instructions and $O(1)$ calls to operations of $D$.

4. If $D$ supports the $\text{WeightedRank}$ operation, then the blocked set structure supports the $\text{Rank}$ operation in $O(1)$ instructions and one call to $\text{WeightedRank}$. If $D$ supports the $\text{WeightedSelect}$ operation, then the blocked set structure supports the $\text{Select}$ operation using $O(1)$ instructions and one call to $\text{WeightedSelect}$.

6 Efficient Arrays and Dictionaries

A common thread in the data structures I have discussed is the need to store variable-length bit strings. For graphs and meshes, the amount of data stored for each vertex is variable; for ordered sets, the size of the difference between each pair of elements is variable. In my thesis I will describe a generic approach to storing variable length bit strings in “arrays” and dictionaries. My main motivation is to use these as building blocks for other applications. I describe applications of the data structures to graphs, simplicial meshes, cardinal trees with nodes of varying cardinality and to ordered sets. These applications either generalize or simplify previously known results. I assume the machine has a word length $w > \log |C|$, where $|C|$ is the number of bits used to represent the collection. I assume $|s| \geq 1$ for all bit-strings $s$.

The array problem is simply to support an array of $n$ locations $(1, \ldots, n)$ with lookup and update operations. I denote the $i^{th}$ element of an array $A$ as $A_i$. In this case each location will store a bit-string. In my thesis I will present a data-structure that uses $O(m)$ space where $m = \sum_{i=1}^{n} |A_i|$, supports lookups in $O(1)$ worst-case time and updates in $O(1)$ expected amortized time. If all bit-strings are the same length then this is of course trivial.

The set problem is to support a set with insertion, deletion and membership queries. In my thesis I will present a data-structure that uses $O(m)$ space, where $m = \max_{s \in S} \left( \sum_{i=1}^{m} |s_i| - \log |S| \right)$, and supports membership in $O(1)$ time and insertion and deletion in $O(1)$ expected amortized time. Brodnik and Munro [11] describe a set data structure that can store sets $S \subset \{0, 1, \ldots, m - 1\}$ using $O(n \log((m + n)/n))$ space, which is asymptotically optimal. As with this structure it supports membership in $O(1)$ time and insertion and deletion in $O(1)$ expected amortized time. Since values from $S$ can be represented as log $n$ bit strings, the bounds $O(n(1 + \log m - \log n))$ match their bounds. Their results are therefore a special case of these for fixed-length strings.

The dictionary problem is to support a dictionary with insertion, deletion, and lookup. Each element consists of a key and a value, represented as the pair $(s, v)$. I assume both the key and the value are variable-length strings. By attaching satellite data to the set data structure I can support dictionaries using $O(m)$ space, where $m = \sum_{(s, v) \in D} (|s| + \max(|s| - \log |D|)$, membership in $O(1)$ time, and insertion and deletion in $O(1)$ expected amortized time. I note that this result immediately implies the results for arrays by simply using the array index $i$ as the key and the value stored at $i$ as the value. I use the array implementation to implement sets and dictionaries, however, so I describe it independently.

**Applications.** For graphs I 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 $O(w + n)$ where $w = \sum_{(u,v) \in E} \log |u - v|$, and $n = |V|$. Any graph from a class satisfying an $n^{1-c}$ edge-separator theorem can be labeled so that $w < kn$ 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 [6] by permitting insertion and deletion of edges. I 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 I know this is first compact dynamic graph representation of any kind.

For ordered sets $S \subset \{0, \ldots, m - 1\}$ I support the same operations in the same bounds as described in Section 5, except that the updates are expected amortized time here and were worst case time bounds there. The
structure I describe here, however, is simpler and quite different from the previous structure. It also allows for attaching satellite data to each key consisting of variable length bit strings, to support dictionaries.

For cardinal trees I support a tree in which each node can have a different cardinality. Queries can request the $k^{\text{th}}$ child, or the parent of any vertex. Again I can also attach satellite bit-strings to each vertex. Updates can add or delete the $k^{\text{th}}$ child. For an integer labeled tree the space bound is $O(m)$ where $m = \sum_{v \in V} (\log c(p(v)) + |v - p(v)|)$, 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, 30] to varying cardinality. Again, I do not match the optimal bounds in the first order term.

I define a $d$-simplicial mesh to be a pure simplicial complex of dimension $d$, which is a manifold, possibly with a boundary. For $d$-simplicial meshes I 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 I 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 $O(w + n)$ where $w = \sum_{(u,v) \in S} \log |u - v|$, $n = |V|$, and $S$ are the edges in the 1 skeleton of the mesh, assuming bounded degree. As usual, updates take $O(1)$ amortized expected time and queries take $O(1)$ worst case time. I note that this is very similar to the mesh structure given in a previous section.

7 Coding Techniques

A key concept for all of the representations I discuss is that of difference coding. Difference coding involves representing a value by its offset, or difference, from another value. Combined with logarithmic codes, values can be represented using many fewer bits than a naive representation would require.

