Compact Data Structures with Fast Queries

Daniel K. Blandford

CMU-CS-05-196
February 2006

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy E. Blelloch, chair
Christos Faloutsos
Danny Sleator
Ian Munro, Waterloo

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

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

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation.

Keywords: Data compression, text indexing, meshing

Abstract

Many applications dealing with large data structures can benefit from keeping them in compressed form. Compression has many benefits: it can allow a representation to fit in main memory rather than swapping out to disk, and it improves cache performance since it allows more data to fit into the cache. However, a data structure is only useful if it allows the application to perform fast queries (and updates) to the data.

This thesis describes compact representations of several types of data structures including variable-bit-length arrays and dictionaries, separable graphs, ordered sets, text indices, and meshes. All of the representations support fast queries; most support fast updates as well. Several structures come with strong theoretical results. All of the structures come with experimental results showing good compression results. The compressed data structures are usually close to as fast as their uncompressed counterparts, and sometimes are faster due to caching effects.

Acknowledgments

Thanks to my advisor, for being awesome. Thanks to my thesis committee. Thanks to people who helped proofread, including Ann Blandford, Benoit Hudson, Rebecca Lambert, William Lovas, Tom Murphy VII, and Allison Naaktgeboren.

Thanks to my parents.

Contents

Chapter 1

Introduction

Many applications dealing with large data structures can benefit from keeping them in compressed form. Compression has many benefits: it can allow a representation to fit in main memory rather than swapping out to disk, and it improves cache performance since it allows more data to fit into the cache. However, a data structure is only useful if it allows the application to perform fast queries (and updates) to the data.

There has been considerable previous work on compact data structures [68, 91, 29, 46]. However, most of the previous work has been exclusively theoretical, in that the structures are too complex to implement or suffer from very high associated constant factors. Further, the compression techniques used in previous work have been ad-hoc and are usually specific to the data structure being compressed. This work uses a unified approach based on difference coding to achieve practical compact representations for a wide variety of structures.

This thesis describes compact representations of several types of data structures including variable-bit-length arrays and dictionaries, separable graphs, ordered sets, text indices, and meshes. All of the representations support fast queries; most support fast updates as well. Several structures come with strong theoretical results:

- The variable-bit-length dictionaries generalize recent work on dynamic dictionaries [29, 103] to variable-length bit-strings.
- The ordered set structure supports a wider range of operations than previous compact structures for sets [29, 96].
- The graph structures represent a generalization of previous work [46, 65, 68, 91, 40] and are the first dynamic compact structures known.

All of the structures come with experimental results showing good compression results. The compact data structures are usually close to as fast as their uncompressed counterparts, and sometimes are faster due to caching effects.

These data structures are united by a common theme: the use of *difference coding* (see Section 2.4) to represent data by its difference from other, previously known, data. For example, a compact graph

Structure	Chp	Space (in bits)	Operations
Arrays$\{s_1 \ldots s_n\}$	3.2	$O(\sum_i \|s_i\|)$	$O(1)$ `lookup` $O(1)$ exp amort `insert`
Dictionaries $\{(s_1, t_1) \ldots (s_n, t_n)\}$	3.3	$O(\sum_i \max(\|s_i\| - \log n, 1) + \|t_i\|)$	$O(1)$ `lookup` $O(1)$ exp amort `map`
Cardinal Trees (cardinality of v is $c(v)$)	3.4	$O(\sum_v 1 + \log(1 + c(\text{parent}(v))))$ (semidynamic)	$O(1)$ `parent/child` $O(1)$ exp amort `insert/delete`
Ordered Sets $\{s_1 \ldots s_n\}$	4	$O(\sum_i \log(s_{i+1} - s_i))$	$O(k \log \frac{\|S_1\|+\|S_2\|}{k})$ `union/intersect` (k is Block Metric [31]) many more
Graphs (vtx separable)	5.2	$O(n)$ (static)	$O(1)$ `getDegree` $O(1)$ `adjacent` $O(1)$ per neighbor `listNeighbors`
Graphs (edge separable)	5.3	$O(n)$ (semidynamic)	as above, plus $O(1)$ exp amort `insert/delete`
Text Indices	6	14.4% additional compression	same as original index
2D Simplicial Meshes (well shaped)	7.3	$O(n)$ (semidynamic)	$O(1)$ `findTriangle`(v_1, v_2) $O(1)$ exp amort `insert/delete`
3D Simplicial Meshes (well shaped)	7.3	$O(n)$ (semidynamic)	$O(1)$ `findTetrahedron`(v_1, v_2, v_3) $O(1)$ exp amort `insert/delete`

Table 1.1: Space bounds and operations supported for our data structures. Structures that are marked as *semidynamic* have space bounds that depend on the locality of a vertex labeling (see Section 5.2 for details).

structure represents the neighbors of a vertex by the difference between the neighbor label and the original vertex label. For many structures this is combined with a relabeling scheme which ensures that most of the differences encoded are small. (This relabeling effect is shown visually in Figure 1.1.) The variable-bit-length arrays and dictionaries represent a general framework for creating compressed queryable data structures. This represents an improvement for many structures, which would otherwise need to be built ad-hoc.

We describe our data structures as *compact*, meaning that they use a number of bits that is within a constant factor of the optimal bound. The structures, and the bounds corresponding to those structures, are summarized in Table 1.1.

Arrays and Dictionaries (Chapter 3). In the design of compact data structures, two useful building blocks are the variable-bit-length "array" and dictionary structures. The "array" structure maintains a set of bit strings numbered $0 \ldots (n-1)$, permitting constant-time `lookup` and expected amortized constant-time `update` operations. The dictionary structure permits constant-time `lookup` and expected constant-time `map` operations in which both keys and data are variable-length bit strings. In each case the space usage is within a constant factor of optimal. This represents a generalization of recent work on dynamic dictionaries [29, 103] to variable-length bit strings (although it does not match the optimal constant on the high-order term of its space usage).

Using these variable-bit-length data structures it is possible to implement a wide variety of compressed data structures with fast queries. One example is a compressed representation of cardinal trees in which the degree can vary per node (described in Section 3.4). Finding the parent or the kth child of a node takes constant time, and the space usage is within a constant factor of optimal. Other applications appear throughout this thesis.

Section 3.5 presents experimental results from using the dictionary to store variable-bit-length data describing edges in a tetrahedral mesh (see Chapter 7 for more details). The dictionary can be implemented using various types of difference codes representing different tradeoffs between compression and speed. Using the *byte code* (described in Section 2.3), the dictionary is a factor of 6.5 more space-efficient than a naive hashtable structure. For small input sizes the dictionary is a factor of 1.7 slower than the hashtable; for larger input sizes, the two are nearly equivalent in speed. The difference is due to caching effects, in that the dictionary can fit into cache much better than the hashtable.

Ordered Sets (Chapter 4). One important application for data compression is in the compact representation of ordered sets. Chapter 4 presents a compact representation for sets of integers from some fixed range $U = \{0 \ldots m-1\}$. The representation supports a wide range of operations while maintaining the data in a compressed form. This is based on 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.

For example, applying this technique to a functional implementation of treaps produces a compressed data structure which supports rapid set union and intersection operations. The time required to compute the union or intersection of two sets S_1, S_2 is optimal $O(k \log \frac{|S_1|+|S_2|}{k})$ where k is the Block Metric of Carlsson, Levcopoulos, and Petersson [31]. The space required per set S is $O(|S| \log \frac{|U|+|S|}{|S|})$ bits, which matches the information-theoretic lower bound. This is an improvement over the dynamic compressed-set structures of Brodnik and Munro [29] and Pagh [96], which are based on hashing and thus do not support fast union and intersection.

Representations of ordered sets are useful for many applications. In particular, search engines maintain *posting lists* which describe, for each possible search term, the set of documents containing that term. These posting lists are represented as ordered sets of document numbers. The compact functional treaps described above provide a means to maintain posting lists in compressed form while still permitting fast union and intersection operations.

Section 4.7 contains experimentation describing the performance of compressed red-black trees (using the C STL implementation) and functional treaps. For the largest problem size tested (insertion and deletion of 2^{18} elements from $U = \{0 \ldots 2^{30}-1\}$), the compressed red-black trees took twice as long but used only $1/3$ as much space as the uncompressed trees. The quality of compression is better for denser sets (as predicted by the space bound given above).

Separable Graphs (Chapter 5). Recently there has been a great deal of interest in compact representations of graphs [125, 72, 65, 82, 64, 105, 92, 68, 91, 40, 46, 65, 28, 1, 119, 22]. Using difference coding it is possible to create several different compact representations for separable graphs. (A graph is defined to be *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 representations are based on relabeling the vertices using graph separators (as shown in Figure 1.1), then encoding a vertex's neighbors by their difference from the original vertex. The first representation given is a simple static structure based on edge separators; the second is a more general structure based on vertex separators. The third representation is a dynamization of the first representation, supporting adding and removing edges (v_1, v_2) in expected amortized $O(|v_1| + |v_2|)$ time (where $|v|$ is the degree of v). It makes use of the variable-bit-length array structure from Chapter 3. The fourth representation is a dynamic structure that supports adding and removing edges in expected amortized $O(1)$ time using the variable-bit-length dictionary structure from Chapter 3. The static representations use $O(n)$ bits for separable graphs. The dynamic representations use $O(n)$ bits as well, but the space bound is "semidynamic" in that it depends on the labeling of the vertices remaining good as the graph is updated.

The static representations described here are an improvement over the work of Deo and Litow [46] and He, Kao and Lu [65], who use separators for graph compression but do not support queries. They are a generalization of the work of Jacobson [68], Munro and Raman [91], and Chuang et. al. [40], who support queries on compressed planar graphs (but not the more general case of separable graphs). The dynamic representations we describe are the first compressed dynamic graph representations we know of.

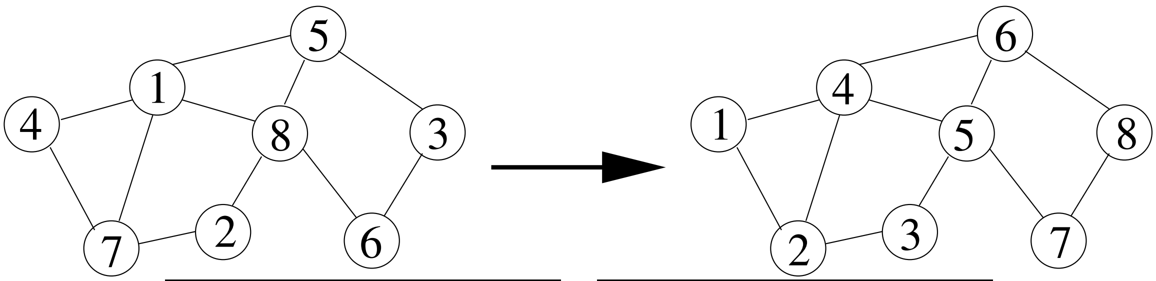
Section 5.7 contains detailed experimentation for the first and third representations. Using the byte code, the static representation is less than 10% slower than a standard neighbor-array representation, but uses a factor of 3 less space. The dynamic representation uses a factor of 4 less space than a linked-list representation. The time performance of a linked-list representation is strongly dependent on the locality of the linked-list pointers. The compressed dynamic representation is usually faster than a linked-list, and is within 20% of the linked-list's speed even when the linked-list is laid out in order.

Text Indices (Chapter 6). The idea of separator-based reordering (from Chapter 5) can also be applied to the problem of index compression. This gives a heuristic technique which uses document relabeling to reduce the space used when representing posting lists as ordered sets (as described in Chapter 4).

Posting lists are 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 $\{0, \ldots, n-1\}$. (In fact, the Binary-Interpolative code of Moffat and Stuiver [88] was designed to take advantage of such locality.) Locality is produced when similar documents are close together in the numbering. The reordering technique renumbers the documents to accomplish this.

Section 6.4 contains experimentation involving compressing an index of disks 4 and 5 of the TREC database. The reordering algorithm runs in a matter of minutes and improves the compression quality by over 14%.

When this material was first published, there had been no previous work on the subject. Since then, several authors [113, 115, 11] have addressed the topic. Their contributions are discussed in Section 6.1.



Before reordering		
Vtx	Neighbors	Differences
1	4,5,7,8	3,1,2,1
2	7,8	5,1
3	5,6	2,1
4	1,7	-3,6
5	1,3,8	-4,2,5
6	3,8	-3,5
7	1,2,4	-6,1,2
8	1,2,5,6	-7,1,3,1

After reordering		
Vtx	Neighbors	Differences
1	2,4	1,2
2	1,3,4	-1,2,1
3	2,5	-1,3
4	1,2,5,6	-3,1,3,1
5	3,4,6,7	-2,1,2,1
6	4,5,8	-2,1,3
7	5,8	-2,3
8	6,7	-2,1

Figure 1.1: Several of our compression techniques use a relabeling step to ensure that the vertex labels of a graph have good locality. This decreases the cost of difference coding the edges.

Meshes (Chapter 7). Difference coding can also be used in compact representations for triangular and tetrahedral meshes. Standard mesh representations use a minimum of 6 pointers (at least 24 bytes) per triangle in 2D or 8 pointers (32 bytes) per tetrahedron in 3D. The compact representations described here use as little as 5 bytes per triangle or 7.5 bytes per tetrahedron. This is important for many applications since meshes are often limited by the amount of RAM available.

Chapter 7 describes two mesh representations. One is based on storing difference-encoded triangles (or tetrahedra) in a variable-bit-length dictionary structure (as described in Chapter 3) and has constant expected amortized time for insertion and deletion of simplices. The other representation is based on difference coding and storing the cycle of neighbors around a vertex in 2D or the cycle of vertices around an edge in 3D. That representation takes $O(|v|)$ expected time for dealing with a vertex of degree $|v|$, but the compression has a more favorable constant.

This is the first work we know of dealing with dynamic compressed meshes.

Section 7.6 contains experimentation involving the representation that compresses based on cycles. The representation is used to construct 2D and 3D Delaunay meshes. The 2D representation is about 10% slower than Shewchuk's Triangle code [110]; the 3D representation is slightly faster than our beta version of Shewchuk's Pyramid code [109].

A Parallel Meshing Algorithm (Chapter 8). The Delaunay meshing algorithm from Chapter 7 can be parallelized. The variable-bit-length dictionary structure is modified to support locks to prevent concurrent access. Experimentation shows that the resulting algorithm can rapidly generate a mesh of over 10 billion tetrahedra (using 1.51 billion vertices randomly chosen from the unit cube). The algorithm took 6036 seconds for 64 processors on an HP GS 1280 SMP machine; this was a speedup of 34.25 compared to its performance on one processor. All data (including vertex coordinates, mesh connectivity data, and the work queue) fit within a memory footprint of 197GB of RAM.

Chapter 2

Preliminaries

This chapter discusses some concepts which will be useful throughout this document.

2.1 Terminology

Throughout this thesis, when dealing with a graph G we let n denote the number of vertices of G and m denote the number of edges of G. The degree of a vertex v is written $|v|$. Without loss of generality we assume all vertices have degree at least 1.

Given a bitstring s we let $|s|$ denote the number of bits in the string.

We denote a dictionary entry mapping key k to data d by $((k), (d))$. For some applications either the key or data may be a tuple: $((k_1, k_2), (d_1, d_2))$.

All logarithms are base 2.

2.2 Processor Model

Throughout all of our work we assume 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 the processor supports operations including bit-shifts (multiplication or division by powers of 2) as well as bitwise AND, OR, and XOR.

For some theoretical bounds we make use of *table-lookup* operations. A table-lookup operation makes use of a *lookup table* of size $2^{\epsilon w}$ entries. Each entry in the table contains the result of the operation on the bitstring corresponding to the entry. Examples of table-lookup operations are given in Section 2.3 and 2.5.

If each entry contains $O(\epsilon w)$ bits, then the total space used by the lookup table is $O(2^{\epsilon w} \epsilon w)$ bits. By simulating a word size of $\Theta(\log |C|)$ this can often 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 packing multiple small words into a larger one.

	Unary	Binary	Gamma	Nibble
1	1	1	1	0000
2	01	10	010	0001
3	001	11	011	0010
4	0001	100	00100	0011
5	00001	101	00101	0100
6	...	110	00110	0101
7	...	111	00111	0110
8	...	1000	0001000	0111
9	...	1001	0001001	10000000
10	...	1010	0001010	10010000
11	...	1011	0001011	10100000
12	...	1100	0001100	10110000
13	...	1101	0001101	11000000
14	...	1110	0001110	11010000
15	...	1111	0001111	11100000
16	...	10000	000010000	11110000
17	...	10001	000010001	10000001

DECODE-GAMMA(B)
  $\ell \leftarrow 0$
  do
    $b \leftarrow B[\ell \ldots \ell + \epsilon w - 1]$
    $\ell \leftarrow \ell +$`first-one`$[b]$
  loop while(`first-one`$[b] = \epsilon w$)
  $\gamma \leftarrow B[0 \ldots 2\ell]$
  return (`(int)`$\gamma, 2\ell + 1$)

Figure 2.1: Left: The Unary, Binary, Gamma and Nibble codes. Right: Pseudocode for the DECODE-GAMMA algorithm.

2.3 Variable-Length Coding

A *variable-length code* represents a positive integer v using a variable number of bits. An example of a variable-length code is the *unary code*, which represents v using $v - 1$ zeroes followed by a one. Another example is the *binary code*, which represents v using the $(\lfloor \lg(v) \rfloor + 1)$-bit binary representation of v. Examples of these codes are shown in Figure 2.1.

When using variable-length codes for compression, it is useful to concatenate large numbers of codes together for storage. For this it is convenient to use *prefix-free codes*. A prefix-free code is a variable-length code for which there do not exist positive integers $v \neq v'$ such that the code for v is a prefix of the code for v'. Prefix-free codes have the property that, when the codes for many integers are concatenated, the resulting string has a unique decoding.

As an example, the binary code is not a prefix-free code: the string `10110` can be read as the concatenation of the codes for 5 and 2, the concatenation of the codes for 2 and 6, the single code for 22, et cetera. It is possible to convert the binary code into a prefix-free code by prepending to each codeword a number of zeroes equal to that codeword's length minus one. This code is the *gamma code* [50]. The gamma code is only one of a wide class of prefix-free codes (see [136] for many others). For theoretical work this thesis will use gamma codes as they are easy to describe and conceptually easy to encode and decode.

Decoding gamma codes. Using a lookup table of size $O(2^{\epsilon w} \log(\epsilon w))$ it is possible to decode gamma codes in $O(\frac{|s|}{\epsilon w} + 1)$ time, where $|s|$ is the length of the code (and ϵ is a parameter). Given a bitstring B which is the concatenation of several gamma codes, the algorithm DECODE-GAMMA finds the size of the first code and the value it represents.

The first step is to compute the location of the first 1 in B. For this the algorithm makes use of a pre-computed lookup table `first-one`, defined as follows: If b is a bitstring of size ϵw, then `first-one`(b) gives the location of the first 1 in b (or ϵw if b contains no 1s). The algorithm examines ϵw-bit chunks of B until it finds a chunk containing at least one 1. The algorithm uses the table to find the bit-position of the first 1, and from this deduces the total bit-length of the gamma code. The algorithm extracts the code from B using shifts. Once the code is extracted, decoding it is equivalent to reinterpreting it as a binary integer. Pesudocode for this algorithm is shown in Figure 2.1.

For many of the applications we will examine, all values encoded in our data structures will be $O(|C|)$ (where $|C|$ is the number of bits used by the structure). For these applications we use a table word size of $\frac{\log |C|}{2}$, giving a space usage of $O(|C|^{.5} \log |C|)$, which is $o(|C|)$. The time required for DECODE-GAMMA is $O(\frac{|s|}{\log |C|})$, which is $O(1)$.

Byte-aligned codes. Gamma codes are easy to describe in theory; however, for implementation the use of large lookup tables is undesirable. It is more convenient to work with a class of *byte-aligned codes*. These codes have sizes that fall along byte boundaries, making them easy to manipulate.

These codes are special 2-, 4-, and 8-bit versions of a more general k-bit code which encodes integers as a sequence of k-bit blocks. We describe the k-bit version. Each block starts with a *continue bit* which specifies whether there is another block in the code. An integer i is encoded by checking whether it is less than or equal to 2^{k-1}. If so, a single block is created with a 0 in the continue bit and the binary representation for $i - 1$ in the other $k - 1$ bits. If not, the first block is created with a 1 in the continue-bit and the binary representation for $(i - 1) \mod 2^{k-1}$ in the remaining bits (the $\mod$ is implemented with a bitwise and). This block is then followed by the code for $\lfloor (i - 1)/2^{k-1} \rfloor$ (the / is implemented with a bitwise shift).

The 8-bit version of this code is particularly fast to encode and decode since all its memory accesses are byte-aligned (and since it makes use of fewer continue bits). The 4-bit version (nibble code) and 2-bit version (snip code) are often more space-efficient, but are somewhat slower since they require more bit-manipulation during encoding and decoding.

As an optimization, to further improve the time-performance of the 8-bit code, for that code we do not subtract one from i at each iteration. Thus we store the binary representation of $(i \mod 2^{k-1})$ in each block, followed if necessary by a code for $\lfloor i/2^{k-1} \rfloor$. This can sometimes use more space, but it permits faster encoding and decoding since those operations require only bit-shifts (rather than addition and subtraction). We refer to this variant as the *byte code*. Performance of these codes is compared in detail in Section 5.7.

Throughout the rest of this thesis we will assume that all variable-length codes used are prefix-free codes.

2.4 Difference Coding

Variable-length codes are a way to compactly represent values which are "on average" small. For many applications, the data to be represented are not small values; however, it is often possible to represent a value by its difference from previously known values. The resulting difference is more likely to be small. This is known as *difference coding*.

One common form of difference coding is in the encoding of a set of n integers from the set $\{1 \ldots n\}$. An information-theoretic lower bound on the space needed to represent n elements from m possibilities is $\Omega(\log \binom{m}{n})$ bits; assuming $n \leq m/2$, this is $\Omega(n \log \frac{m}{n})$.

Let $x_1 \ldots x_n$ be the integers to be stored, in sorted order such that $x_i < x_{i+1}$. x_1 is stored directly, but the remaining values are represented by their difference from the previous value: $x_1, x_2 - x_1, x_3 - x_2, x_4 - x_3, \ldots, x_n - x_{n-1}$. The codes are concatenated into a single bitstring for storage.

Gamma codes require $2\lfloor \lg(v) \rfloor + 1$ bits to represent a value v. If the differences above are represented by gamma codes, then the total space required is $2\lfloor \lg(x_1) \rfloor + 1 + \sum(2\lfloor \lg(x_i - x_{i-1}) \rfloor + 1)$ bits. The worst case (greatest space usage) for this expression occurs when the x_is are equally spaced (that is, $x_i \simeq \frac{im}{n}$). The space usage is then $O(n \log \frac{m}{n})$ bits, which is within a constant factor of the optimal bound given by information theory.

In fact it is not necessary to use gamma codes to achieve this performance; any code using $O(\log v)$ bits to store a value v will suffice. We call such a code a *logarithmic code*.

In our example here the goal was to encode a set of values from $\{1 \ldots m\}$. In subsequent chapters we will explore many more applications for difference coding.

2.5 Decoding Multiple Gamma Codes

Suppose that a set of integers $x_1 \ldots x_k$ are difference coded and concatenated into a bitstring B. This section describes how to quickly access the encoded data. In particular, we consider the problem: given B and a value v, find the greatest i such that $x_i < v$. To do this it is necessary to decode and sum the gamma codes for $x_1, x_2 - x_1, x_3 - x_2, \ldots$ until, after summing $i + 1$ codes, the total reaches v. Our algorithm will return the value x_i and the bit-position of the gamma code for $x_{i+1} - x_i$.

One method for solving this problem would use the DECODE-GAMMA operation from Section 2.3, which can decode a gamma code of length $|s|$ in $O(\frac{|s|}{\epsilon w}+1)$ time. To decode i codes of total length $|S|$ would require $O(\frac{|S|}{\epsilon w} + i)$ time. This section will describe the SUM-GAMMA-FAST operation, which uses a more powerful table-lookup step to decode i codes of total length $|S|$ in $O(\frac{|S|}{\epsilon w} + 1)$ time.

To decode multiple gamma codes at once, the SUM-GAMMA-FAST algorithm makes use of two lookup tables `sum-of-codes` and `end-of-codes`, defined as follows: given a bitstring b of size ϵw, `sum-of-codes`(b) gives the sum of all the full gamma codes in b and `end-of-codes`(b) gives the bit-position of the end of the last full gamma code in b. Using these tables the SUM-GAMMA-FAST algorithm can decode and sum up to ϵw gamma codes at once. If the algorithm encounters a gamma code of size greater than ϵw (that is, if `end-of-codes`(b) evaluates to zero), it applies the DECODE-GAMMA algorithm as a subroutine.

```
SUM-GAMMA-FAST(B, v)
  ℓ ← 0
  t ← 0
  do
    b ← B[ℓ ... ℓ + εw − 1]
    s ← sum-of-codes[b]
    e ← end-of-codes[b]
    if (s = 0) then
      (s, e) ← DECODE-GAMMA(B[ℓ ... |B| − 1])
    if (t + s ≥ v) then
      (s, e) ← (sum-up-to-(v − t)[b], end-up-to-(v − t)[b])
      return (t + s, ℓ + e)
    t ← t + s
    ℓ ← ℓ + e
  loop while(ℓ < |B|)
  return (t, ℓ)
```

Figure 2.2: Pseudocode for the SUM-GAMMA-FAST algorithm.

The SUM-GAMMA-FAST algorithm always decodes at least ϵw bits of code per two lookup steps. (The first lookup step decodes all but the last code in b, and the second lookup step decodes at least the last code.) Thus the time needed to decode $|s|$ bits using SUM-GAMMA-FAST is $O(\frac{|s|}{\epsilon w})$.

The algorithm decodes chunks of bits until the sum of all gamma codes decoded reaches or exceeds v. At this point the algorithm requires an array of additional tables `sum-up-to-`v and `end-up-to-`v. These give the sum and ending bit-position, respectively, of the maximal number of (consecutive) gamma codes in b whose sum is less than v. The algorithm uses separate tables for each value of v from 2 to $2^{\epsilon w}$. Using the appropriate tables the algorithm computes and returns the result in $O(1)$ time. Pseudocode for this algorithm is shown in Figure 2.2.

It remains to bound the space used by these lookup tables. Each of the lookup tables described above stores, for each of $2^{\epsilon w}$ entries, a value between 0 and $2^{\epsilon w}$. There are $O(2^{\epsilon w})$ tables allocated, so the total cost is $O(2^{2\epsilon w}\epsilon w)$ bits. As in Section 2.3, for applications in which the largest values stored are $O(|C|)$, this expression can be made a low order term while still running in constant time.

2.6 Rank and Select

It is quite straightforward to store a group of prefix-free codes if access time is not a concern. The codes can be concatenated into one large bitstring B; since the codes are prefix-free, they can be uniquely decoded one-by-one. However, for some applications it is necessary to access individual codes—in particular, to access the i^{th} code stored in $O(1)$ time.

This problem has been studied extensively [68, 90] and is usually called the SELECT problem. Given a bitstring S of size n bits, SELECT(S, i) is a query which returns the position of the i^{th} 1 in S. These queries can be resolved using a *select data structure* created by preprocessing S. Munro [90] presented an algorithm which used $O(1)$ time to answer SELECT queries using an auxiliary data structure of $o(n)$ bits.

The SELECT data structure permits access to individual codes as follows. Let the bitstring S have size equal to B. If any code i begins at position j in B, then let $S[j] = 1$. All other locations in S are set to 0. The location of the i^{th} code in B is given by $\text{SELECT}(S, i)$.

The inverse of the SELECT operation is called RANK. Given a bitstring S of size n bits, $\text{RANK}(j)$ returns the number of 1s that occur before position j in S. Jacobson [68] showed that RANK queries can be resolved in $O(1)$ time using an $o(n)$-bit RANK data structure.

In practice we find that the $o(n)$-bit data structures have high associated constants—and, regardless, the need to maintain the n-bit bitstring S makes the $o(n)$ bound on the auxiliary data structure moot. For our experiments we generally use $O(n)$-bit data structures of our own devising.

2.7 Graph Separators

Let S be a class of graphs that is closed under the subgraph relation. S is defined to satisfy a $f(n)$-*separator theorem* if there are constants $\alpha < 1$ and $\beta > 0$ such that every graph in S with n vertices has a cut set with at most $\beta f(n)$ vertices that separates the graph into components with at most αn vertices each [81].

In this thesis we are particularly interested in the compression of classes of graphs for which $f(n)$ is n^c for some $c < 1$. One such class is the class of planar graphs, which satisfies a $n^{\frac{1}{2}}$-separator theorem. The results will apply to other classes as well: for example, Miller et al. [85] demonstrated that every well-shaped mesh in $\mathbb{R}^d$ has a separator of size $O(n^{1-1/d})$. We define a graph to be *separable* if it is a member of a class that satisfies an n^c-separator theorem.

A class of graphs has *bounded density* if every n-vertex member has $O(n)$ edges. Lipton, Rose, and Tarjan [80] prove that any class of graphs that satisfies a $n/(\log n)^{1+\epsilon}$-separator theorem with $\epsilon > 0$ has bounded density. Hence separable graphs have bounded density.

Another type of graph separator is an *edge separator*. A class of graphs S satisfies a $f(n)$-*edge separator theorem* if there are constants $\alpha < 1$ and $\beta > 0$ such that every graph in S with n vertices has a set of at most $\beta f(n)$ edges whose removal separates the graph into components with at most αn vertices each. Edge separators are less general than vertex separators: every graph with an edge separator of size s also has a vertex separator of size at most s, but no similar bounds hold for the converse. This thesis will mostly deal with edge separators, but will show theoretical results for graphs with vertex separators.

For theoretical purposes we will assume the existence of a graph separator algorithm that returns a separator within the $O(n^c)$ bound. For experimental purposes we find that the Metis [71] heuristic graph separator library works well.

Chapter 3

Compact Dictionaries With Variable-Length Keys and Data

3.1 Introduction

The dictionary problem is to maintain an n-element set of keys s_i with associated data ("satellite data") t_i.[1] 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 bitstrings. Our main motivation is to use such dictionaries as building blocks for various other applications. As an example application we present a representation of cardinal trees with nodes of varying cardinality. Other applications of our variable-bit array and dictionary structure appear in Sections 4.2, 5.3, 5.4, and 7.3.

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 bitstrings s_i and t_i.

There has been significant recent work involving data structures that use near optimal space while supporting fast access [68, 91, 40, 29, 96, 57, 102, 51, 15, 103]. The dictionary problem in particular has been well-studied in the case of fixed-length keys. The information-theoretic lower bound for representing n elements from a universe U is $B = \log\lceil\binom{|U|}{n}\rceil = n(\log |U| - \log n) + O(n)$. Cleary [42] 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* [74], 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 [29] described 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 [96] 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 [103] 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 $B + n|t| +$

[1]This chapter is based on work with Guy Blelloch [17].

$o(B + n|t|)$ bits. None of this considers variable-bit keys or data.

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 dictionary supports 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 bitstring. 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 bitstrings were the same length then this would be trivial.

Cardinal Trees. As an example application we present a representation of cardinal trees (aka tries) in which each node can have a different cardinality. Queries can request the k^{th} child, or the parent of any vertex. We can attach satellite bitstrings 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 [10, 102] to varying cardinality. We do not match the optimal constant in the first order term.

Experimentation. We present experimental results for our dictionary structure on a trace of operations performed by a simplicial meshing algorithm [14]. We analyze the structure's performance using difference codes that are optimized for speed and for compression. We compare the structure to a naive hashtable; the hashtable is slightly more time-efficient than our structure but uses a factor of $6.5 - 8.5$ more space.

3.2 Arrays

We define a *variable-bit-length array structure* to be one that maintains bitstrings $a_1 \ldots a_n$, supporting `update` and `lookup` operations. (An `update` changes one of the bitstrings, potentially changing its length as well as the data. A `lookup` returns one of the bitstrings to the user.) Our array representation supports strings of size $1 \leq |a_i| \leq w$; it performs lookups in $O(1)$ time and updates in $O(1)$ expected amortized time. Strings of size more than w must be allocated separately, and w-bit pointers to them can be stored in our structure. The memory allocation system used for this must be capable of allocating or freeing $|s|$ bits of memory in time $O(|s|/w)$, and may use $O(|s|)$ space to keep track of each allocation. It is well known how to do this (*e.g.*, [8]).

Overview. We begin with an overview of our array structure. We partition the strings a_i into *blocks* of contiguous elements, containing on average $\Theta(w)$ bits of data per block. We maintain the blocks in a conventional data structure (such as a hashtable) using $O(w)$ bits per block. We keep an auxiliary bit-array that allows us to determine which block contains a given element in constant time. We keep auxiliary data with each block that allows us to locate any element within the block in constant time. Using these operations we can support `update` and `lookup` in constant time.

We now present the structure in more detail.

Our structure consists of two parts: a set of blocks B and an index I. The bitstrings 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 bitstrings (in increasing order) a_i, a_{i+1}, …, a_{i+k}. The block stores the concatenation of the strings $b_i = a_i a_{i+1} \dots a_{i+k}$, together with information from which the start location of each string can be found. It suffices to store a second bitstring b'_i such that b'_i contains a 1 at position j if and only if some bitstring a_k ends at position j in b_i.

A block B_i consists of the pair (b_i, b'_i). We define the size of a block to be $|b_i| = \sum_{j=0}^{k} |a_{i+j}|$. 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 $A[1 \dots n]$ and a hashtable H. (In practice we use an optimized, space efficient variant of a hashtable.) 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 [97] or the dynamic version of the FKS perfect hashing scheme [47] have these properties. If expected rather than worst-case lookup bounds are acceptable, then a standard implementation of chained hashing will work as well.

Bit-Select and Bit-Rank. We assume that the processor supports two special operations, BIT-SELECT and BIT-RANK, defined as follows. Given a bitstring s of length w bits, BIT-SELECT(s, i) returns the least position j such that there are i ones in the range $s[0] \dots s[j]$. BIT-RANK(s, j) returns the number of ones in the range $s[0] \dots s[j]$. These operations mimic the function of the `rank` and `select` data structures, as described in Section 2.6.

If the processor does not support these operations, we can implement them using constant-time table-lookup, similar to the table-lookup described in Section 2.5.

Operations. We begin by observing that no block can contain more than w bitstrings (since blocks have maximum size w and each bitstring contains 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 $s = A[k - w] \dots A[k-1]$ and compute BIT-SELECT$(s,$ BIT-RANK$(s, w-1))$. To find the nearest one on the right, we let $s = A[k+1] \dots A[k+w]$ and compute BIT-SELECT$(s, 1)$. 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 of k. 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 BIT-SELECT$(b'_i, k - i + 1)$.

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 since shifting and copying 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 bitstrings 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 3.2.1 *Our variable-bit-length array representation can store bitstrings of length* $1 \leq a_i \leq w$ *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.*

3.3 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 bitstrings $s_1 \ldots s_n$, for $1 \leq |s_i| \leq w + \log n$. (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 $O(m)$ bits 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 [47] of the FKS perfect hashing scheme [52] that improves the query time to $O(1)$ worst-case time.

Quotienting. For representing sets of fixed length elements a space bound is already known [96]: 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 bitstrings 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 [74, Section 6.4, exercise 13] and has been used in several contexts [42, 29, 103, 51]. Previous quotienting schemes, however, were not concerned with variable length keys, and so the s'' strings they produce do not have the length properties we need.

In this chapter we develop our own variable-bit-length quotienting scheme. For this 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 bitstrings s_i as integers. Accordingly we reinterpret, when necessary, each bitstring 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 $h \in H$, $\Pr(h(x_1) = h(x_2)) \leq k/2^q$ [32], and is *k-pairwise independent* if for random $h \in H$, $\Pr(h(x_1) = y_1 \wedge h(x_2) = y_2) \leq k/2^{2q}$ for any $x_1 \neq x_2$ 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 $h' : \{0,1\}^{w+q+1} \rightarrow \{0,1\}^q$. 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 $h''(x_i)$.

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:

$$\begin{array}{cc} \overline{x}_i = x_i \operatorname{div} 2^q & \underline{x}_i = x_i \bmod 2^q \\ h''(x_i) = \overline{x}_i & h'(x_i) = (h_0(\overline{x}_i)) \oplus \underline{x}_i \end{array}$$

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

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

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 $x_1 \neq x_2$, 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 $\Pr(h'(x_1) = h'(x_2)) \leq 2/2^{2q}$.

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 (as described in Section 2.3) 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.

The gamma code for the length of an element can be read in constant time with the use of a lookup table, as described in Section 2.3. The length of any element is $O(|C|)$ (where $|C|$ is the total size of the data structure), so using a lookup table word of size $(\log |C|)/2$ makes the table size $O(2^{(\log |C|)/2} \log |C|) = o(|C|)$ while still allowing $O(1)$ time decoding.

Thus it takes $O(1)$ time to decode any element in the bucket (reading the gamma code for the length, then extracting the element using 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 bitstring stored for each s_i has size $O(\max(|s_i| - q, 1))$; the bitstring 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 items 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. [47].

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 [47]: 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 (similar to that described in Section 2.5). 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 code 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 simulating $w = \log |C|$ and choosing $\epsilon = 1/4$, the table usage can be made a lower order term while still running in $O(1)$ time.

This gives us the following theorem:

Theorem 3.3.1 *Our variable-bit-length dictionary representation can store bitstrings of any size using $O(m)$ bits where $m = \sum(\max(|s_i| - \log n, 1) + t_i)$ while allowing updates in $O(1)$ amortized expected time and accesses in $O(1)$ worst-case time.*

3.4 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.

We consider these operations "semidynamic": the time bounds will hold for any sequence of operations, but the compression achieved will depend on the labeling of the vertices. If the tree changes shape significantly, the vertices may need to be relabeled to maintain the space bounds.

We begin with a dictionary-based representation for cardinal trees. For each vertex v we store a dictionary entry $((v), (c(v), p(v) - v))$—that is, the dictionary maps v to the pair $(c(v), p(v) - v)$. (To encode a pair of values, we gamma code each value and concatenate them to form a bitstring.) For each child of v we store an entry $((v, i), (v[i] - v))$. Given this representation we can support cardinality queries, parent queries, and child queries.

Lemma 3.4.1 *The representation we describe supports parent and child queries in $O(1)$ time and insertion and deletion of leaves in $O(1)$ expected amortized time. With a variable-bit-length dictionary the space used is $O(m)$ bits where $m = \sum_{v \in V}(\log c(p(v)) + \log |p(v) - v|)$.*

Proof. The space usage of our variable-bit-length dictionary structure is $m = \sum_{(s,t)\in D}(|t| + \max(1, |s| - \log |D|))$. The first type of dictionary entry we store is $((v, i), (v[i] - v))$. The cost of storing v is absorbed by the $\log |D|$. The cost of storing i for each vertex is the $\log c(p(v))$ above. The cost of storing $(v[i] - v)$ for each child is the same as the cost of storing $p(v) - v$ for each vertex, so it is handled by the $\log |p(v) - v|$ given above.

The second type of entry we store is $((v), (c(v), p(v) - v))$. As before, the v is absorbed by the $\log |D|$. The cost of storing $p(v) - v$ for each vertex is the $\log |p(v) - v|$ given above. The cost of $c(v)$ is charged to the first child of the vertex if $c(v) > 0$; otherwise the cost is $O(1)$ bits and is charged to the $\log |p(v) - v|$.

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 traversal of T_s. We call such a labeling a *tree-separator labeling*.

Lemma 3.4.2 *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 3.4.1 *Cardinal trees with a tree-separator labeling can be stored in $O(m)$ bits, where $m = \sum_{v\in V}(1 + \log(1 + c(p(v))))$.*

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 3.4.2 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 3.4.1 this gives the required bounds.

3.5 Experimentation

To understand the time- and space-efficiency of our dictionary structure we tested it using a real-world application: an algorithm to perform 3D Delaunay tetrahedralization (described more fully in Chapter 7). For that structure it was necessary to map edges (v_a, v_b) to blocks of data. Edges could be inserted or deleted, and the data could be updated. We used a variant of our dictionary structure to support these operations.

For our tests we captured traces of the updates and lookups involved in constructing a mesh of between 2^{15} and 2^{20} vertices. We used these traces to test our variable-bit-length dictionary structure implemented

# vtxs			VarArray(Nibble)		VarArray(Byte)		Hashtable	
	Updates	Lookups	Time	Space	Time	Space	Time	Space
2^{15}	1019320	1498357	0.795	11.12	0.632	14.46	0.382	96.17
2^{16}	2043269	3006491	1.59	11.10	1.27	14.56	0.883	96.23
2^{17}	4108355	6052525	3.34	11.32	2.63	14.65	2.07	96.40
2^{18}	8267102	12180810	6.96	11.43	5.54	14.82	4.56	96.70
2^{19}	16590922	24442256	14.3	11.34	11.5	14.83	12.6	96.81
2^{20}	33217081	48919922	29.6	11.56	23.7	14.87	22.3	96.71

Table 3.1: Time (in seconds) and space (in bytes per vertex) to store and update data for each edge in a tetrahedral Delaunay mesh.

using two coding techniques: the *byte-aligned* and *nibble-aligned* codes, as described in Section 2.3. (The byte-aligned code is optimized for good time performance, while the nibble-aligned code is preferred for a high compression ratio.) For each test we ran all of the lookups from the trace, using one byte of data (rather than the larger amount of data from the original application). We compared the results to those for a standard bucketed hash table. The bucketed hashtable is initially faster but loses its advantage for large sizes; we suspect this is because it requires too much memory to fit in the cache. The results from our experiments are shown in Table 3.1; further implementation details are given below.

Dictionary Structure. The data structure we use to represent this information is a modification of our variable-bit-length dictionary structure. Every edge (v_a, v_b) is mapped to a bucket from an array of $|V|$ buckets. We use quotienting to save $\log |V|$ bits from the cost of storing each key, as described in Section 3.3: we let

$$K = v_b - v_a \qquad\qquad B = v_a \oplus h_0(K)$$

and store key K in bucket number B. For the base hash function h_0 we use a random number table of size 256: $h_0(K) = \texttt{table}[K \,\&\, 255]$.

Additionally, we note that our 3D meshing algorithm shows considerable locality of access, in that frequently it performs many accesses to vertices with similar labels. Accordingly we restrict the hash function h_0 to a smaller range, $[0..G-1]$. This effectively partitions the buckets in the array into groups of size G, to be determined later. We keep some information associated with each group (to be discussed later).

The description of our dictionary structure in Section 3.3 specifies that the buckets should be elements of a variable-bit-length array structure, so that underfull buckets should not cause a space penalty. The variable-bit-length array structure has a significant constant overhead, though; for our application we instead keep the buckets sufficiently full that underfull buckets do not cause problems.

Initially each bucket is allocated a fixed number of bytes. If more space is required, the bucket is allocated additional blocks of memory from a secondary pool of blocks, as required. The last byte in a block stores a one-byte pointer to the next block, if there is one. (This makes use of a hashing trick—see Section 5.5 for details.) To preserve memory locality, the secondary pool of blocks and allocation structures are kept separately for each bucket group. The space cost of the allocation structure is amortized over the cost of the buckets in the group.

The original meshing application contains a great deal of data per bucket (in the form of vertex lists for each data item); accordingly it uses a bucket group size $G = 16$. This application uses less data per bucket, so we amortize the allocation structure over a larger group size $G = 64$.

The byte-aligned code is less space-efficient (but more time-efficient) than the nibble code. Accordingly we allocate more space for the dictionary using the byte-aligned code.

After some experimentation we chose to allocate 10 bytes for each bucket initially when using the byte-aligned code, and to allocate additional memory in blocks of 4 bytes. We allocate 0.7 secondary blocks for each bucket, and can expand the secondary block pool if necessary.

The nibble code is more space-efficient than the byte code, so the dictionary does not require as much space when using it. Using the nibble code we initially allocate 7 bytes per bucket rather than 10, and 0.65 secondary blocks per bucket rather than 0.7.

In each case the sizes are chosen such that about 25% of bucket groups require additional blocks to be allocated from the secondary block pool.

Hashtable. We compare our structure to a naive hashtable. Each (`key`, `data`) pair in the structure uses one listnode containing a 4-byte word each for v_a, v_b, and the data, and an 8-byte pointer to the next node. On our 64-bit architecture the listnodes are rounded up to the nearest word size, making them 24 bytes each. (On a 32-bit architecture the listnodes would be only 16 bytes each.) Each bucket uses one 8-byte pointer as well. As in the variable-bit-length dictionary structure, we keep $|V|$ buckets in the hashtable.

3.6 Discussion

We have presented two data structures, the variable-bit-length array and dictionary structure, which can serve as useful building blocks for other structures. The structures have strong theoretical bounds: $O(1)$ lookup and amortized expected $O(1)$ update operations. Our experimentation here, and further experimentation in Chapters 5, 7, and 8, shows that (variants of) the structures are useful in practice as well.

For practical applications we modify the structure as discussed in Section 3.5 above. We divide the structure into groups, each with its own subhashtable, to improve locality of access. For the variable-bit-length dictionary structure we do not use an underlying variable-bit-length array structure; instead we choose settings that keep the buckets of the dictionary close to full. Finally, we use our own memory allocator to assign blocks to store difference codes.

Further details of our implementation of the dictionary structure can be found in Chapter 7.

Chapter 4

Compact Representations of Ordered Sets

4.1 Introduction

In this chapter we describe a data structure to compactly represent an ordered set $S = \{s_1, s_2, \ldots, s_n\}, s_i < s_{i+1}$, from a universe $U = \{0, \ldots, m-1\}$.[1] This data structure supports a wide variety of operations and can operate in a purely functional setting [69]. (In a purely functional setting data cannot be overwritten. This means that all data is fully persistent.)

This data structure has many applications, especially in the design of search engines. Memory considerations are a serious concern for search engines. Some web search engines index billions of 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, and reports them to the user. It may be desirable to maintain the documents ordered, for example, by a ranking of the pages based on importance [95]. Using difference coding (as described in Section 2.4) these lists can be compressed into an array of bits using 5 or 6 bits per edge [136, 88, 12], but such representations are not well suited for merging lists of different sizes.

The data structure we describe can be used to represent a posting list from a search engine. The structure supports dynamic operations including set union and intersection while maintaining the data within a constant factor of the information-theoretic bound. Also, since it operates in a purely functional setting, the search engine can perform set operations on posting lists without spending time and memory to make copies of the sets.

There has been significant research on compact representation of sets taken from U. An information-theoretic bound shows that representing a set of size n (for $n \le \frac{m}{2}$) requires $\Omega(\log \binom{m}{n}) = \Omega(n \log \frac{m+n}{n})$ bits. Brodnik and Munro [29] demonstrate a structure that is optimal in the high-order term of its space usage, and supports lookup in $O(1)$ worst-case time and insert and delete in $O(1)$ expected amortized time.

[1]This chapter is based on work with Guy Blelloch [13].

Pagh [96] simplifies the structure and improves the space bounds slightly. These structures, however, are based on hashing and do not support ordered access to the data: for example, they support searching for a precise key, but not searching for the next key greater (or less) than the search key. Pagh's structure does support the `Rank` operation (as described in Section 2.6) but only statically, *i.e.*, without allowing insertions and deletions. As with our work they assume the unit cost RAM model with word size $\Omega(\log |U|)$.

The set union and intersection problems are directly related to the list merging problem, which has received significant study. Carlsson, Levcopoulos, and Petersson [31] considered a block metric $k = Block(S_1, S_2)$ which represents the minimum number of blocks that two ordered lists S_1, S_2 need to be broken into before being recombined into one ordered list. Using this metric, they show an information-theoretic lower bound of $\Omega(k \log \frac{|S_1|+|S_2|}{k})$ on the time complexity of list merging in the comparison model.

Moffat, Petersson, and Wormald [86] show that the list merging problem can be solved in $O(k \log \frac{|S_1|+|S_2|}{k})$ time by any structure that supports *fast split and join* operations. 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 in 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 list merging algorithm requires only that the split and join operations run in $O(\log |S_1|)$ time.

In this chapter we present two representations for ordered sets. The first, in Section 4.2, is a simple representation using the variable-bit-length dictionary from Section 3.3. It is simple to describe but does not support the full range of operations that we need for a posting-list data structure.

Our second representation, described in Section 4.4 and Section 4.5, is a compression technique which improves the space efficiency of structures for ordered sets taken from U. Given a base structure supporting a few basic operations, our technique can improve the structure's space bound to $O(n \log \frac{m+n}{n})$ bits. Our technique allows a wide range of operations as long as they are supported by the base structure.

Section 4.6 gives experimental results for the second representation. To show the versatility of the compression technique, we applied it to two separate data structures: red-black trees [60] and functional treaps [6].

4.2 Representation With Dictionaries

Here we describe a representation for ordered sets based on our variable-bit-length dictionary from Section 3.3.

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 $\min\{k_2 \in S | k_2 > k_1\}$; *fingerSearch* 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 $l = |\{k \in S | k_1 \leq k \leq k_2\}|$.

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 [20], 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 4.2.1 *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 $\sum_{v\in T}(\log|p(v)-v|) \in O(n\log(m/n))$.*

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 $|L_i| \le \alpha|L_{i+1}|, 1 \le i < l, \alpha > 1$, 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 $M(S) = \sum_{j\in S}\log(\textit{next}(j,S)-j)$. Since logarithms are concave, the sum is maximized when the elements are evenly spaced. Thus $M(S) \le |S|\log(m/|S|)$. For any proper level covering L of a set S this gives:

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

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 the 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 4.2.1 *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 4.2.1. This gives the desired bounds.

The representation described here is powerful, but it supports only the operations allowed by a red-black tree. (Also, it cannot be easily made purely functional.) The next representation we describe will support a greater range of operations.

4.3 Supported Operations

Our ordered-set structures can support 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 $d = |\{s \in S \mid y < s < x \ \vee \ x < s < y\}|$.
- `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`: This operation assumes that a weight $w(x)$ is provided with every element x as it is inserted. Given an element y, this operation finds $r = \sum_{x\in S, x<y} w(x)$. In the unweighted variant, all weights are considered to be 1.
- `(Weighted) Select`: This operation assumes that a weight $w(x)$ is provided with every element x as it is inserted. Given r, this operation finds the greatest y such that $\sum_{x\in S, x<y} w(x) \leq r$. It returns both y and the associated sum. In the unweighted variant, all weights are considered to be 1.

Given any ordered dictionary structure D using $O(n \log m)$ bits to store n values from $U = \{0, \ldots, m-1\}$, the blocking technique we demonstrate 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. Our 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. Our technique also makes use of a lookup table of size $O(m^{2\alpha} \log m)$ for a parameter $\alpha > 0$. (For most of the applications in this thesis we could use a table of size $O(2^{\epsilon w})$ entries; we could simulate $w = \log |C|$ and choose ϵ to make the table size a low-order term. Here, though, we do not assume m is related to $|C|$. We must decode gamma codes of size $\log m$ in constant time, so we must explicitly count the cost $O(m^{2\alpha} \log^2 m)$ against our space usage.)

Our data structure works as follows. Elements in the structure are *difference coded* (as described in Section 2.4) and stored in fixed-length blocks of size $\Theta(\log m)$. The first element of every block is kept uncompressed. The blocks are kept in a dictionary structure (with the first element as the key). The data structure needs to know nothing about the actual implementation of the dictionary. A query consists of first searching for the appropriate block in the dictionary, and then searching within that block. We provide a framework for dynamically updating blocks as inserts and deletes are made to ensure that no block becomes too full or too empty. For example, inserting into a block might overflow the block. This requires it to be split and a new block to be inserted into the dictionary. The operations we use on blocks correspond almost directly to the operations on the tree as a whole. We use table-lookup to implement the block operations efficiently.

Our structure can support a wide range of operations, depending on the operations the dictionary D supports. In all cases the cost of our operations is $O(1)$ instructions and $O(1)$ operations on D.

If the input structure D supports the `Search`$^-$, `Search`$^+$, `Insert`, and `Delete` operations, then our structure supports those operations.

If D supports `FingerSearch` and supports `Insert` and `Delete` at a finger, then our structure supports those operations.

If D supports `First`, `Last`, `Split`, and `Join`, then our structure supports those operations. If the bounds for `Split` and `Join` on D are $O(\log \min(|D_1|, |D_2|))$, then our structure meets these bounds (despite the $O(1)$ calls to other operations).

If D supports `WeightedRank`, then our structure supports `Rank`. If D supports `WeightedSelect`, then our structure supports `Select`. Our algorithms need the weighted versions so that they can use the number of entries in a block as the weight.

The catenable-list structure of Kaplan and Tarjan [69] can be adapted to support all of these operations. The time bounds (all worst-case) are $O(\log n)$ for `Search`$^-$, `Search`$^+$, `Insert`, and `Delete`; $O(\log d)$ for `FingerSearch`, where d is as defined above; $O(1)$ for `First` and `Last`; and $O(\log \min(|D_1|, |D_2|))$ for `Split` and `Join`. Our structure meets the same bounds. As another example, our representation based on a simpler dictionary structure based on Treaps [108] supports all these operations in the time listed in the expected case. Both of these can be made purely functional. As a third example, our representation using a skip-list dictionary structure [100] supports these operations in the same time bounds (expected case) but is not purely functional.

$\{306, 309, 312, 314, 315, 319\}$

306	3	3	2	1	4

0100110010	011	011	010	1	00100

Figure 4.1: The encoding of a block of size 15. In this case the universe has size 1024, so the head is encoded with 10 bits.

4.4 Block structure

Our representation consists of two structures, nested using a form of structural bootstrapping [30]. The base structure is the *block*. In this section we describe our block structure and the operations supported on blocks. Then, in Section 4.5, we describe how blocks are kept in an ordered-dictionary structure to support efficient operations.

The block structure, the given dictionary structure and our combined structure all implement the same operations except that the block structure has an additional `BMidSplit` operation, and only the given dictionary supports the weighted versions of `Rank` and `Select`. For clarity, we refer to operations on blocks with the prefix B (*e.g.*, `BSplit`), operations on the given dictionary structure with the prefix D (*e.g.*, `DSplit`), and operations on our combined structure with no prefix.

Block encoding. A *block* B_i is an encoding of a series of values (in increasing order) $v_1, v_2, \ldots, v_k$. The block is encoded as a $\log m$-bit representation of v_1 (called the "head") followed by difference codes (as in Section 2.4) for $v_2 - v_1, v_3 - v_2, \ldots, v_k - v_{k-1}$. (See Figure 4.1 for an example.) We say that the *size* of a block `size`(B) is the total length of the difference codes contained in that block. In particular we are interested in blocks of size $O(\log m)$ bits.

It is important for our time bounds that the operations on blocks are fast—they cannot take time proportional to the number of values in the block. We make use of table lookup for fast decoding, as described in Section 2.5, using a table word size of $\alpha \log m$ for some parameter α. Since blocks have size $O(\log m)$, the `sum-gamma-fast` algorithm from that section allows access to any value in the block in $O(\frac{\log m}{\alpha \log m}) = O(1/\alpha)$ time. The cost of the lookup tables for `sum-gamma-fast` is $O(m^{2\alpha} \log m)$ bits.

We use M to denote the maximum possible length of a difference code. In the case of gamma codes, $M = 2\lfloor \log m \rfloor + 1$ bits. Throughout Sections 4.4 and 4.5 we will assume the use of gamma codes.

We define the following operations on blocks. All operations require constant time assuming constant α and that the blocks have size $O(\log m)$. Some operations increase the size of the blocks; we describe in Section 4.5 how the block sizes are bounded.

`BSearch`$^-$ (`BSearch`$^+$): Given a value v and a block B, these operations return the greatest (least) value in B that is less than or equal (greater than or equal) to v. This is an application of the `sum-gamma-fast` routine.

`BInsert`: Given a value v and a block B, this operation inserts v into B. If v is less than the head for B, then our algorithm encodes that head by its difference from v and adds that code to the block. Otherwise, our algorithm searches B for the value v_j that should precede v. The gamma code for $v_{j+1} - v_j$ is deleted and replaced with the gamma codes for $v - v_j$ and $v_{j+1} - v$. (Some shift operations may be needed to make room for the new codes. Since each shift affects $O(\log m)$ bits, this requires constant time.)

`BDelete`: Given a block B and a value v_j contained in B, this operation deletes v_j from B. If v_j is the head for B, then its successor is decoded and made into the new head for B. Otherwise, our algorithm searches B for v_j. It deletes the gamma codes for $v_j - v_{j-1}$ and for $v_{j+1} - v_j$ and replaces them with the gamma code for $v_{j+1} - v_{j-1}$. (Part of the block may need to be shifted. As in the Insert case, this requires a constant number of shifts.)

`BMidSplit`: Given a block B of size b bits (where $b > 2M$), this operation splits off a new block B' such that B and B' each have size at least $b/2 - M$. It searches B for the first code c that starts after position $b/2 - M$ (using the second array stored with each table entry). Then c is decoded and made into the head for B'. The codes after c are placed in B', and c and its successors are deleted from B. B now contains at most $b/2$ bits of codes, and c contained at most M bits, so B' contains at least $b/2 - M$ bits. This takes constant time since codes can be copied $\Omega(\log m)$ bits at a time.

`BFirst`: Given a block B, this operation returns the head for B.

`BLast`: Given a block B, this operation scans to the end of B and returns the final value.

`BSplit`: Given a block B and a value v, this operation splits a new block B' off of B such that all values in B' are greater than v and all values in B are less than v. This is the same as `BMidSplit` except that c is chosen by a search rather than by its position in B. This operation returns v if it was in B.

`BJoin`: The join operation takes two blocks B and B' such that all values in B' are greater than the greatest value from B. It concatenates B' onto B. To do this it first finds the greatest value v in B. It represents the head v' from B' with a gamma code for $v' - v$ and appends this code to the end of B. It appends the remaining codes from B' to B. This takes constant time since codes can be copied $\Omega(\log m)$ bits at a time.

`BRank`: To support the `BRank` operation the `sum-gamma-fast` lookup tables need to be augmented: along with the sum of the gamma codes in a chunk, the table needs to contain information on the number of codes decoded. To find the rank of an element v within a block B, our algorithm searches for the element while keeping track of the number of elements in each chunk skipped over.

`BSelect`: To support the `BSelect` operation the `sum-gamma-fast` lookup tables need to be augmented: in addition to the information needed for `BRank`, each chunk needs to have an array containing the decoded values. (The table needed for this has m^α entries of $(\alpha \log m) 2^{\alpha \log m}$ bits each; the total is $O(m^{2\alpha} \log m)$ bits, which does not alter the tables' asymptotic space complexity.) To find the element with a given rank, our algorithm searches for the chunk containing that element, then accesses the appropriate index of the array.

4.5 Representation

To represent an ordered set $S = \{s_1, s_2, \ldots, s_n\}$, $s_i < s_{i+1}$, our approach maintains S as a set of blocks B_i where $B_i = \{s_{b_i}, s_{b_i+1}, \ldots, s_{b_{i+1}-1}\}$. The values $b_1 \ldots b_k$ are maintained such that the size of each block is between M and $4M$. The first block and the last block are permitted to be smaller than M. (Recall that $M = 2\lfloor \log m \rfloor + 1$ is the maximum possible length of a gamma code.) This property is maintained throughout all operations performed on S.

Lemma 4.5.1 *Given any set S from $U = \{0, \ldots, m-1\}$, let $|S| = n$. Given any assignment of b_i such that $\forall B_i, M \leq \mathtt{size}(B_i) \leq 4M$, the total space used for the blocks is $O(n \log \frac{n+m}{n})$.*

Proof. We begin by bounding the space used for the gamma codes. The cost to gamma code the differences between every pair of consecutive elements in S is

$$\sum_{i=2}^{n}(2\lfloor \log(s_i - s_{i-1}) \rfloor + 1).$$

This sum is maximized when the values are evenly spaced in the interval $1 \ldots m$; at that point the sum is $\sum_{i=2}^{n}(2 \log \frac{m}{n} + 1)$, which is $O(n \log \frac{m}{n} + n) = O(n \log \frac{m+n}{n})$.

The gamma codes contained in the blocks are a subset of the ones considered above (since the head of each block is not gamma coded). For every $\log m$ bits used by a head there are at least M bits used by gamma codes; since $M > 2 \log m$ the amount of additional space used by heads is at most half that used by gamma codes.

The blocks B_i are maintained in an ordered-dictionary structure D. The key for each block is its head. We refer to operations on D with a prefix D to differentiate them from operations on blocks and from the interface to our representation as a whole. D may use $O(\log m)$ bits to store each value. Since each value stored in D contains $\Theta(\log m)$ bits already, this increases our space bound by at most a constant factor. Our representation, as a whole, supports the following operations. They are not described as functional but can easily be made so: rather than change a block, our algorithm could delete it from the structure, copy it, modify the copy, and reinsert it into the structure.

`Search`$^-$: First, our algorithm calls `DSearch`$^-(k)$, returning the greatest block B with head $k' \leq k$. If $k' = k$, return k'. Otherwise, call `BSearch`$^-(k)$ on B and return the result.

`Search`$^+$: First, our algorithm calls `DSearch`$^-(k)$, returning the greatest block B with head $k' \leq k$. If $k' = k$, return k'. Otherwise, call `BSearch`$^+(k)$ on B. If this produces a value, return that value; otherwise, call `DSearch`$^+(k+1)$ and return the head of the result.

`Insert`: First, our algorithm calls `DSearch`$^-(k)$, returning the block B that should contain k. (If there is no block with head less than k, our algorithm uses `DSearch`$^+(k)$ to find a block instead.) Our algorithm then calls `BInsert`(k) on B. If `size`$(B) > 4M$, our algorithm calls `BMidSplit` on B and uses `DInsert` to insert the new block.

`Delete`: First, our algorithm calls `DSearch`$^-(k)$, returning the block B that contains the target element k. Then our algorithm calls `BDelete`(k) on B. If **size**$(B) < M$, our algorithm uses `DDelete` to delete B from D. It uses `DSearch`$^-$ to find the predecessor of B and `BJoin` to join the two blocks. This in turn may produce a block which is larger in size than $4M$, in which case a `BMidSplit` operation and a `DInsert` operation are needed as in the `Insert` case.

(Under rare circumstances, deleting a gamma-coded element from a block may cause it to grow in size by one bit. If this causes the block to exceed $4M$ in size, this is handled as in the `Insert` case.)

We define a "finger" to an element v to consist of a finger to the block B containing v in D.

`FingerSearch`: Our algorithm calls `DFingerSearch`(k) for the block B' which contains k. It then calls `BSearch`$^-(k)$ and returns the result.

`First`: Our algorithm calls `DFirst` and then `BFirst` and returns the result.

`Last`: Our algorithm calls `DLast` and then `BLast` and returns the result.

`Join`: Given two structures D_1 and D_2, our algorithm first checks the size of $B_1 = \mathtt{DLast}(D_1)$ and $B_2 = \mathtt{DFirst}(D_2)$. If $\mathtt{size}(B_1) < M$, our algorithm uses `DSplit` to remove B_1 and its predecessor, `BJoin` to join them, and `BMidSplit` if the resulting block is oversized. It uses `DJoin` to join the resulting block(s) back onto D_1. If $\mathtt{size}(B_2) < M$, our algorithm joins B_2 onto its successor using a similar method. Then our algorithm uses `DJoin` to join the two structures.

`Split`: Given an element k, our algorithm first calls `DSplit`(k), producing structures D_1 and D_2. If the split operation returns a block B, then our algorithm uses `BDelete` on B to delete the head, uses `DJoin` to join B to D_2, and returns (D_1, k, D_2). Otherwise, our algorithm calls `BSplit`(k) on the last block `DLast`(D_1). If this produces an additional block, this block is joined onto D_2.

`Rank`: The weighted rank of a block is defined to be the number of elements it contains. Our algorithm calls `DSearch`$^-(k)$ to find the block B that should contain k. It calls `DWeightedRank`(B) and `BRank`(k) and returns the sum.

`Select`: The size of a block is defined to be the number of elements it contains. Our algorithm uses `DWeightedSelect`(r) to find the block B containing the target, then uses `BSelect` with the appropriate offset on B to find the target.

Lemma 4.5.2 *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, our blocking technique produces a structure that uses $O(n \log \frac{n+m}{n})$ bits.*

1. *If D supports* `DSearch`$^-$, `DSearch`$^+$, `DInsert`, *and* `DDelete`, *the blocked set structure supports those operations using $O(1)$ instructions and $O(1)$ calls to operations of D.*
2. *If D supports* `DFingerSearch`, *the blocked set structure supports* `FingerSearch` *in $O(1)$ instructions and one call to* `DFingerSearch`. *If D supports* `DInsert` *and* `DDelete` *at a finger,*

```
union(S_1,S_2)
  if S_1 = null then
    return S_2
  if S_2 = null then
    return S_1
  (S_2A,v,S_2B) ← DSplit(S_2,DFirst(S_1))
  S_B ← union(S_2B,S_1)
  return DJoin(S_2A,S_B)
```

Figure 4.2: Pseudocode for a `union` operation.

then the blocked set structure supports those operations using $O(1)$ instructions and $O(1)$ calls to `DInsert` *and* `DDelete` *at a finger.*

3. *If D supports the* `DFirst`, `DLast`, `DSplit`, *and* `DJoin` *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* `DWeightedRank` *operation, then the blocked set structure supports the* `Rank` *operation in $O(1)$ instructions and one call to* `DWeightedRank`. *If D supports the* `DWeightedSelect` *operation, then the blocked set structure supports the* `Select` *operation using $O(1)$ instructions and one call to* `DWeightedSelect`.

The proof follows from the descriptions above.

4.6 Applications

By combining the `Split` and `Join` operations it is possible to implement efficient set union, intersection, and difference algorithms. An example implementation of `union` is shown in Figure 4.2. If `Split` and `Join` run in $O(\log |D_1|)$ time, then these set operation algorithms run in $O(k \log \frac{|D_1|+|D_2|}{k} + k)$ time, where k is the least possible number of blocks that we can break the two lists into before reforming them into one list. (This is the Block Metric of Carlsson et al. [31].)

As described in the introduction, the catenable ordered list structure of Kaplan and Tarjan [69] can be modified to support all of the operations described here in worst-case time. (To do this, we use `Split` as our search routine; to support `FingerSearch` we define a finger for k to be the result when the structure is split on k. To support weighted Rank and Select, we let each node in the structure store the weight of its subtree.) Thus our representation using their structure supports those operations in worst-case time using $O(n \log \frac{n+m}{n})$ bits. This structure may be somewhat unwieldy in practice, however.

If expected-case rather than worst-case bounds are acceptable, Treaps [108] are an efficient alternative. Treaps can be made to support the Split and Join operations by flipping the pointers along the left spine of the trees—each node along the left spine points to its parent instead of its left child. To split such a treap on a key k, an algorithm first travels up the left spine until it reaches a key greater than k, then splits the treap

$\|U\|$	$\|S\|$	Insert Times		Delete Times		Space Needed	
		Standard	Blocked	Standard	Blocked	Standard	Blocked
2^{20}	2^{10}	0.001	0.004	0.001	0.003	12	4.62
2^{20}	2^{12}	0.010	0.016	0.012	0.013	12	3.80
2^{20}	2^{14}	0.061	0.067	0.058	0.076	12	3.02
2^{20}	2^{16}	0.363	0.348	0.343	0.369	12	2.28
2^{20}	2^{18}	2.007	1.790	1.920	1.901	12	1.64
2^{25}	2^{10}	0.004	0.001	0.000	0.006	12	6.37
2^{25}	2^{12}	0.009	0.013	0.010	0.017	12	5.67
2^{25}	2^{14}	0.062	0.073	0.058	0.087	12	4.96
2^{25}	2^{16}	0.351	0.393	0.347	0.465	12	4.18
2^{25}	2^{18}	1.875	2.071	1.828	2.365	12	3.42
2^{30}	2^{10}	0.001	0.005	0.002	0.003	12	8.15
2^{30}	2^{12}	0.012	0.013	0.011	0.019	12	7.43
2^{30}	2^{14}	0.061	0.078	0.062	0.093	12	6.68
2^{30}	2^{16}	0.357	0.424	0.346	0.515	12	5.89
2^{30}	2^{18}	1.865	2.283	1.798	2.745	12	5.33

Table 4.1: Performance of a standard treap implementation versus our blocked treap implementation, averaged over ten runs. Time is in seconds; space is in bytes per value.

as normal. Seidel and Aragon showed that the expected path length of such a traversal is $O(\log |T_1|)$. By copying the path traversed this can be made purely functional.

4.7 Experimentation

We implemented our blocking technique in C using both treaps and red-black trees. Rather than the gamma code, we use the nibble code, as described in Section 2.3. We decode blocks nibble-by-nibble rather than with a lookup table as described above. For very large problems, using such a table might improve performance.

We use a maximum block size of 46 nibbles (23 bytes) and a minimum size of 16 nibbles (8 bytes). We use one byte to store the number of nibbles in the block, for a total of 24 bytes per block.

We combined our blocking structure with two separate tree structures. The first is our own (purely functional) implementation of treaps [6]. Priorities are generated using a hash function on the keys. Each treap node maintains an integer key, a left pointer, and a right pointer, for a total of 12 bytes per node. In our blocked structure each node also keeps a pointer to its block. Since each block is 24 bytes, the total space usage is 40 bytes per treap node.

The second tree structure is the implementation of red-black trees [60] provided by the RedHat Linux implementation of the C++ Standard Template Library [114]. We used the `map<int, unsigned char*>` template for our blocked structure and the `set<int>` template for the unblocked equivalent. A red-black

$\|U\|$	$\|S\|$	Insert Times		Delete Times		Space Needed	
		Standard	Blocked	Standard	Blocked	Standard	Blocked
2^{20}	2^{10}	0.001	0.002	0.000	0.003	20	5.49
2^{20}	2^{12}	0.004	0.006	0.003	0.014	20	4.55
2^{20}	2^{14}	0.013	0.033	0.023	0.054	20	3.62
2^{20}	2^{16}	0.064	0.136	0.100	0.230	20	2.74
2^{20}	2^{18}	0.357	0.559	0.538	0.972	20	1.97
2^{25}	2^{10}	0.001	0.003	0.000	0.000	20	7.66
2^{25}	2^{12}	0.004	0.008	0.004	0.015	20	6.80
2^{25}	2^{14}	0.012	0.037	0.022	0.056	20	5.96
2^{25}	2^{16}	0.064	0.152	0.098	0.247	20	5.02
2^{25}	2^{18}	0.384	0.634	0.583	1.066	20	4.10
2^{30}	2^{10}	0.000	0.003	0.002	0.003	20	9.79
2^{30}	2^{12}	0.003	0.010	0.005	0.015	20	8.91
2^{30}	2^{14}	0.013	0.040	0.020	0.060	20	8.01
2^{30}	2^{16}	0.066	0.170	0.100	0.262	20	7.08
2^{30}	2^{18}	0.385	0.714	0.589	1.143	20	6.39

Table 4.2: Performance of a standard red-black tree implementation versus our blocked red-black tree implementation, averaged over ten runs. Time is in seconds; space is in bytes per value.

tree node includes a key, three pointers (left, right, and parent), and a byte indicating the color of the node. Since a C compiler allocates memory to data structures in multiples of 4, this requires a total of 20 bytes per node for the unblocked implementation, and 48 bytes for our blocked implementation.

We ran our simulations on a 1GHz processor with 1GB of RAM.

For each of our tree structures we tested the time needed to insert and delete elements. We used universe sizes of 2^{20}, 2^{25}, and 2^{30}, with varying numbers of elements. Elements were chosen uniformly from U. All elements in the set were inserted, then deleted in the same order. We calculated the time needed for insertion and deletion and the space required by each implementation, and computed the average over ten runs.

Results for the treap implementations are shown in Table 4.1. Our blocked version uses considerably less space than the non-blocked version; the improvement is between a factor of 1.45 and 7.3, depending on the density of the set. The slowdown caused by blocking varies but is usually less than 50%. (In fact, sometimes the blocked variant runs faster. We suspect this is because of caching and memory issues.)

Results for the red-black tree implementations are shown in Table 4.2. Here the space improvement is between a factor of 2 and 10. However the slowdown is sometimes as much as 150%.

Note that the STL red-black tree implementation is significantly faster than our treap implementation. In part this is because our treap structure is purely functional (and thus persistent). The red-black tree structure is not persistent.

For our treap data structure we also implemented the serial merge algorithm described in Section 4.6. We computed the time needed to merge sets of varying sizes in a universe of size 2^{20}. Results are shown in

$\|A\|$	$\|B\|$	Union Time	
		Standard	Blocked
2^{14}	2^{10}	0.003	0.011
2^{14}	2^{12}	0.015	0.036
2^{14}	2^{14}	0.036	0.086
2^{16}	2^{10}	0.005	0.014
2^{16}	2^{12}	0.028	0.048
2^{16}	2^{14}	0.067	0.157
2^{16}	2^{16}	0.151	0.370
2^{18}	2^{10}	0.006	0.015
2^{18}	2^{12}	0.043	0.059
2^{18}	2^{14}	0.119	0.208
2^{18}	2^{16}	0.293	0.703
2^{18}	2^{18}	0.616	1.540

Table 4.3: Performance of our serial merge algorithm implemented using standard treaps and blocked treaps. All values are averaged over ten runs. The universe size is 2^{20}. Time is in seconds.

Figure 4.3. The slowdown caused by blocking was at most 150%.

Chapter 5

Compact Representations of Graphs

5.1 Introduction

We are interested in representing graphs compactly while supporting queries and updates efficiently.[1] The goal is to store large graphs in core memory for use with standard algorithms requiring random access. Our representations have applications to computation on large graphs (*e.g.*, the link graph of the web, telephone call graphs, or graphs representing large meshes), and in addition can be used for medium-size graphs on devices with limited memory (e.g. map graphs on a hand-held device). Furthermore even if the application is not limited by physical memory, the compact representations can be faster than standard representations because they have better cache characteristics. Our experiments confirm that this is the case on many real-world graphs.

For random graphs the space that can be saved by graph compression is quite limited—the information-theoretic lower bound for representing a random graph is $\Theta(m \log \frac{n^2}{m})$, where n is the number of vertices, and m is the minimum of the number of edges, or the number of edges in the complement. This bound can be matched by using difference encoded adjacency lists [136], and for sparse graphs the approach only saves a small constant factor over standard adjacency lists. Fortunately most graphs in practice are not random, and considerable savings can be achieved by taking advantage of structural properties.

Probably the most common structural property that real-world graphs have is that they have small separators. As described in Section 2.7, a graph has small separators if it and 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 $\Theta(m)$, for $m \geq 2n$. Planar graphs have $O(n^{1/2})$ separators [81] and play an important role in any partitioning of 2-dimensional space, such as 2-dimensional triangulated meshes. In fact there has been considerable work on compressing planar graphs (see related work below). 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 [85], as do most nearest-neighbor graphs in 3-dimensions. Furthermore many graphs without pre-defined embeddings

[1]This chapter is based on work done with Guy Blelloch and Ian Kash [15, 16].

in low dimensional spaces have small separators [132]. For example, the link structure of the web has small separators, as our experiments show.

In this chapter we are interested in compact representations of separable graphs (as described in Section 2.7). We describe four possible representations, each using $O(n)$ bits and supporting constant-time degree queries and listing of the neighbors in constant time per neighbor. Three of the four representations support constant-time adjacency queries as well. We assume the graphs are unlabeled—we are free to number the vertices. Even if a graph is not strictly separable (*e.g.*, some component cannot be effectively separated), our representations are likely to do well since they will compress the components that are separable. Our computational model is a Random-Access-Machine with constant-time operations on $O(\log n)$-bit words. We take advantage of the $O(\log n)$-bit parallelism in our algorithms.

Related Work. There has been considerable work on compressing unlabeled graphs. Turan [125] 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 [72], and He, Kao and Lu [65] later describe a technique that is optimal in the first order term. These results generalize to any graph with constant genus [82]. There have also been many results for sub-classes of planar graphs such as trees, triangulated meshes or triconnected planar graphs [72, 64, 105]. For dense graphs, Naor [92] describes a representation that reduces a lower order term over what is required by an adjacency matrix.

None of this work considers implementing fast queries. Jacobson [68] first showed how planar graphs can be represented using $O(n)$ bits while permitting adjacency queries in $O(\log n)$ time. Munro and Raman [91] improved the time for adjacency queries to $O(1)$ time. Chuang et. al. [40] improved the constant on the high order term for the space bound. All of these techniques were based on using representations for balanced parentheses. It seems unlikely the techniques will extend to the general case of graphs with small separators.

Using separators to compress graphs has been considered before. Deo and Litow [46] showed that separators can be used to compress graphs with bounded genus to $O(n)$ bits. He, Kao and Lu [65] use planar-graph separators to compress planar graphs to the optimal number of bits within a low-order term. Chakrabarti et al. [33] describe an experimental approach for compressing graphs that are represented as sparse matrices. None of these techniques, however, support queries.

There has been additional related work for the special case of representing the link structure of the Web [28, 1, 119, 22]. For this case, authors have taken advantage of the high degree of similarity between individual web pages. Authors have developed techniques for representing the outlinks of one page by its difference from the outlinks of another page. Exploiting this similarity allows strong compression: Boldi and Vigna [22] get 3 to 5 bits per link on a webgraph of 118 million pages, not counting the indexing structure. However, the references between compressed nodes mean that multiple nodes must be decompressed per query. Also, it is not clear that general separable graphs would have the similarity property they exploit.

Chakrabarti et al. [34] consider graph compression from the perspective of data mining: by examining the compressed representation of a graph, they seek to gain insight into its underlying structure. They make use of the graph-separator technique described here.

Our Structures. All of our data structures are based on recursively separating a graph and using the separators to renumber the vertices (first numbering one subgraph, then the other). Because of the properties of small separators, most edges will connect vertices that are close in this numbering. We take advantage of this property in encoding the edges.

The time to construct our representation depends on the time needed to recursively separate the graph (all other aspects take linear time). A polylogarithmic approximation of the separator size is sufficient for our bounds so the Leighton-Rao separator [78] gives a polynomial time separator for graphs satisfying an $O(n^c)$, $c < 1$ edge-separator theorem. For special graphs more efficient solutions are known, *e.g.*, for planar graphs [81] and well shaped meshes [85]. In practice fast heuristics work well for most graphs [71].

Our static graph representations are based on a difference-coded adjacency list (as described in Section 2.4). We sort the neighbor indices for each vertex, and store the differences d between adjacent pairs of neighbors using a logarithmic code. The vertex encodings are concatenated and indexed using a *select* structure (see Section 2.6) for fast access.

For graphs which allow edge separators, we show how to relabel the vertices using a recursive edge-separator decomposition (an *edge separator tree*). We show that this relabeling, combined with difference coding, reduces the cost of an adjacency table to $O(n)$ bits. This permits degree queries in $O(1)$ time and neighbor queries in $O(1)$ time per neighbor. To support constant time adjacency queries, we describe a separate structure based on directing the graph such that all vertices have bounded outdegree, then storing only the out-edges from each vertex.

For graphs which require vertex separators, we use a *vertex separator tree* to relabel the graph. A vertex of degree d is assigned d "shadow labels," and each adjacency list that refers to it uses a different label. An auxiliary data structure (making use of table lookup) can map the shadow labels to a unique label for each vertex in $O(1)$ time. We show that the space required for this is $O(n)$ bits. Adjacency queries are handled as in the edge-separator case.

Both of the representations described above are static. The third graph representation we present is a *semidynamic* version of the first representation using the variable-bit-length array structure of Section 3. It permits dynamic updates to vertices: the neighbors of a vertex can be rewritten in expected $O(|v|)$ time, where $|v|$ is the degree of the vertex. We say the representation is semidynamic since, although edges can be inserted and deleted at will, the space usage of the representation depends on the locality of the new edges with respect to the initial ordering.

The fourth representation we present is a semidynamic representation which permits dynamic updates to individual edges in expected $O(1)$ time. It is based on the variable-bit-length dictionary structure of Section 3. Edges are compressed and stored in the dictionary using a linked-list-like structure which allows access to individual elements in $O(1)$ time.

We implemented the first and third data structures described above and present results from extensive experimentation. We compare several methods for finding separators and for indexing the structure. We present results from several different prefix codes. We compare the performance of our representations on two machines with different cache characteristics. We compare our code to an array representation and to several variants of a linked-list representation. Finally, we present experimental results from two algorithms making use of the application. Our experiments show that our representations mostly dominate standard representations in terms of both space and query times. Our dynamic representation is slower than adjacency lists for updates.

In Section 5.2 we discuss our two static graph representations. In Section 5.3 and 5.4 we discuss our dynamic representations. In Section 5.5 we describe details specific to our implementation. In Sections 5.6 and 5.7 we report on experiments analyzing time and space for both the static and dynamic graphs. Our comparisons are made over a wide variety of graphs including graphs taken from finite-element meshes, VLSI circuits, map graphs, graphs of router connectivity, and link graphs of the web. All the graphs are sparse. To analyze query times we measure the time for a depth-first search (DFS) over the graph. We picked this measure since it requires visiting every edge exactly once (in each direction) and since it is a common subroutine in many algorithms.

For static graphs we compare our static representation to adjacency arrays. An adjacency array stores for each vertex an array of pointers to its neighbors. These arrays are concatenated into one large array with each vertex pointing to the beginning of its block. This representation takes about a factor of two less space than adjacency lists (requiring only one word for each directed edge and each vertex). For our static representation we compare four codes for encoding differences: gamma codes, snip codes, nibble codes, and byte codes. The different codes represent a tradeoff between time and space.

Averaged over our test graphs, the static representation with byte codes uses 12.5 bits per edge, and the snip code uses 9 bits per edge. This compares with 38 bits per edge for adjacency arrays. Due to caching effects, the time performance of adjacency arrays depends significantly on the ordering of the vertices. If the vertices are ordered randomly, then our static representation with byte codes is between 2.2 and 3.5 times faster than adjacency arrays for a DFS (depending on the machine). If the vertices are ordered using the separator order we use for compression, then the byte code is between .95 and 1.3 times faster than adjacency arrays.

For dynamic graphs we compare our dynamic representation to an optimized implementation of adjacency lists. The performance of the dynamic separator-based representation depends on the size of blocks used for storing the data. We present results for two settings, one optimized for space and the other for time. The representation optimized for space uses 11.6 bits per edge and the one optimized for time uses 18.8 bits per edge (averaged over all graphs). This compares with 76 bits per edge for adjacency lists.

As with adjacency arrays, the time performance of adjacency lists depends significantly on the ordering of the vertices. Furthermore, for adjacency lists the performance also depends significantly on the order in which edges are inserted (*i.e.*, whether adjacent edges end up on the same cache line). The runtime of the separator-based representation does not depend on insertion order. It is hard to summarize the time results other than to say that the performance of our time-optimized representation ranges from .9 to 8 times faster than adjacency lists for a DFS. The .9 is for separator ordering, linear insertion, and on the machine with a large cache-line size. The 8 is for random ordering and random insertion. The time for insertion on the separator-based representation is up to 4 times slower than adjacency lists.

In Section 5.8 we describe experimental results analyzing the performance of two algorithms. The first is a maximum-bipartite-matching algorithm and the second is an implementation of the Page et al. page-rank algorithm [95]. In both algorithms the graph is used many times over so it pays to use a static representation. We compare our static representation (using nibble codes) with both adjacency arrays and adjacency lists. For both algorithms our representation runs about as fast or faster, and saves a factor of between 3 and 4 in space.

All experiments run within physical memory so our speedup has nothing to do with disk access.

5.1.1 Real-world graphs have good separators

An edge-separator is a set of edges that, when removed, partitions a graph into two almost equal sized parts (see [104] for various definitions of "almost equal"). Similarly a vertex separator is a set of vertices that when removed (along with its incident edges) partitions a graph into two almost equal parts. The minimum edge (vertex) separator for a graph is the separator that minimizes the number of edges (vertices) removed. Informally we say that a graph has good separators if it and its subgraphs have minimum separators that are significantly better than expected for a random graph of its size. Having good separators indicates that the graph has some form of locality—edges are more likely to attach "near" vertices than far vertices.

Along with sparsity, having good separators is probably the most universal property of real-world graphs. The separator property of graphs has been used for many purposes, including VLSI layout [4], nested dissection for solving linear systems [80], partitioning graphs on to parallel processors [116], clustering [118], and computer vision [112]. Although finding a minimum separator for a graph is NP-hard, there are many algorithms that find good approximations [104]. Here we briefly review why graphs have good separators.

One reason that many graphs have good separators is because they are based on communities and hence have a local structure to them. Link graphs for the web have good separators since most links are either within a local domain or within some other form of community (e.g. computer science researchers, information on gardening, ...). This is not just true at one level (*i.e.*, either local or not), but is true hierarchically. Most graphs based on social networks have similar properties. Such graphs include citation graphs, phone-call graphs, and graphs based on friendship-relations. In fact Watts and Strogatz [132] conjecture that locality is one of the main properties of graphs based on social networks.

Another reason many graphs have good separators is that they are embedded in a low dimensional space. Most meshes that are used for various forms of simulation (e.g. finite element meshes) are embedded in two- or three-dimensional space. 2D meshes are often planar (although not always) and hence satisfy an $O(n^{1/2})$ vertex-separator theorem [81]. Well shaped 3D meshes are known to satisfy an $O(n^{2/3})$ vertex-separator theorem [85]. Graphs representing maps (roads, power-lines, pipes, the Internet) are embedded in a little more than two dimensions. Road maps are very close to planar, except in Pittsburgh. Power-line graphs and Internet graphs can have many crossings, but still have very good separators. Graphs representing the connectivity of VLSI circuits also have a lot of locality since ultimately they have to be laid out in two dimensions with only a small constant number of layers of connections. It is well understood that the size of the layout depends critically on the separator sizes [126].

Clearly certain graphs do not have good separators. Expander graphs by their very definition cannot have small separators.

5.2 Static Representation

We will consider three kinds of queries: degree queries, neighborhood queries, and adjacency queries. A degree query returns the degree of a vertex. A neighborhood query lists all the neighbors of a given vertex.

An adjacency query tests whether two vertices are adjacent.

Our primary data structure is a *difference coded adjacency list*, represented as an ordered set and encoded as described in Section 2.4. We assume the vertices have integer labels. If a vertex v has neighbors v_1, v_2, v_3, …, v_d in sorted order, then the data structure encodes the differences $v_1 - v$, $v_2 - v_1$, $v_3 - v_2$, …, $v_d - v_{d-1}$ contiguously in memory. The differences are encoded using any logarithmic code (as described in Section 2.3). The value $v_1 - v$ might be negative, so we store a sign bit for that value. At the start of each encoded list we also store a code for the number of entries in the list.

We form an *adjacency table* by concatenating the adjacency lists together in the order of the vertex labels. To access the adjacency list for a particular vertex we need to know its starting location. If we have n vertices and a total of $O(n)$ bits in the lists, keeping an $O(\log(n))$-bit pointer for each vertex would exceed our space bound; instead, we use a *select* data structure (as described in Section 2.6) to store the start locations using $O(n)$ bits.

Lemma 5.2.1 *An adjacency table supports degree queries in $O(1)$ time, and neighborhood queries in $O(|v|)$ time, where $|v|$ is the degree of the vertex being queried.*

Proof. The *select* operation allows access to the adjacency list for any vertex in constant time. Any $O(\log(n))$-bit value v can be decoded in constant time (using the `decode-gamma` routine from Section 2.3), so it takes $O(d)$ time to decode the contents of the list.

Edge Separators. We begin by discussing the case of graphs that admit edge separators. Our data structure for this case is highly practical and is used as the basis for our experimentation. We will later describe an extension to vertex separators. We note that, in practice, we found that all the real-world graphs we tested were edge-separable.

Our algorithm builds an edge-separator tree for the target graph by recursively computing an edge separator for each subgraph. The resulting separator tree contains one leaf for each vertex in the graph. The vertices are labeled in order from left to right using a traversal of the separator tree.

Lemma 5.2.2 *Suppose that the edges in a graph G are encoded in such a way that each edge (v_1, v_2) uses $O(\log|v_1 - v_2|)$ bits. If G is a member of a class of edge-separable graphs, and its vertices are labeled using an edge-separator tree, then the total space used to encode all the edges is $O(n)$ bits.*

Proof. If a node of the separator tree contains n vertices, then its separator contains $O(n^c)$ edges. Each of those edges connects a pair of vertices which are at most n apart in the labeling, so the cost of the edges in that separator is $O(n^c \log n)$. Because the graph is edge-separable, it has an $O(n^c)$ separator that guarantees each side of the partition will contain at most αn vertices. Let $S(n)$ be an upper bound on the the number of bits used to encode the edges of a graph with n vertices. If we let $\alpha < a < 1 - \alpha$, $S(n)$ satisfies the recurrence:

$$S(n) \leq S(an) + S(n - an) + O(n^c \log n)$$

This recurrence solves to $S(n) = O(n)$ (*e.g.*, using induction assuming $S(n) \leq k_1 n - k_2 n^{c'}$, $c < c' < 1$).

Consider an adjacency table representation which represents the neighbors of v by their direct differences from v: $v_1 - v, v_2 - v, v_3 - v, v_4 - v, \ldots$. By Lemma 5.2.2 the space usage of that representation would be $O(n)$ bits. Our adjacency table representation instead sorts the vertices and represents the differences $v_1 - v, v_2 - v_1, v_3 - v_2, v_4 - v_3, \ldots$. This representation is an improvement over the direct-difference representation; it also uses $O(n)$ bits.

If, for a given labeling of the vertices, the sum of the edge costs has the property

$$\sum_{(v_1,v_2)\in E} \log |v_1 - v_2| < kn,$$

we call the labeling *k-compact*. We have shown that an edge separator tree produces a k-compact labeling for classes of graphs which are edge-separable. This property will be useful for discussions of dynamic graphs.

Adjacency Queries. Using the difference-coded adjacency table described above, we can find all the neighbors of a vertex in optimal $O(d)$ time. However, resolving adjacency queries also takes $O(d)$ time, since it requires decoding the adjacency list of either u or v to see if it contains the other vertex. To answer adjacency queries in constant time, we first convert the target graph to a directed graph with bounded in-degree.

Lemma 5.2.3 *If a class of undirected graphs satisfies an n^c-separator theorem, then it is possible to direct the edges of any graph in that class so that the resulting graph has bounded in- (or out-) degree.*

Proof. We make use of the fact from Section 2.7 that any class of graphs satisfying such a theorem must have bounded density. We present an algorithm that directs the edges of such a graph so as to ensure that the result has bounded in-degree.

Given a graph G and a density bound b, our algorithm first selects the set V of vertices in G that have degree at most $2b$. At least half of the vertices in G must have this property. Our algorithm greedily directs all edges that have vertices in V such that those edges point toward vertices in V. This cannot cause vertices in V to exceed their in-edge bound, and it does not add in-edges to vertices that are not in V. Our algorithm then subtracts V from G and repeats the process on the remaining graph. When all vertices are eliminated, the process is complete, and no vertex has an in-degree greater than $2b$.

To handle adjacency queries we build an *in-edge adjacency table* that, for any vertex v, lists the label u corresponding to each in-edge (u, v). To do this we start with a full adjacency table (using $O(n)$ bits as described above) and discard the neighbors corresponding to out-edges. To test if vertices u and v are adjacent, an algorithm examines the adjacency list information for u and for v, and returns `true` if either vertex appears in the other's list. This takes $O(1)$ time since the lists are constant length.

It remains to calculate the space usage of the new table. If a neighbor v_i is discarded from a difference-encoded list, then the two differences $v_{i+1} - v_i$ and $v_i - v_{i-1}$ are replaced with the difference $v_{i+1} - v_{i-1}$. Since all the differences are integral, we have $\log(v_{i+1} - v_{i-1}) < \log(v_{i+1} - v_i) + \log(v_i - v_{i-1})$; asymptotically the space usage decreases. (For gamma codes, there exist cases where deleting an entry v_i

BUILDTREE(V, E)
  if $|E| = 1$ **then**
    return V

  $(V_a, V_{sep}, V_b) \leftarrow$ FINDSEPARATOR(V, E)
  $E_a \leftarrow \{(u, v) \in E | u \in V_a \vee v \in V_a\}$
  $E_b \leftarrow E - E_a$
  $V_{a,sep} \leftarrow V_a \cup V_{sep}$
  $V_{b,sep} \leftarrow V_b \cup V_{sep}$

  $T_a \leftarrow BuildTree(V_{a,sep}, E_a)$
  $T_b \leftarrow BuildTree(V_{b,sep}, E_b)$
  return SeparatorTree(T_a, V_{sep}, T_b)

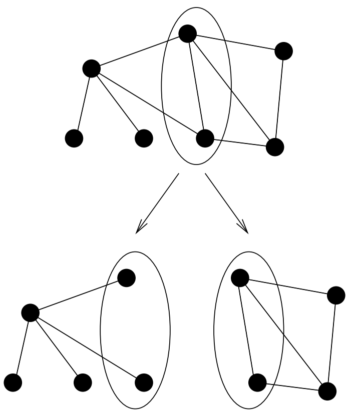


Figure 5.1: The BUILDTREE algorithm, and an example of the partition it produces.

from the list may increase the length of the list by one bit. This can contribute at most $O(n)$ bits to the table.) The new table is formed by discarding entries from a table of size $O(n)$ bits, so the new table has size $O(n)$ bits as well.

This gives us:

Lemma 5.2.4 *If class of undirected graphs satisfies an n^c-separator theorem, then an in-edge adjacency table for a graph in that class uses $O(n)$ bits and supports adjacency queries in $O(1)$ time.*

Vertex Separators. We now deal with the more general case of classes of graphs which allow vertex separators. As before, our algorithm builds a separator tree from the target graph, then uses it to order the vertices. The algorithm for building the separator tree is given in Figure 5.1. Without loss of generality, we assume that the graph separator algorithm always returns a separator with at least one vertex on each side (unless the target graph is a clique). If the target is a clique, we assume the separator contains all but one of the vertices, and that the remaining vertex is on the left side of the partition.

The algorithm we describe produces a separator tree in which the separator vertices at one level are included in both of the subgraphs at the next level [80]. Since each call to BUILDTREE partitions the edges, and the base case contains a single edge, the separator tree will have one leaf per edge. Consider a single vertex with degree d. Every time it appears in a separator, its edges are partitioned into two sets, and the vertex is copied to both recursive calls. Since the vertex will appear in d leaves, it must appear in $d - 1$ separators, so it will appear in $d - 1$ internal nodes of the separator tree. These $2d - 1$ total appearances define their own binary tree for the vertex, which we call the *shadow tree* for that vertex. An example is shown in Figure 5.2.

We label the appearance of vertices in the separator tree recursively: first the vertices on the left, then the vertices in the separator, then the vertices on the right. Note that a vertex of degree d will receive $2d - 1$ labels: one for each time it appears in the separator tree (i.e. one for each node in its shadow tree). We call the label assigned to the root of a shadow tree the *root label*, and use this label as the representative of the

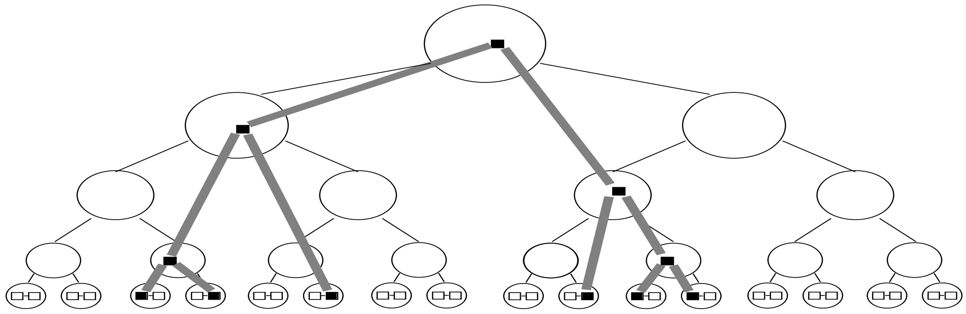


Figure 5.2: The separator tree, and a shadow tree corresponding to a vertex of degree 6.

vertex. (The labeling of representatives is sparse, but we can use the *select* and *rank* data structures (see Section 2.6) to efficiently convert it to a dense representation.) We refer to labels assigned to the leaves of a shadow tree as the *shadow labels* of that vertex. Note that, if a vertex has degree 1, then its root label will be a shadow label.

Property 5.2.1 *The separator tree of an n-vertex bounded-density graph is assigned $O(n)$ contiguous labels.*

This property holds since a vertex of degree d is assigned $2d-1$ labels, giving a total of $4m-n$ labels, and since $m = O(n)$ for bounded density graphs. If a graph is separable, then all graphs in the separator tree have this property.

We will represent graphs using two data structures. The first, the *shadow adjacency table*, will map the root label of each vertex to an adjacency list of shadow labels. The second, the *root-find structure*, will map each shadow label to the label of its root.

The Shadow Adjacency Table. The *shadow adjacency table* contains a difference coded adjacency list for each vertex, which is accessed using the root label of the vertex. If vertices u and v have shadow labels u' and v', and a leaf of the separator tree contains (u', v'), then the adjacency list for v contains u' and the adjacency list for u contains v'.

Lemma 5.2.5 *For classes of graphs satisfying an n^c-separator theorem with $c < 1$, any n-vertex member has a shadow adjacency table with $O(n)$ bits.*

Proof. Consider the adjacency list and shadow tree for a vertex v of degree d. There is a one-to-one correspondence between the d labels in the adjacency list and the d leaves of the shadow tree, and the corresponding labels differ by ± 1. We charge the difference between each adjacent pair of adjacency list labels to the least common ancestor of the corresponding leaves in the shadow tree. If the ancestor is a separator in a graph with s vertices, then the difference is $O(s)$ (by property 5.2.1), so the difference code uses $O(\log(s))$ bits. We treat the first difference in the list, $v_1 - v$, as a special case, and charge its bits to the root label. Note that this charges every node in the shadow tree at most twice.

We have charged $O(\log(s))$ bits to every appearance of a vertex in a separator for a graph with s vertices. Recall that the target graph has a βn^c separator that guarantees that each side of the partition will contain at most αn vertices. Let $S(n)$ be an upper bound on the the number of bits used to encode a graph with n vertices. If we let $\alpha < a < 1 - \alpha$, $S(n)$ satisfies the recurrence:

$$S(n) \leq S(an + \beta n^c) + S(n - an) + O(n^c \log n)$$

This recurrence solves to $S(n) = O(n)$ (*e.g.*, using induction assuming $S(n) \leq k_1 n - k_2 n^{c'}$, $c < c' < 1$). The number of bits to encode the lengths of each list is bounded by $O(n)$ since the total number of edges is $O(n)$ and the logarithm is a concave function.

We note that even if the separators are polylogarithmic approximations of the best cut, the recurrence still solves to $O(n)$.

The Root-Find Structure. The shadow adjacency table can find a set of shadow labels corresponding to the neighbors of any root label. We now describe a data structure that maps shadow labels to their root labels. We begin with a structure that allows us to perform lookups in $O(\min(\log(n), d))$ time for a vertex of degree d; we then show how to improve the structure so that we can perform these lookups in $O(1)$ time.

To allow root lookups, we assign to each label a pointer to its parent, encoded using the difference between the two labels, and indexed using a *select* data structure. (We use a one-bit token per label to indicate whether it is the root of its shadow tree.) If the parent of a label is in a separator of a graph with s vertices, then the pointer use $O(\log s)$ bits (by property 5.2.1). We charge the two child pointers to the parent, resulting in the same recurrence as in Lemma 5.2.5 and $O(n)$ bits. Using parent pointers, we can climb the tree from a shadow label to its root. The separator tree is $O(\log(n))$ levels high, and the height of the shadow tree is less than the number of nodes it contains, so the total time is $O(\min(\log(n), d))$.

To achieve a constant-time bound we use a blocking structure. We divide the labels into three categories based on their location in the separator tree. The category a label is in will determine the size of the pointer we allocate to each of its two children.

Labels appearing as separators in graphs containing at least $\log^{\frac{1}{1-c}}(n)$ vertices will be placed in the first category; we allocate a full $O(\log(n))$-bit root pointer for each of their children. Labels which appear as separators in graphs containing between $\log^{\frac{1}{1-c}}(n)$ and $\log^{\frac{1}{1-c}}(\log^{\frac{1}{1-c}}(n))$ vertices will be placed in the second category; they cannot have any children with a label that differs by more than $O(\log^{\frac{1}{1-c}}(n))$, so for their children we use an $O(\log\log^{\frac{1}{1-c}}(n)) = O(\log(\log(n)))$-bit offset pointer. These pointers will point to the topmost second-category label that is an ancestor of the child label in question; that label is guaranteed to have a first-category parent (if it has a parent at all). Labels in graphs containing less than $n_b = \log^{\frac{1}{1-c}}(\log^{\frac{1}{1-c}}(n))$ vertices will be considered “leaf labels” and will be placed in the third category. Rather than encoding these vertices explicitly, we will encode the graphs in which they appear. We will consider a maximal block of contiguous leaf labels to be a “leaf block”.

We examine each leaf block and remove from it all of the parent pointers that point to locations outside of the block; labels that had such pointers are marked as the roots of their shadow trees. We then make a table that lists all of the distinct leaf blocks in the data structure, and replace the individual leaf blocks with pointers into the table. The leaf blocks (with the parent pointers removed) all have less than n_b vertices, so each individual block requires $O(n_b)$ bits to encode. This means there can be at most $O(2^{kn_b})$ distinct leaf blocks, so the size of the pointer required per leaf block is also $O(n_b)$ bits. There are $O(n/n_b)$ pointers, so this is within our space bound.

We now examine the table, which contains $O(2^{kn_b})$ leaf blocks. We provide each shadow label in each leaf block with an $O(\log(n_b))$-bit pointer to its greatest ancestor within that block.

To shrink the number of graphs in the table we had to strip out all parent pointers that pointed out of the leaf graphs themselves. We include these pointers as an appendix to each leaf table pointer. The space for these pointers has already been charged to first- and second-category labels in the tree. We index the pointers using another application of our *select* data structure. For each leaf block we also store the label of the first entry modulo n_b (call this v_L). We charge this $\log(n_b)$ space to the $O(n_b)$ space of the table pointer. Each leaf block therefore contains a table pointer, an appendix, and v_L.

Given a shadow label s, we use the following procedure to find the root of its shadow tree. We first use the *select* data structure to find a pointer to the leaf block L containing s. We compute $s - v_L$ modulo n_b to find the index of the entry in L corresponding to s. That entry contains a pointer to the greatest ancestor of s in L. If this ancestor is not a root, we examine the appendix of the leaf block to find the greatest second-category ancestor s' of s. We use the *select* data structure again to find the greatest first-category ancestor s'' of s' (if needed). These operations all require constant time.

Lemma 5.2.6 *The root-find structure allows constant-time lookup of the root label corresponding to any shadow label, and uses $O(n)$ bits.*

Proof Outline. There are $O(n/\log(n))$ labels that receive $\log(n)$-bit pointers and O$(n/\log(\log(n)))$ labels that receive $O(\log(\log(n)))$-bit pointers; the space for these pointers is $O(n)$. There are $O(n/n_b)$ leaf table pointers, each using $O(n_b)$ bits; this space is also $O(n)$. The table contains $O(2^{kn_b})$ entries, each of which contains n_b pointers of $O(\log(n_b))$ bits each; the total space used is $O(2^{kn_b}n_b\log(n_b))$ which is sublinear.

The time bound is described above.

Adjacency queries can be handled using an *in-edge shadow adjacency table*. As in the edge-separator case (see Lemma 5.2.4), we begin by directing the graph so that all vertices have constant in-degree. We then discard from the shadow adjacency table all entries corresponding to out-edges.

Theorem 5.2.1 *For a class of graphs satisfying an n^c-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.*

Proof. To resolve degree queries and neighborhood queries we use a shadow adjacency table. Extracting the degree from a shadow adjacency table takes $O(1)$ time since it is encoded first; extracting the neighborhood takes $O(d)$ time (Lemmas 5.2.1 and 5.2.6). To resolve adjacency queries we use an in-edge shadow adjacency table. For an adjacency query on vertices u and v, we need only examine u and v in the second table since either (u, v) or (v, u) will be in the table. This takes $O(1)$ time since the lists are constant length.

5.3 Semidynamic Representation

Using the variable-bit-length array structure from Section 3.2, we can build a graph representation that supports insertion (and deletion) of edges in the graph. Although the insertions and deletions are dynamic, the space bound depends on the vertex labeling remaining k-compact. Thus we describe our representation as a whole as *semidynamic*.

In the static data structure, the data for each vertex is concatenated and stored in one chunk of memory, with a separate index to allow finding the start of each vertex. In the dynamic data structure, the data for each vertex is simply stored in the variable-bit-length array structure. The new representation, like the old, supports degree queries in $O(1)$ time and neighbor listing in $O(|1|)$ time per neighbor. In addition, the new representation allows insertion or deletion of edges by rewriting the data for the associated vertices. Inserting or deleting an edge (v_1, v_2) requires $O(|v_1| + |v_2|)$ expected time.

The space bound for the data structure is $\sum_{(v_1,v_2)\in E} \log |v_1 - v_2|$ bits. For a $k-$compact labeling of the graph this is $O(kn)$ (see Lemma 5.2.2). If the graph to be compressed is edge-separable, then the initial labeling will be $k-$compact for constant k; however, the space bound for the structure depends on the labeling remaining $k-$compact. Note that, for graphs having a fixed embedding in a low-dimensional space, any labeling which takes advantage of the embedding will remain $k-$compact as long as the edges have locality.

5.4 Semidynamic Representation with Adjacency Queries

Using the variable-bit dictionary structure from Section 3.3, we can build a graph representation supporting adjacency queries as well as neighbor queries.

Theorem 5.4.1 *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 $(u, v_i), 1 \leq i \leq d$ 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 adjacency testing, neighbor listing, and insertion and deletion of edges, using functions of the dictionary. Pseudocode for these operations is shown in Figure 5.3.

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 $O((|E| + |V|)w)$.

ADJACENT(u, v)
  return (LOOKUP$((u, v)) \neq$ **null**)

FIRSTEDGE(u)
  $(v_p, v_n) \leftarrow$LOOKUP$((u, u))$
  return v_n

NEXTEDGE(u, v)
  $(v_p, v_n) \leftarrow$LOOKUP$((u, v))$
  return v_n

ADDEDGE(u, v)
  $(v_p, v_n) \leftarrow$LOOKUP$((u, u))$
  INSERT$((u, u), (v_p, v))$
  INSERT$((u, v), (u, v_n))$

DELETEEDGE(u, v)
  $(v_p, v_n) \leftarrow$LOOKUP$((u, v))$
  $(v_{pp}, v) \leftarrow$LOOKUP$((u, v_p))$
  $(v, v_{nn}) \leftarrow$LOOKUP$((u, v_n))$
  INSERT$((u, v_p), (v_{pp}, v_n))$
  INSERT$((u, v_n), (v_p, v_{nn}))$
  DELETE$((u, v))$

Figure 5.3: Pseudocode to support our graph operations.

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 (v_1, v_2) with a logarithmic code is $O(\log |v_1 - v_2|)$. Each edge appears $O(1)$ times in the structure, so the total cost to encode all the edges is $\sum_{(v_1,v_2)\in E} \log |v_1 - v_2|$.

For a k-compact labeling, $\sum_{(v_1,v_2)\in E} \log |v_1 - v_2|$ is $O(kn)$ (see Lemma 5.2.2).

5.5 Implementation

Separator trees. We implemented three base algorithms for constructing edge-separator trees. Two of our algorithms are top-down; they begin with a graph and recursively compute its edge-separators. The remaining algorithm is bottom-up; it collapses edges of the graph, combining vertices into multivertices.

The first algorithm we considered, *bfs*, generates separators through breadth-first search (BFS). The algorithm finds an "extremal" vertex v_n by starting a BFS at a random vertex and using the last vertex encountered. The algorithm starts a second BFS at v_n and continues until it has visited half of the vertices in the graph. This is taken as the partition. We apply the BFS separator recursively to produce a separator tree.

Our second algorithm, *metis*, uses the Metis [71] graph partitioning library to construct a separator tree. Metis uses a multilevel partitioning technique in which the graph is coarsened, the coarse graph is partitioned, and the result is projected back onto the original graph using Kernighan-Lin refinement. This class of partitioning heuristic is the best known at this time [130]. We apply Metis recursively to produce a separator tree.

Our third algorithm, *bu* (for "bottom-up"), begins with the complete graph and repeatedly collapses edges until a single vertex remains. There are many heuristics that can be used to decide in what order to collapse the edges. After some experimentation, we settled on the priority metric $\frac{w(E_{AB})}{s(A)s(B)}$, where $w(E_{AB})$ is the number of edges between the multivertices A and B, and $s(A)$ is the number of original vertices contained in multivertex A. The resulting process of collapsing edges creates a separator tree, in which every two merged vertices become the children of the resulting multivertex. To improve performance we also use a variant of *bu*, which we call *bu-bpq*, that uses a bucketed priority queue with $O(\log n)$ buckets.

There is a certain degree of freedom in the way we construct a separator tree: when we partition a graph, we can arbitrarily decide which side of the partition will become the left or right child in the tree. To take advantage of this degree of freedom we can use an optimization called *child-flipping*. A child-flipping algorithm traverses the separator tree, keeping track of the nodes containing vertices which appear before and after the current node in the numbering. (These nodes correspond to the left child of the current node's left ancestor and the right child of the current node's right ancestor.) If those nodes are N_L and N_R, the current node's children are N_1 and N_2, and E_{AB} denotes the number of edges between the vertices in two nodes, then our child-flipping heuristic rotates N_1 and N_2 to ensure that $E_{N_L N_1} + E_{N_2 N_R} \geq E_{N_L N_2} + E_{N_1 N_R}$. This heuristic can be applied to any separator tree as a postprocessing phase.

Indexing structures. Our algorithms use a select data structure to map the vertex numbers to the bit position of the start of the appropriate adjacency list. We will henceforth call this the *indexing structure*. We implemented four versions, representing different tradeoffs between space required and lookup time.

The simplest indexing structure, *direct*, stores an array of offset pointers, one for each vertex. Each pointer uses $\Theta(\log(n))$ bits, giving a total of $\Theta(n\log(n))$ bits. Only one memory access is required to locate the start of any vertex, making this method very fast.

We implemented a structure, *indirect*, that uses $O(n)$ bits and has constant access time. This is significantly simpler than the $o(n)$-bit structure of Munro [90]. To index the vertices, we first divide them into blocks of $\log(n)$ vertices each. We divide the blocks into subblocks, each of which contains a minimal number of vertices totaling at least $k\log(n)$ bits for some constant k. We store a $\log(n)$-bit pointer to each block in a global array, and we store an $O(\log(n))$-bit pointer to each subblock at the start of its parent block. Each block also contains a bit vector with one bit per vertex. A vertex's bit is set to 1 if that vertex is the first in its subblock. This all requires $O(n)$ bits. We consider two settings of k: $k = 1$ (*indirect-1*) and $k = 16$ (*indirect-16*).

To find the location of any vertex we first perform an array lookup to find the location of the block containing the target vertex. We then examine that block's bit vector to see which subblock the vertex is in, find the subblock offset using the subblock pointers, and decode the subblock. This all takes constant time—determining the subblock and decoding the subblock can both be implemented using table lookup on $\Theta(\log(n))$ bits in constant time.

As a compromise between the two indexing structures we consider a class of structures called *semidirect*, based on allocating one full pointer for each group of k vertices, and representing the remainder of the offsets as differences. The *semidirect-4* structure uses a one-word pointer per four vertices and fits three ten-bit offsets into a second word. If one of the offsets doesn't fit in ten bits, they are stored elsewhere, and the second word is a pointer to them.

The *semidirect-16* structure stores the start locations for sixteen vertices in five 32-bit words. The first word contains the offset to vertex 0—that is, the first of the sixteen vertices being represented. The second word contains three ten-bit offsets from the first vertex to starts of vertices 4, 8, and 12. The next three words contain twelve eight-bit offsets to the remaining twelve vertices. Each of the twelve vertices is stored by an offset relative to one of the four vertices already encoded. For example, the start of vertex 14 is encoded by its offset from the start of vertex 12. As before, if at any point the offsets do not fit in the space provided, they are stored elsewhere, and the table contains a pointer to them.

Codes and Decoding. We considered several logarithmic codes for use in our representations. In addition to the *gamma code* [50] we considered the *snip code*, *nibble code*, and *byte code*, as described in Section 2.3.

We also implemented a variant Huffman code. Rather than store a frequency table and Huffman tree for the entire range of n possible differences, we truncated the table at 4096 entries. Any gaps larger than that were coded using an escape sequence and then stored using a flat $\log n$ bits. (In all cases fewer than 4% of gaps were over 4096 in length.) To decode the Huffman codes we used a decoding table of width 8 bits. At most thirty leaves of the Huffman tree represented codewords of length 8 or less, but those leaves always represented at least 75% of the weight of the tree. If a codeword was longer than 8 bits, the decoding table gave a pointer to the eighth level of the tree, and the remainder of the word was decoded using the tree.

Dynamic Structure. Our dynamic structure manages memory in blocks of fixed size. The data structure initially contains an array with one memory block for each vertex. If additional memory is needed to store the data for a vertex, the vertex is assigned additional blocks, allocated from a pool of spare memory blocks. The blocks are connected into a linked list.

When we allocate an additional block for a vertex, we use part of the previous block to store a pointer to the new one. We use a hashing technique to reduce the size of these pointers to only 8 bits. To work efficiently the technique requires that a constant fraction of the blocks remain empty. This requires a hash function that maps (address, i) pairs to addresses in the spare memory pool. Our representation tests values of i in the range 0 to 127 until the result of the hash is an unused block. It then uses that value of i as the pointer to the block.

If the hash function is drawn from a uniform family, and the memory pool is at most 80% full, then the probability that this technique will fail is at most $.80^{128} \simeq 4 * 10^{-13}$. In practice we use a hash function $h(a, i) = pa + r[i] \bmod s$ where a is the address, p is a prime, r is a table of 128 random numbers, and s is the size of the memory pool. This is sufficient to fill the pool to slightly more than 80%.

To help ensure memory locality, a separate pool of contiguous memory blocks is allocated for each 1024 vertices of the graph. If a given pool runs out of memory, it is resized. Since the pools of memory blocks are fairly small this resizing is relatively efficient.

For graph operations that have high locality, such as repeated insertions to the same vertex, it may be

Graph	Vtxs	Edges	Max Degree	Source
auto	448695	6629222	37	3D mesh [130]
feocean	143437	819186	6	3D mesh [130]
m14b	214765	3358036	40	3D mesh [130]
ibm17	185495	4471432	150	circuit [3]
ibm18	210613	4443720	173	circuit [3]
CA	1971281	5533214	12	street map [127]
PA	1090920	3083796	9	street map [127]
googleI	916428	5105039	6326	web links [56]
googleO	916428	5105039	456	web links [56]
lucent	112969	363278	423	routers [107]
scan	228298	640336	1937	routers [107]

Table 5.1: Properties of the graphs used in our experiments.

inefficient to repeatedly encode and decode the neighbors of a vertex. We implemented a variant of our structure that uses caching to improve access times. When a vertex is queried, its neighbors are decoded and stored in a temporary adjacency list structure. Memory for this structure is drawn from a separate pool of list nodes of limited size. The pool is managed in first in first out (FIFO) mode. A modified vertex that is flushed from the pool is written back to the main data structure in compressed form. We maintain the uncompressed adjacency lists in sorted order (by neighbor label) to facilitate writing them back.

5.6 Experimental Setup

Graphs. We drew test graphs for our experiments from several sources: 3D Mesh graphs from the online Graph Partitioning Archive [130], street connectivity graphs from the Census Bureau Tiger/Line data [127, 117], graphs of router connectivity from the SCAN project [107], graphs of webpage connectivity from the Google [56] programming contest data, and circuit graphs from the ISPD98 Circuit Benchmark Suite [3]. The circuit graphs were initially hypergraphs; we converted them to standard graphs by converting each net into a clique. Properties of these graphs are shown in Table 5.1. For edges we list the number of directed edges in the graph. For the directed graphs (googleI and googleO) we take the degree of a vertex to be the number of elements in its adjacency list.

Machines and compiler. The experiments were run on two machines, each with 32-bit processors but with quite different memory systems. The first uses a .7GHz Pentium III processor with .1GHz frontside bus and 1GB of RAM. The second uses a 2.4GHz Pentium 4 processor with .8GHz frontside bus and 1GB of RAM. The Pentium III has a cache-line size of 32 bytes, while the Pentium 4 has an effective cache-line size of 128 bytes. The Pentium 4 also supports quadruple loads and hardware prefetching, which are very effective for loading consecutive blocks from memory, but not very useful for random access. The Pentium 4 therefore performs much better on the experiments with strong spatial locality (even more than the factor of

3.4 in processor speed would indicate), but not particularly well on the experiments without spatial locality. All code is written in C and C++ and compiled using g++ (3.2.3) using RedHat Linux 7.1.

Benchmarks. We present times for depth-first-search as well as times for reading and inserting all edges. We select a DFS since it visits every edge once, and visits the vertices in a non-trivial order exposing caching issues better than simply reading the edges for each vertex in linear order. Our implementation of DFS uses a character array of length n to mark the visited vertices, and a stack to store the vertices to return to. It does nothing other than traverse the graph. For reading the edges we present times both for accessing the vertices in linear order and for accessing them in random order. In both cases the edges within a vertex are read in linear order. For inserting we insert in three different orders: *linear*, *transpose*, and *random*. Linear insertion inserts all the out-edges for the first vertex, then the second, etc.. Transpose insertion inserts all the in-edges for the first vertex, then the second, etc.. Note that an in-edge (i, j) for vertex j goes into the adjacency list of vertex i not j. Random insertion inserts the edges in random order.

We compare the performance of our data structure to that of standard linked-list and array-based data structures, and to the LEDA [84] package. Since small differences in the implementation can make significant differences in performance, here we describe important details of these implementations.

Adjacency lists. We use a singly linked-list data structure. The data structure uses a *vertex-array* of length n to access the lists. Each array element i contains the degree of vertex i and a pointer to a linked list of the out-neighbors of vertex i. Each link in the list contains two words: an integer index for the neighbor and a pointer for the next link. We use our own memory management for the links using free lists—no space is wasted for header or tail words. The space required is therefore $2n + 2m + O(1)$ words (32 bits each for the machines we used). Assuming no deletions, sequential allocation returns consecutive locations in memory—this is important for understanding spatial locality.

In our experiments we measured DFS runtimes after inserting the edges in three orders: linear, transpose, and random. These insertion orders are described above. The insertion orders have a major effect on the runtime for accessing the linked lists—the times for DFS vary by up to a factor of 11 due to the insertion order. For linear insertion all the links for a given vertex will be in adjacent physical memory locations giving a high degree of spatial locality. This means when an adjacency list is traversed, most of the links will be found in the cache—they are likely to reside on the same cache line as the previous link. This is especially true for our experiments on the Pentium 4 which has 128-byte cache lines (each cache line can fit 16 links). For random insertion, and assuming the graph does not fit in cache, accessing every link is likely to be a cache miss since memory is being accessed in completely random order.

We also measured runtimes with the vertices labeled in two orders: *randomized* and *separator*. In the randomized labeling the integer labels are assigned randomly. In the separator labeling we use the labeling generated by our graph separator—the same as used by our compression technique. The separator labeling gives better spatial locality in accessing both the vertex-array and the visited-array during a DFS. This is because loading the data for a vertex will load the data for nearby vertices which are on the same cache-line. Following an edge to a neighbor is then likely to access a vertex nearby in the ordering and still in cache. If linear insertion is used, the separator labeling also improves locality on accessing the links during a DFS. This is because the links for neighboring vertices will often fall on the same cache lines. We were actually surprised at what a strong effect labeling based on separators had on performance. The performance varied by up to a factor of 7 for the graphs with low degree and the machine with 128-byte cache lines.

Adjacency Array. The adjacency array data structure is a static representation. It stores the out-edges of each vertex in an edge-array, with one integer per edge (the index of the out neighbor). The edge-arrays for the vertices are stored one after the other in the order of the vertices. A separate vertex-array points to the start of the edge-array for each vertex. The number of out-edges of vertex i can be determined by taking the difference of the pointer to the edge array for vertex i and the edge array for vertex $i + 1$. The total space required for an adjacency array is $n + m + O(1)$ words. For static representations it makes no sense to talk about different insertion orders of the edges. The ordering of the vertex labeling, however, can make a significant difference in performance. As with the linked-list data-structure we measured runtimes with the vertices labeled in randomized and separator order. Also as with linked lists, using the separator ordering improved performance significantly, again by up to a factor of 7.

LEDA. We also ran all our experiments using LEDA [84] version 4.4.1. Our experiments use the LEDA `graph` object and use the `forall_outedges` and `forall_vertices` for the loops over edges and vertices. All code was compiled with the flag `LEDA_CHECKING_OFF`. For analyzing the space for the LEDA data structure we use the formula from the LEDA book [84, page 281]: $52n + 44m + O(1)$ bytes. We note that comparing space and time to LEDA is not really fair since LEDA has many more features than our data structures. For example the directed graph data structure in LEDA stores a linked list of both the in-edges and out-edges for each vertex. Our data structures only store the out-edges. LEDA also stores the edges in a doubly-linked list allowing traversal in either direction and a simpler deletion of edges.

5.7 Experimental Results

Our experiments measure the tradeoffs of various parameters in our data structures. This includes the type of prefix code used in both the static and dynamic cases, and the block size used and the use of caching in the dynamic case. We also study a version that difference encodes out-edges relative to the source vertex rather than the previous out-edge. This can be used by applications which need control of the ordering of the out-edges. For example, our compact representation of simplicial meshes (described in Chapter 7) encodes out-edges relative to the source vertex.

5.7.1 Separator Algorithms.

In analyzing the efficiency of our techniques, there are three parameters of concern: the query times, the time to create the structures, and the space usage. The space usage has two components: the space for the adjacency lists, and the space for the indexing structure. The time to create the structure is dominated by time to order the vertices. There is a time/space tradeoff between the time used to order the vertices and the space needed for the adjacency table (spending more time on ordering produces better compression of the encoded lists). There is also a space/time tradeoff between the space used for the indexing structure

	dfs		metis-cf		bfs		bu-bpq		bu-cf		Degree
	T_1	Space	T/T_1	Space	T/T_1	Space	T/T_1	Space	T/T_1	Space	Space
auto	0.79s	9.88	153.11	$\boxed{5.17}$	27.69	5.96	7.54	5.90	14.59	5.52	0.56
feocean	0.06s	13.88	388.83	7.66	61.00	$\boxed{7.62}$	17.16	8.45	34.83	7.79	1.15
m14b	0.31s	10.65	181.41	$\boxed{4.81}$	32.0	5.85	8.16	5.45	15.32	5.13	0.54
ibm17	0.44s	13.01	136.43	$\boxed{6.18}$	21.38	9.40	11.0	6.79	20.25	6.64	0.36
ibm18	0.48s	11.88	129.22	$\boxed{5.72}$	22.54	8.29	9.5	6.24	17.29	6.13	0.40
CA	0.76s	8.41	382.67	4.38	88.22	7.05	14.61	4.90	35.21	$\boxed{4.29}$	1.66
PA	0.43s	8.47	364.06	4.45	79.09	7.03	13.95	4.98	33.02	$\boxed{4.37}$	1.64
googleI	1.4s	7.44	186.91	$\boxed{4.08}$	47.12	8.68	12.71	4.18	40.96	4.14	0.82
googleO	1.4s	11.03	186.91	6.78	47.12	13.11	12.71	6.21	40.96	$\boxed{6.05}$	0.95
lucent	0.04s	7.56	390.75	5.52	55.0	15.24	19.5	5.54	45.75	$\boxed{5.44}$	1.43
scan	0.12s	8.00	280.25	5.94	38.75	18.05	23.33	5.76	81.75	$\boxed{5.66}$	1.45
Avg		10.02	**252.78**	5.52	**47.26**	9.66	**13.65**	5.86	**34.54**	5.56	1.00

Table 5.2: The performance (time used and compression achieved) of several of our ordering algorithms, compared to a depth-first-search ordering. Space is in bits per edge for encoding the edges; T_1 is in seconds and the other times are normalized to it. The space to encode the degree of each vertex is listed separately (in bits per edge).

	Array			Our Structure									
	Rand	Sep		Byte		Nibble		Snip		Huffman		DiffByte	
Graph	T_1	T/T_1	Space	T/T_1	Space	T/T_1	Space	T/T_1	Space	T/T_1	Space	T/T_1	Space
auto	0.268s	0.313	34.17	0.294	10.25	0.585	7.42	0.776	6.99	0.828	6.81	0.399	12.33
feocean	0.048s	0.312	37.60	0.312	12.79	0.604	10.86	0.791	11.12	0.813	10.83	0.374	13.28
m14b	0.103s	0.388	34.05	0.349	10.01	0.728	7.10	0.970	6.55	1.078	6.37	0.504	11.97
ibm17	0.095s	0.536	33.33	0.536	10.19	1.115	7.72	1.400	7.58	1.621	6.93	0.747	12.85
ibm18	0.113s	0.398	33.52	0.442	10.24	0.867	7.53	1.070	7.18	1.301	6.44	0.548	12.16
CA	0.920s	0.126	43.40	0.146	14.77	0.243	10.65	0.293	10.55	0.304	11.46	0.167	14.81
PA	0.487s	0.137	43.32	0.156	14.76	0.258	10.65	0.310	10.60	0.314	11.19	0.178	14.80
lucent	0.030s	0.266	41.95	0.3	14.53	0.5	11.05	0.566	10.79	0.600	11.06	0.333	14.96
scan	0.067s	0.208	43.41	0.253	15.46	0.402	11.84	0.477	11.61	0.493	11.67	0.298	16.46
googleI	0.367s	0.226	37.74	0.258	11.93	0.405	8.39	0.452	7.37	0.790	7.01	0.302	13.39
googleO	0.363s	0.250	37.74	0.278	12.59	0.460	9.72	0.556	9.43	0.689	9.03	0.327	13.28
Avg		0.287	38.202	0.302	12.501	0.561	9.357	0.696	9.07	0.803	8.98	0.380	13.662

Table 5.3: Performance of our **static** algorithms compared to performance of an adjacency array representation. Space is in bits per edge; time is for a DFS, normalized to the first column, which is given in seconds.

	List	Array	Direct		semidirect-4		indirect-1		indirect-16	
	T_ℓ	T/T_ℓ	T/T_ℓ	Space	T/T_ℓ	Space	T/T_ℓ	Space	T/T_ℓ	Space
auto	0.60	0.39	0.83	2.17	0.83	1.08	0.98	1.0	1.33	0.4
feocean	0.08	0.53	1.07	5.6	1.09	2.8	1.46	2.36	2.21	0.79
m14b	0.29	0.38	0.81	2.05	0.82	1.02	0.97	0.94	1.30	0.39
ibm17	0.39	0.38	0.83	1.33	0.85	0.7	0.95	0.68	1.12	0.36
ibm18	0.38	0.36	0.80	1.52	0.82	0.79	0.93	0.78	1.14	0.37
CA	0.56	0.60	0.95	11.4	1.03	5.7	1.95	2.87	4.31	1.13
PA	0.31	0.59	0.96	11.32	1.03	5.66	1.94	2.87	4.26	1.11
googleI	0.49	0.48	0.88	5.74	0.92	2.89	1.43	1.78	2.45	0.67
googleO	0.49	0.47	0.98	5.74	1.02	2.88	1.51	2.05	2.36	0.76
lucent	0.03	0.55	1.22	9.95	1.27	4.98	2.11	3.06	3.83	1.11
scan	0.06	0.55	1.20	11.41	1.28	5.73	2.30	3.41	4.36	1.2
Avg		**0.48**	**0.96**	6.20	**1.00**	3.11	**1.50**	1.98	**2.61**	0.75

Figure 5.4: The performance of various direct and indirect indexing schemes on our Pentium III. Space is measured in bits per edge; T_ℓ is in seconds and the other times are normalized to it. Graphs were compressed using gamma codes.

and the time needed for queries (using more space for the indexing structure gives faster query times). Our experiments demonstrate these tradeoffs. Timings given here are from the Pentium III.

Table 5.2 illustrates the tradeoff between the time needed to generate an ordering, and the space needed by the compressed adjacency lists that use that ordering. For these experiments we use gamma codes for compression. In addition to the separator schemes discussed in Section 5.5, we include a very simple ordering based on a depth-first-search post-order numbering of the graphs. In general *bu-cf* and *metis-cf* produce the highest quality orderings (*cf* indicates that it performs child flipping). The bottom up technique (*bu-cf*), however, is significantly faster. We include results for *bu-bpq* (no child flipping, and approximate priorities) since its ordering is almost as good as *bu-cf*, but is a factor of three faster. The *bfs* algorithm does well on regular graphs but badly on highly irregular graphs.

For the rest of our experiments we chose the *bu-cf* ordering, which gave the best performance on many of our test graphs while being significantly faster than *metis-cf*.

5.7.2 Indexing structures

Figure 5.4 illustrates the tradeoff between the query time and the space needed for the indexing structure, and also compares query times to standard uncompressed data structures. To measure query time we use the time to execute a depth-first search (DFS). This is a reasonable measure since it requires visiting all the edges once. We compare the performance of our representations to that of standard linked-list and array-based graph representations. The linked-list representation uses two 32-bit words per edge, one for the neighbor label and one for the next pointer in the linked list. The array-based representation stores the neighbor indices of each vertex contiguously in one large array with the lists for the vertices placed one after

	Direct		semidirect-4		semidirect-16	
Graph	T	Space	T	Space	T	Space
auto	0.351	4	0.343	2.00	0.345	1.25
feocean	0.056	4	0.055	2.00	0.056	1.25
m14b	0.158	4	0.155	2.00	0.156	1.25
ibm17	0.219	4	0.216	2.00	0.216	1.25
ibm18	0.208	4	0.204	2.00	0.206	1.26
CA	0.527	4	0.518	2.00	0.529	1.25
PA	0.291	4	0.285	2.00	0.290	1.25
googleI	0.342	4	0.332	2.00	0.333	1.27
googleO	0.378	4	0.370	2.00	0.371	1.25
lucent	0.026	4	0.027	2.00	0.028	1.25
scan	0.051	4	0.050	2.00	0.052	1.25

Figure 5.5: Comparison of the semidirect-16 indexing structure to other structures implemented on our new codebase on our Pentium III. Space is given in bytes per vertex; time is given in seconds required for a DFS. Graphs were compressed using nibble codes.

the other. It uses one 32-bit word per edge. Both representations use an array to index the vertices using an additional 32-bit word per vertex. We note that linked lists are well suited for insertions and deletions, while, like our representation, arrays are best suited for static graphs. For all versions of DFS we use one byte (8 bits) per vertex to mark whether it has been visited.

The results show that for the direct and semidirect-4 indexing structure our compressed representation is slightly faster than the linked-list representation. This is not surprising since although there is overhead for decoding the adjacency lists, the cache locality is significantly better (loading a single cache line can decode many edges). Our representation is slower than the array-based representation. This is also not surprising since the array-based representation also has good spatial locality (the edges of a vertex are adjacent in memory), but does not have decoding overhead. We note that the graph sizes are such that for all representations the graphs fit into physical memory but do not fit into the cache (except perhaps lucent, scan and feocean).

The semidirect-4 indexing structure saves a factor of two in space over the direct structure while requiring little extra time; we conclude that it is the most practical of the indexing structures tested. After completing these experiments, though, we migrated to a new codebase, supporting a wider variety of coding techniques. Within this new codebase we developed the semidirect-16 indexing structure. Table 5.5 presents a comparison of the semidirect-16 structure to the direct and semidirect-4 structures. (Since the indirect structures were complex and performed poorly, we did not reimplement them for further testing.) The semidirect-16 structure saved about six bits per vertex over the semidirect-4 structure while causing virtually no slowdown. Accordingly we use it exclusively in further experiments.

	3		4		8		12		16		20	
Graph	T_1	Space	T/T_1	Space	T/T_1	Space	T/T_1	Space	T/T_1	Space	T/T_1	Space
auto	0.318s	11.60	0.874	10.51	0.723	9.86	0.613	10.36	0.540	9.35	0.534	11.07
feocean	0.044s	14.66	0.863	13.79	0.704	12.97	0.681	17.25	0.727	22.94	0.750	28.63
m14b	0.146s	11.11	0.876	10.07	0.684	9.41	0.630	10.00	0.554	8.92	0.554	10.46
ibm17	0.285s	12.95	0.849	11.59	0.614	10.44	0.529	10.53	0.491	10.95	0.459	11.39
ibm18	0.236s	12.41	0.847	11.14	0.635	10.12	0.563	10.36	0.521	10.97	0.5	11.64
CA	0.212s	10.62	0.943	12.42	0.952	23.52	1.0	35.10	1.018	46.68	1.066	58.26
PA	0.119s	10.69	0.941	12.41	0.949	23.35	1.0	34.85	1.025	46.35	1.058	57.85
lucent	0.018s	13.67	0.888	14.79	0.833	22.55	0.833	31.64	0.833	41.22	0.888	51.09
scan	0.034s	15.23	0.941	16.86	0.852	26.39	0.852	37.06	0.852	48.08	0.882	59.34
googleI	0.230s	11.91	0.895	12.04	0.752	15.71	0.730	20.53	0.730	25.78	0.726	31.21
googleO	0.278s	13.62	0.863	13.28	0.694	15.65	0.658	19.52	0.640	24.24	0.676	29.66
Avg		12.58	0.889	12.62	0.763	16.36	0.735	21.56	0.721	26.86	0.736	32.78

Table 5.4: Performance of our dynamic algorithm using nibble codes with various block sizes. For each size we give the space needed in bits per edge (assuming enough blocks to leave the secondary hash table 80% full) and the time needed to perform a DFS. Times are normalized to the first column, which is given in seconds.

5.7.3 Static representations

Table 5.3 presents results comparing space and DFS times for the static representations for all the graphs on the Pentium 4. (Tables 5.6 and 5.7 present summary results for a wider set of operations on both the Pentium III and Pentium 4.) In Table 5.3 all times are normalized to the first column, which is given in seconds. The average times in the bottom row are averages of the normalized times, so the large graphs are not weighted more heavily. All times are for a DFS.

For the adjacency-array representation, times are given for the vertices ordered both randomly (Rand) and using our separator ordering (Sep). As can be seen, the ordering can affect performance by up to a factor of 8 for the graphs with low average degree (*i.e.*, PA and CA), and a factor of 3.5 averaged over all the graphs. This indicates that the ordering generated by graph separation is not only useful for compression, but is also critical for performance on standard representations (we will see an even more pronounced effect with adjacency lists). The advantage of using separator orders to enhance spatial locality has been previously studied for use in sparse-matrix vector multiply [122, 62], but not well studied for other graph algorithms. For adjacency arrays the ordering does not affect space.

For our static representation, times and space are given for four different prefix codes: Byte, Nibble, Snip and Gamma. The results show that byte codes are significantly faster than the other codes (almost twice as fast as the next fastest code). This is not surprising given that the byte codes take advantage of the byte instructions of the machine. The difference is not as large on the Pentium III (a factor of 1.45). It should be noted that the Gamma codes are almost never better than Snip codes in terms of time or space.

We also include results for the DiffByte code, a version of our byte code that encodes each edge as the difference between the target and source, rather than the difference between the target and previous target. This increases the space since the differences are larger and require more bits to encode. Furthermore each difference requires a sign bit. It increases time both since there are more bits to decode, and because the sign bits need to be extracted. Overall these effects worsen the space bound by an average of 10% and the time bound by an average of 25%.

Comparing adjacency arrays with the separator structures we see that the separator-based representation with byte codes is a factor of 3.3 faster than adjacency arrays with random ordering but about 5% slower for the separator ordering. On the Pentium III the byte codes are always faster, by factors of 2.2 (.729/.330) and 1.3 (.429/.330) respectively (see Table 5.7). The compressed format of the byte codes means that they require less memory throughput than for adjacency arrays. This is what gives the byte codes an advantage on the Pentium III since more neighbors get loaded on each cache line requiring fewer main-memory accesses. On the Pentium 4 the effective cache-line size and memory throughput is large enough that the advantage is reduced.

Table 5.6, later in the section, describes the time cost of simply reading all the edges in a graph (without the effect of cache locality).

5.7.4 Dynamic representations

A key parameter for the dynamic representation is selecting the size of the blocks used to store difference codes. Large blocks are inefficient since they contain unused space; small blocks can be inefficient since they require proportionally more space for pointers to other blocks. In addition, there is a time cost for traversing from one block to the next. This cost includes both the time for computing the hash pointer and the potential time for a cache miss. Because of this larger blocks are almost always faster.

Table 5.4 presents the time and space for a range of block sizes. The results are based on nibble codes on the Pentium 4 processor. The results for the other codes and the Pentium III are qualitatively the same, although the time on the Pentium III is less sensitive to the block size. For all space reported in this section we size the backup memory so that it is 80% full, and include the 20% unused memory in the reported space. As should be expected, for the graphs with high degree the larger block sizes are more efficient while for the graphs with smaller degree the smaller block sizes are more efficient. It would not be hard to dynamically decide on a block size based on the average degree of the graph (the size of the backup memory needs to grow dynamically anyway). Also note that there is a time-space tradeoff and depending on whether time or space is more important a user might want to use larger blocks (for time) or smaller blocks (for space).

Table 5.5 presents results comparing space and DFS times for the dynamic representations for all the graphs on the Pentium 4. It gives six timings for linked lists corresponding to the two labeling orders and for each labeling, the three insertion orders. The space for all these orders is the same. The table also gives space and time for two settings of our dynamic data structure: Time Opt and Space Opt. Time Opt uses byte codes and is based on a block size that optimizes time.[2] Space Opt uses the more space-efficient nibble codes and is based on a block size that optimizes space.

As with the adjacency-array representation, the vertex label ordering can have a large effect on performance for adjacency-lists, up to a factor of 7. In addition to the label ordering, the insertion ordering can also make a large difference in performance for adjacency-lists. The insertion order can cause up to a factor of 11 difference in performance for the graphs with high average degree (e.g. auto, ibm17 and ibm18) and

[2]We actually pick a setting that optimizes T^3S where T is time and S is space. This is because the time gains for larger blocks become vanishingly small and can be at a large cost in regards to space. For space optimal we optimize TS^3.

	Linked List							Our Structure					
	Random Vtx Order			Sep Vtx Order				Space Opt			Time Opt		
	Rand	Trans	Lin	Rand	Trans	Lin		Block	Time		Block	Time	
Graph	T_1	T/T_1	T/T_1	T/T_1	T/T_1	T/T_1	Space	Size	T/T_1	Space	Size	T/T_1	Space
auto	1.160s	0.512	0.260	0.862	0.196	0.093	68.33	16	0.148	9.35	20	0.087	13.31
feocean	0.136s	0.617	0.389	0.801	0.176	0.147	75.21	8	0.227	12.97	10	0.117	14.71
m14b	0.565s	0.442	0.215	0.884	0.184	0.090	68.09	16	0.143	8.92	20	0.086	13.53
ibm17	0.735s	0.571	0.152	0.904	0.357	0.091	66.66	12	0.205	10.53	20	0.118	14.52
ibm18	0.730s	0.524	0.179	0.890	0.276	0.080	67.03	10	0.190	10.13	20	0.108	14.97
CA	1.240s	0.770	0.705	0.616	0.107	0.101	86.80	3	0.170	10.62	5	0.108	15.65
PA	0.660s	0.780	0.701	0.625	0.112	0.109	86.64	3	0.180	10.69	5	0.115	15.64
lucent	0.063s	0.634	0.492	0.730	0.190	0.142	83.90	3	0.285	13.67	6	0.174	20.49
scan	0.117s	0.735	0.555	0.700	0.188	0.128	86.82	3	0.290	15.23	8	0.170	28.19
googleI	0.975s	0.615	0.376	0.774	0.164	0.096	75.49	4	0.211	12.04	16	0.125	28.78
googleO	0.960s	0.651	0.398	0.786	0.162	0.108	75.49	5	0.231	13.54	16	0.123	26.61
Avg		0.623	0.402	0.779	0.192	0.108	76.405		0.207	11.608		0.121	18.763

Table 5.5: The performance of our **dynamic** algorithms compared to linked lists. For each graph we give the space- and time-optimal block size. Space is in bits per edge; time is for a DFS, normalized to the first column, which is given in seconds.

a factor of 7.5 averaged over all the graphs (assuming the vertices are labeled with the separator ordering). The effect of insertion order has been previously reported (e.g. [84, page 268] and [36]) but the magnitude of the difference was surprising to us—the largest factor we have previously seen reported is about 4. We note that the magnitude is significantly less on the Pentium III with its smaller cache-line size (an average factor of 2.5 instead of 7.5). The actual insertion order will of course depend on the application, but it indicates that selecting a good insertion order is critical. We note, however, that if users can insert in linear order, then they are better off using one of the static representations, which allow insertion in linear order.

For our data structure the insertion order does not have any significant effect on performance. This is because the layout in memory is mostly independent of the insertion order. The only order dependence is due to hash collisions for the secondary blocks. Since each hash try is pseudo-random within the group, the location of the backup blocks has little effect on performance. In fact our experiments (not shown) showed no noticeable effect on DFS times for different insertion orders.

Overall the space optimal dynamic implementation is about a factor of 6.6 more compact than adjacency lists, while still being significantly faster than linked lists in most cases (up to a factor of 7 faster for randomly inserted edges). On the Pentium 4 linked lists with linear insertion and separator ordering take about 50% less time than our space-optimal dynamic representation and 10% less time than our time-optimal dynamic representation. On the Pentium III, linked lists with linear insertion and separator ordering take about a factor of 1.2 more time than our space optimal dynamic representation and 1.7 more time than our time optimal dynamic representation.

Times for insertion are reported in Tables 5.6 and 5.7.

Graph	DFS	Read		Find	Insert			Space
		Linear	Random	Next	Linear	Random	Transpose	
ListRand	1.000	0.099	0.744	0.121	0.571	28.274	3.589	76.405
ListOrdr	0.322	0.096	0.740	0.119	0.711	28.318	0.864	76.405
LEDARand	2.453	1.855	2.876	2.062	16.802	21.808	16.877	432.636
LEDAOrdr	1.119	0.478	2.268	0.519	7.570	20.780	7.657	432.636
DynSpace	0.633	0.440	0.933	0.324	14.666	23.901	15.538	11.608
DynTime	0.367	0.233	0.650	0.222	9.725	15.607	10.183	18.763
CachedSpace	0.622	0.431	0.935	0.324	2.433	28.660	8.975	13.34
CachedTime	0.368	0.240	0.690	0.246	2.234	19.849	6.600	19.073
ArrayRand	0.945	0.095	0.638	0.092	—	—	—	38.202
ArrayOrdr	0.263	0.092	0.641	0.092	—	—	—	38.202
Byte	0.279	0.197	0.693	0.205	—	—	—	12.501
Nibble	0.513	0.399	0.873	0.340	—	—	—	9.357
Snip	0.635	0.562	1.044	0.447	—	—	—	9.07
Gamma	0.825	0.710	1.188	0.521	—	—	—	9.424

Table 5.6: Summary of space and normalized times for various operations on the Pentium 4.

Graph	DFS	Read		Find	Insert			Space
		Linear	Random	Next	Linear	Random	Transpose	
ListRand	1.000	0.631	0.995	0.508	1.609	17.719	3.391	76.405
ListOrdr	0.710	0.626	0.977	0.516	1.551	17.837	1.632	76.405
LEDARand	3.163	2.649	3.038	2.518	17.543	19.342	17.880	432.636
LEDAOrdr	2.751	2.168	2.878	1.726	11.846	19.365	11.783	432.636
DynSpace	0.626	0.503	0.715	0.433	17.791	22.520	18.423	11.608
DynTime	0.422	0.342	0.531	0.335	13.415	16.926	13.866	17.900
CachedSpace	0.614	0.498	0.723	0.429	2.616	25.380	7.788	13.36
CachedTime	0.430	0.355	0.558	0.360	2.597	20.601	6.569	17.150
ArrayRand	0.729	0.319	0.643	0.298	—	—	—	38.202
ArrayOrdr	0.429	0.319	0.639	0.302	—	—	—	38.202
Byte	0.330	0.262	0.501	0.280	—	—	—	12.501
Nibble	0.488	0.411	0.646	0.387	—	—	—	9.357
Snip	0.684	0.625	0.856	0.538	—	—	—	9.07
Gamma	0.854	0.764	1.016	0.640	—	—	—	9.424

Table 5.7: Summary of space and normalized times for various operations on the Pentium III.

5.7.5 Timing Summary.

Tables 5.6 and 5.7 summarize the time complexity of various operations using the data structures we have discussed. For each structure we list the time required for a DFS, the time required to read all the neighbors of each vertex (examining vertices in linear or random order), the time required to search each vertex v for a neighbor $v + 1$, and the time required to construct the graph by linear, random, or transpose insertion. All times are normalized to the time required for a DFS on an adjacency list with random labeling, and the normalized times are averaged over all graphs in our dataset.

List refers to adjacency lists. LEDA refers to the LEDA implementation. For List, LEDA and Array, Rand uses a randomized ordering of the vertices and Ordr uses the separator ordering. The times for DFS, Read, and Find Next reported for List and LEDA are based on linear insertion of the edges (*i.e.*, this is the best case for them). Dyn refers to a version of our dynamic data structure that does not cache the edges for vertices in adjacency lists. Cached refers to a version that does. For the "DynSpace" and "CachedSpace" structures we used a space-efficient block size; for "DynTime" and "CachedTime" we used a time-efficient one. Array refers to adjacency arrays. Byte, Nibble, Snip and Gamma refer to the corresponding static representations.

Note that the cached version of our dynamic algorithm is generally slightly slower, but for the linear and transpose insertions it is much faster than the non-cached version. Those insertions are the operations that can make use of cache locality. For linear insertion our cached dynamic representations are a factor of 3-4 times slower than adjacency lists on the Pentium 4 and a factor of about 1.5 slower on the Pentium III.

LEDA is significantly slower and less space-efficient than the other representations, but as previously mentioned LEDA has many features these other representations do not have.

5.7.6 Randomized Graphs

To emphasize the fact that real-world graphs have good separators, we created randomized versions of several of the graphs in our test set. The randomized versions have the same vertex count, edge count, and degree distribution as the original graphs, but edges are randomly assigned as follows. Each vertex receives a number of slots equal to its degree in the previous graph. Every edge is randomly assigned two slots from the set of all empty slots. Duplicate edges and self-edges are discarded after the process is over; this reduces the edge count of the randomized graphs slightly.

For several graphs we compared the compression achieved on the original graph to that of the randomized graph. We compute the compression achieved with byte codes and snip codes, not counting the cost of an index. We compare this to the naive rate of $\log n$ bits per edge that could be achieved with a flat code. In all cases the compression is significantly worse when the graph is randomized.

The "Google" graph is an undirected version of googleI and googleO.

	Original		Random		
Graph	Byte	Snip	Byte	Snip	$\log n$
auto	9.58	6.31	20.37	26.43	18.78
ibm17	9.80	6.30	16.72	22.27	17.50
lucent	11.42	7.71	15.79	15.71	16.79
google	9.93	6.22	19.82	25.22	19.81

Table 5.8: The compression (in bits per edge) achieved with snip codes on various real-world graphs. When the edges in a graph are randomized to remove locality (but the degree distribution is maintained), the compression worsens significantly.

5.8 Algorithms

Here we describe results for two algorithms that might have the need for potentially very large graphs: Google's PageRank algorithm and a maximum bipartite matching algorithm. They are meant to represent a somewhat more realistic application of graphs than a simple DFS.

PageRank. We use the simplified version of the PageRank algorithm [95]. The algorithm involves finding the eigenvector of a sparse matrix $(1-\epsilon)A+\epsilon U$, where A is the matrix representing the link structure among pages on the web (normalized), U is the uniform matrix (normalized) and ϵ is a parameter of the algorithm. This eigenvector can be computed iteratively by maintaining a vector R and computing on each step $R_i = ((1-\epsilon)A + \epsilon U)R_{i-1}$. Each step can be implemented by multiplication of a vector by a sparse 0-1 matrix representing the links in A, followed by adding a uniform vector and normalizing across the resulting vector to account for the out degrees (since A needs to be normalized). The standard representation of a sparse matrix is the adjacency array as previously described. We compare an adjacency-array implementation with several other implementations.

We ran this algorithm on the Google out-link graph for 50 iterations with $\epsilon = .15$. For each representation we computed the time and space required. Figure 5.9 lists the results. On the Pentium III, our static representation with the byte code is the best. On the Pentium 4, the array with ordered labeling gives the fastest results, while the byte code gives good compression without sacrificing too much speed.

Bipartite Matching. The maximum bipartite matching algorithm is based on representing the graph as a network flow and using depth first search to find augmenting paths. It takes a bipartite graph from vertices on the left to vertices on the right and assigns a capacity of 1 to each edge. For each edge the implementation maintains a 0 or 1 to indicate the current flow on the edge. It loops through the vertices in the left set using DFS to find an augmenting path for each vertex. If it finds one it pushes one unit of flow through and updates the edge weights appropriately. Even though conceptually the graph is directed, the implementation needs to maintain edges in both directions to implement the depth-first search. To avoid an $\Omega(n^2)$ best-case runtime, a stack was used to store the vertices visited by each DFS so that the entire bit array of visited vertices did not need to be cleared each time. This optimization is suggested in the LEDA book [84, page 372]. We also implemented an optimization that does one level of BFS before the DFS. This improved performance by

Representation	Time (sec) PIII	Time (sec) P4	Space (b/e)
Dyn-B4	30.40	11.05	17.54
Dyn-N4	32.96	12.48	13.28
Dyn-B8	26.55	9.23	19.04
Dyn-N8	30.29	11.25	15.65
Gamma	38.56	15.60	9.63
Snip	34.19	13.38	9.43
Nibble	26.38	10.94	9.72
Byte	21.09	8.04	12.59
ArrayOrdr	21.12	6.38	37.74
ArrayRand	33.83	27.59	37.74
ListOrdr	30.96	6.12	75.49
ListRand	44.56	28.33	75.49

Table 5.9: Performance of our PageRank algorithm on different representations.

40%. Finally we used a strided loop through the left vertices, using a prime number (11) as the stride. This reduced locality, but greatly improved performance since the average depth of the DFS to find an unmatched pair was reduced signficantly.

Since the graph is static the static representations are sufficient. We ran this algorithm using our byte code, nibble code, and adjacency array implementations. The bit array for the 0/1 flow flags is accessed using the same indexing structure (semidirect-16) as used for accessing the adjacency lists. A dynamically sized stack is used for the DFS and for storing the visited vertices during a DFS. We store 1 bit for every edge (in each direction) to indicate the the current flow, 1 bit for every vertex to mark visited flags, and 1 bit for every vertex on the right to mark whether it is matched.

The maximum bipartite matching algorithm was run on a modified version of the Google-out graph. Two copies were created for each vertex, one on the left and one on the right. The out links in the Google graph point from the left vertices to the right ones. The results are given in Figure 5.10. The memory listed is the total memory including the representation of the graph, the index for 0/1 flow flags, the flow flags themselves, the visited and matched flags and the stacks. For all three representations we assume the same layout for this auxiliary data, so the only difference in space is due to the graph representation. The space needed for the two stacks is small since the largest DFS involves under 10000 vertices.

5.9 Discussion

Here we summarize what we feel are the most important or surprising results of the experiments.

First we note that the simple and fast separator heuristic we used seems to work very well for our purposes. This is likely because the compression is much less sensitive to the quality of the separator than other applications of separators, such as nested dissection [80]. For nested dissection more sophisticated separa-

Representation	Time (sec) PIII	P4	Space (b/e)
Nibble	75.8	27.6	13.477
Byte	59.9	19.9	16.363
ArrayOrdr	57.1	18.6	41.678
ArrayRand	83.2	28.0	41.678

Table 5.10: Performance of our bipartite maximum matching algorithm on different static representations.

tors are typically used. It would be interesting to study the theoretical properties of the simple heuristic. For our bounds rather sloppy approximations on the separators are sufficient since any separator of size kn^c, $c < 1$ will give the required bounds, even if actual separators might be much smaller.

We note that all the "real-world" graphs we were able to find had small separators—much smaller than would be expected for random graphs. This is a property of real world graphs that is sometimes not properly noted.

Our experiments indicate that the additional cost needed to decode the compressed representation is small or insignificant compared to other costs for even an algorithm as simple as depth-first search. As noted, under most situations the compressed representations are faster than standard representations even though many more operations are needed for the decoding. This seems to be because the performance bottleneck is accessing memory and not the bit operations used for decoding. The one place where the standard representations are slightly faster for DFS is when using separator orderings and linear insertion on the Pentium 4.

We were somewhat surprised at the large effect that different orderings had on the performance on the Pentium 4 for both adjacency lists and adjacency arrays. The performance differed by up to a factor of 11, apparently purely based on caching effects (the number of edges traversed is identical for any DFS on a fixed graph). The differences indicate that performance numbers reported for graph algorithms should specify the layout of memory and ordering used for the vertices. The differences also indicate that significant attention needs to be paid to vertex ordering in implementing fast graph algorithms. We note that the same separator ordering as used for graph compression seems to work very well for improving performance on adjacency lists and adjacency arrays. This is not surprising since both compression and memory layout can take advantage of locality in the graphs so that most accesses are close in the ordering.

In our analysis we do not consider applications that have a significant quantity of information that needs to be stored with the graphs, such as large weights on the edges or labels on vertices. Clearly such data might diminish the advantages of compressing the graph structure. We note, however, that such data might also be compressed. In fact the locality of the separator labeling could be useful for such compression. For example on the web graphs, vertices nearby in the vertex ordering are likely to share a large prefix of their URL. Similarly, for the finite-element meshes, vertices nearby in the vertex ordering are likely to be nearby in space, and hence might be difference encoded.

The ideas used in this chapter can clearly be generalized to other structures beyond simple graphs. For example, the reordering-via-separator idea is used in Chapter 6, and the simplicial mesh data structure of

Chapter 7 is also based on difference coded adjacency lists.

Our work could be adapted to an out-of-core setting. The decrease in total memory usage from compression is not so important in an out-of-core setting; however, by compressing the data we can increase the amount that can fit in cache. The locality provided by our reordering could also be very useful to an out-of-core algorithm. One complication that could arise involves the algorithm to perform the reordering: most of the reordering algorithms we present assume that the graph structure can be held in RAM. This problem could be addressed by using a crude partitioning algorithm at the high levels, then a more sophisticated reordering algorithm the subgraphs reached a manageable size.

Chapter 6

Index Compression through Document Reordering

6.1 Introduction

In this chapter we are interested in the compression of *inverted indices*.[1] An inverted index is a collection of *posting lists*, each of which is a subset of the set $U = \{1 \ldots m\}$. The compressed posting lists must be stored individually (since they may need to be accessed individually). However, by using properties of the index as a whole it is possible to improve the compression of individual lists. This chapter describes a heuristic relabeling technique: it uses a permutation to relabel the elements of U in order to improve the compression of posting lists.

There are many possible representations for compressed posting lists (some were discussed in Chapter 4). However, in this chapter our focus is on the quality of compression achievable for an index. Accordingly, in this chapter we consider compression using the difference coded representation discussed in Section 2.4, which is more compact than those of Chapter 4. The reordering technique we describe would apply to the structures of Chapter 4 as well.

Compact inverted indices are very important in the design of search engines, where memory considerations are a serious concern. Some web search engines index billions of 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 which maps search terms to lists of documents containing those terms. Each 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* (as described in Section

[1]This chapter is based on work with Guy Blelloch [12].

2.4).

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 [87, 23] 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.

Here, we will study 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. Our 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. This is similar to the graph-reordering algorithm of Chapter 5.

We have implemented this idea and tested it on indexing data from the TREC-8 ad hoc track [129] (disks 4 and 5, excluding the Congressional Record). We tested a variety of codes in combination with difference coding. Our algorithm was able to improve the performance of the best compression technique we found by fourteen percent simply by reordering the document numbers. The improvement offered by our algorithm increases with the size of the index, so we believe the improvement on larger real-world indices would be greater.

Conceptually, our ORDER-INDEX algorithm is divided into three parts. The first part, BUILD-GRAPH, constructs a document-document similarity graph from an index. The second part, SPLIT-INDEX, makes calls to the Metis [71] graph partitioning package to recursively partition the graphs produced by BUILD-GRAPH. It uses these partitions to construct a hierarchical clustering tree for the index. The third part of our algorithm, ORDER-CLUSTERS, applies rotations to the clustering tree to optimize the ordering. It then numbers the documents with a simple depth-first traversal of the clustering tree. At all levels we apply optimizations and heuristics to ensure that the time and memory requirements of our algorithm will scale well.

In practice, constructing the full hierarchical clustering would be infeasible, so the three parts of our algorithm are combined into a single recursive procedure that makes only one pass through the clustering tree.

Related Work. When this research was originally published [12], there was no previous work dealing with index compression by reordering. Since then several other authors have examined the subject. Shieh et al. [113] published concurrently, presenting a reassignment method based on the Traveling Salesman Problem. They achieved 15% compression over the original ordering, but the reordering phase took much longer: they reorder 1.8 documents/second on a collection of 130k documents, whereas we achieve 300 documents/second and 13% compression on a similar collection.

Blanco and Barreiro [11] use heuristics to improve the TSP-based algorithm (most notably, they use dimensionality reduction through singular value decomposition). They present an algorithm achieving 6%

to 8% compression on two collections of 130k documents, with a speed of about 125 documents/second. It is unclear how their algorithm's RAM requirements compare to ours. We note, however, that the largest index they test is 130k, whereas Silvestri et al. [115] were able to use our algorithm to reorder over $600,000$ documents in 1GB of RAM. (Blanco and Barreiro incorrectly claim that the above figure was only $60,000$ documents.)

Silvestri et al. [115] presented several algorithms which were much faster and used only two-thirds as much memory as our algorithm (using our default settings), but their best algorithm achieved only two-thirds as much compression as ours. We note that their compression quality deteriorates for larger indices (whereas our algorithm achieves better compression on larger indices).

The remainder of this chapter is organized as follows. Section 6.2 formalizes the problem. Section 6.3 describes our algorithm in detail. Section 6.4 demonstrates the performance of our algorithm when run on the TREC-8 database.

6.2 Definitions

We describe an inverted index I as a set of terms $t_1 \ldots t_m$. For every term t_i there is an associated list of $|t_i|$ document numbers $d_{i,1} \ldots d_{i,|t_i|}$. The document numbers are in the range $1 \leq d_{i,j} \leq n$. We are interested in the cost of representing these documents using a difference code. Thus we define, first, $s_i(d_i) = s_{i,1} \ldots s_{i,|t_i|}$ to be the sequence of documents $d_{i,j}$, rearranged so that $s_{i,j} < s_{i,j+1}$ for all j. That is, s_i is the sorted version of the sequence of documents d_i. (For convenience we also define $s_{i,0} = 0$ for all i.) Then, if we have an encoding scheme c which requires $c(d)$ bits to store the positive integer d, we can write the cost C of encoding our index as follows:

$$C(I) = \sum_{i=1}^{n} \sum_{j=1}^{|t_i|} c(s_{i,j} - s_{i,j-1})$$

We wish to minimize $C(I)$ by creating a permutation σ which reorders the document numbers. Since c is convex for most useful encoding schemes, this means we need to cluster the documents to improve the locality of the index.

6.3 Our Algorithm

Document Similarity. Up to this point, we have viewed an inverted index as a set of terms, each of which contains some subset of the documents. Now it will be convenient to consider it as a set of documents, each of which contains some subset of the terms. Specifically, we can consider a document to be an element of $\{0,1\}^m$, where the i^{th} element of a document is 1 if and only if the document contains term t_i. Eventually our algorithm will need to compute centers of mass of groups of documents, and then it will be convenient to allow documents to contain fractional amounts of terms—that is, to represent a document as an element of $\Re^m$.

Our algorithm uses the *cosine measure* to determine the similarity between a pair of documents:

$$\cos(A, B) = \frac{A \cdot B}{((A \cdot A)(B \cdot B))^{\frac{1}{2}}}$$

Build-Graph. Using this similarity measure our BUILD-GRAPH algorithm can construct a document-document similarity graph. For large databases, creating a full graph with n^2 edges is not feasible. However, most of the documents contain only a small fraction of the total set of terms. It seems reasonable that the graph might be sparse: many of the edges in the similarity graph might actually have a weight of zero. This is especially true if we remove common "stopwords" from consideration, as described below.

To save space, BUILD-GRAPH uses the following method to generate the graph. Consider the index to be a bipartite document-term graph from which BUILD-GRAPH needs to generate a document-document graph. For each term in the document-term graph, our algorithm eliminates that term and inserts edges (weighted with the cosine measure) to form a clique among that node's neighbors. After eliminating all of the terms in the document-term graph, BUILD-GRAPH has produced a document-document graph which contains an edge between every pair of documents that shares a common term.

If term t_i contains $|t_i|$ documents, then BUILD-GRAPH will compute $O(\sum |t_i|^2)$ cosine measures in computing the edge graph. Our algorithm can improve this bound slightly by being careful never to compute the same cosine measure more than once, but the worst-case number of cosine measures will still be $O(\sum |t_i|^2)$.

However, BUILD-GRAPH does not actually need all this information in order to represent the structure of the similarity graph. In particular, a lot of the documents in the index are likely to be "trivially" similar because they share terms such as "a", "and", or "the". The most frequently occurring terms in the index are also the least important to the similarity measure. Removing the edges corresponding to those terms should not have much of an impact on the quality of the ordering (as demonstrated in Section 6.4), while it should decrease the work required for BUILD-GRAPH considerably. Thus, when generating the graph, our algorithm creates cliques among the neighbors of only those terms with less than a threshold number of neighbors τ. All other terms are simply deleted from the graph. (This technique is similar to that used by Broder et al. [27] for identifying near-duplicate web pages.) Pseudocode for this part of our algorithm is shown in Figure 6.1.

Split-Index. Once BUILD-GRAPH has produced a similarity graph, the next step is to derive a hierarchical clustering of that graph. There are a large number of hierarchical clustering techniques that we could choose from (for example, those of [24, 135], and additional references from those papers), but most of those techniques are not designed for data sets as large as the ones we are dealing with. Many of them, in fact, require as input an $O(n^2)$ similarity matrix. We do not have enough space or time to even construct a matrix of that size, much less run a clustering algorithm on it. Furthermore, this problem has certain special features which are not captured by any general clustering algorithms. Therefore we have created our own hierarchical clustering algorithm based on graph partitioning.

A naive hierarchical clustering algorithm would work as follows. Given an index, compute a similarity graph from that index. Partition the graph into two pieces. Continue partitioning on the subgraphs until all

```
SPLIT-INDEX(I):
  I' ← SUBSAMPLE(I, |I|^ρ)
  G ← BUILD-GRAPH(I')
  (G1, G2) ← PARTITION(G)
  d1 ← G1.centerofmass
  d2 ← G2.centerofmass
  I1, I2 ← empty indices
  foreach d in I
    if COS(d, d1) > COS(d, d2) then
      add d to I1
    else
      add d to I2
  return (I1, I2)
```

```
BUILD-GRAPH(I):
  G ← new Graph
  foreach d1 in I
    foreach t in d1
      if |t| ≤ τ then
        foreach d2 in DOCLIST(t)
          e ← new Edge(d1, d2, COS(d1, d2))
          add e to G
  return G
```

Figure 6.1: Our BUILD-GRAPH and SPLIT-INDEX algorithms.

pieces are of size one. Use the resulting partition tree as the clustering hierarchy.

Unfortunately, this algorithm uses too much memory. Our BUILD-GRAPH algorithm requires less than the full $O(n^2)$ memory, but it is still infeasible to apply it to the full index. Instead, SPLIT-INDEX uses a sampling technique on the index: at each recursive step, it subsamples some fraction of the documents from the original index. It runs BUILD-GRAPH on this subindex and partitions the result. Once it has done this, SPLIT-INDEX uses the subgraph partition to partition the original index. To do this, it computes the centers of mass of the two subgraph partitions. It then partitions the documents from the original index based on which of the centers of mass they are nearest to. Pseudocode for SPLIT-INDEX is shown in Figure 6.1.

An interesting point to note is that SPLIT-INDEX recreates the document similarity graph at each node of the recursion tree. This offers our BUILD-GRAPH algorithm significantly more flexibility when creating the similarity graph: BUILD-GRAPH only needs to create the graph in such a way that the *first* partition made on it will be a good one. This allows BUILD-GRAPH to use a very small value of τ: if a term occurs more than, say, 10 times at a given partition level, it is likely that any partition SPLIT-INDEX computes will have documents containing this term on both sides anyway. Thus BUILD-GRAPH ignores that term until later iterations.

Order-Clusters. Once SPLIT-INDEX has produced a hierarchical clustering, ORDER-INDEX uses that clustering to create a numbering of the leaves. To do this it performs an inorder traversal of the tree. At each step, however, it needs to decide which of the two available partitions to traverse first. In essence, our ORDER-CLUSTERS algorithm looks at every node in the hierarchy and decides whether or not to swap its children.

Within any given subtree S, there are four variables to consider. We denote the children of S by I_1 and I_2. We also define I_L and I_R to be the documents that will appear to the immediate left and right of S in the final ordering. (At the first recursion we initialize I_L and I_R to place equal weight on each term. This causes infrequently-occurring terms to be pulled away from the middle of the ordering.) Since ORDER-CLUSTERS operates with a depth-first traversal, we take I_L to be the left child of S's left ancestor, and I_R to be the right child of S's right ancestor. ORDER-CLUSTERS tracks the centers of mass of each of these clusters, and it

ORDER-CLUSTERS(I_L, I_1, I_2, I_R):
  $m_L \leftarrow I_L.centerofmass$
  $m_1 \leftarrow I_1.centerofmass$
  $m_2 \leftarrow I_2.centerofmass$
  $m_R \leftarrow I_R.centerofmass$
  $s_1 \leftarrow$ COS(m_L, m_1) * COS(m_R, m_2)
  $s_2 \leftarrow$ COS(m_L, m_2) * COS(m_R, m_1)
  if $s_2 > s1$ **then**
    return (I_2, I_1)
  else
    return (I_1, I_2)

\\ Assigns the numbers between ℓ and h
\\ to the documents of an index I,
\\ which must have exactly $(h - \ell + 1)$
\\ documents.
ORDER-INDEX(I, ℓ, h, I_L, I_R):
  if $\ell = h$