Logarithmic Codes. A prefix code can be described as a correspondence $\mathbb{N} \rightarrow S$, where $S \subset \{0, 1\}^*$ such that no $s \in S$ is a prefix of any $s' \in S$. A value is encoded as the corresponding bit string; multiple values are encoded as the concatenation of the bit strings. Because no value’s string is a prefix of any other string, this can be decoded uniquely in one left-to-right pass.

A logarithmic code is a prefix code such that for any value $v$, the length of the code for $v$ is $O(\log v)$ bits.

An example of such a code is the gamma code, which stores a value $v$ in two parts. The first part is a unary code describing the value’s length: it consists of $\lfloor \log v \rfloor$ zeroes. The second part is, simply, the binary code for the value itself. Note that the binary code always begins with a one, so the number of zeroes in the first part can always be uniquely determined.

The binary code for $v$ requires $1 + \lfloor \log v \rfloor$ bits, so the total size of a gamma code is $1 + 2\lfloor \log v \rfloor$ bits, which is $O(\log v)$.

A useful class of logarithmic codes is that of chunked codes. Codes in this class represent values using chunks of $k$ bits each. Out of each chunk, $k - 1$ bits denote the least significant bits of $v - 1$. The remaining bit is a continue bit. If all of $v$ is described by the $k - 1$ bits, the continue bit is set to zero; otherwise, it is set to one, and the chunk is followed by a recursive encoding of $(v - 1) >> (k - 1)$.

A chunked code with $k = 8$ represents all values with byte-sized chunks. This is convenient since the decoding operation does not require shifting to extract a chunk. Such a code is called a byte code. A chunked code with $k = 4$ is called a nibble code, and with $k = 2$ it is called a snip code. Chunked codes represent a value $v$ with (approximately) $k \log_2 v$ bits, which is $O(\log v)$.

In terms of coding efficiency a chunked code with parameter $k$ is identical to a skewed Bernoulli code [39] with parameter $k - 1$.

Difference Coding. When storing an increasing sequence of values $v_1, v_2, v_3, \ldots, v_k$ from a set $S = \{1 \ldots n\}$ ($k \leq n/2$), considerable space can be saved by representing the values by their differences. That is, I let

\[ d_i = v_{i+1} - v_i \quad \text{and} \quad d_0 = v_1, \]

and store the sequence $d_0 \ldots d_{k-1}$. If logarithmic codes are used to encode these differences, then the total space needed to encode the numbers is $\sum_{i} O(\log d_i) = O(\sum_{i} 1 + \log d_i)$ bits. In the worst case, all the $d_i$ are equal (and each has value $n/k$), so the space used is $O(k + \log \log \frac{n}{k}) = O(k \log \frac{n}{k})$.

The number of increasing sequences of $k$ values from $S$ is $\binom{n}{k}$; at least $\Omega(\log \binom{n}{k}) = \Omega(k \log \frac{n+k}{k})$ bits are needed to tell them apart. Thus the bound from difference coding is within a constant factor of optimal.

Rank and Select. When dealing with variable-length compression schemes it frequently occurs that many variable-length codewords are concatenated to form a single large string. When this happens it is useful to allow a means of accessing individual words from the string.
This operation can be formally defined as the Select operation, as follows: Given a bitstring \( S \) of length \( n \), construct an auxiliary data structure \( s \) such that the operation \( \text{Select}(i,s) \) returns the bit position in \( S \) of the \( i \)th one. (To apply that data structure to this problem, imagine constructing a bit string containing a one at the start of each codeword, and zeroes everywhere else.) This problem has been studied extensively, and there are various data structures that solve this problem using \( o(n) \) space and use \( O(1) \) time per query.

For theoretical purposes I make use of the data structures of Munro [26]. For practical implementations I will generally use simpler structures which have \( O(n) \) space usage but better constants.

8 Expected Contributions

My thesis will describe a variety of techniques using vertex relabeling and difference coding to produce space-and time-efficient data structures for a variety of problems. These problems include graph compression, mesh construction, text indexing, set compression, and construction of efficient arrays and dictionaries. I will present experimental results demonstrating the practical value of these data structures.

At present I have completed work on data structures for graph [6], mesh [5], and set [4] compression. The graph and set compression code is fairly presentable; I am in the process of rewriting the mesh compression code to be better organized.

I have completed the reordering portion of an index compression data structure [3] and will implement the structure as part of my thesis.

I have plans for the construction of efficient arrays and dictionaries but have not yet proven theoretical bounds on them.

9 Execution Plan

<table>
<thead>
<tr>
<th>Spring 2004</th>
<th>Finish writing mesh compression code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summer 2004</td>
<td>Write index compression code</td>
</tr>
<tr>
<td>Fall 2004</td>
<td>Implement efficient arrays and hashtables</td>
</tr>
<tr>
<td>Spring 2005</td>
<td>Write thesis</td>
</tr>
</tbody>
</table>

References


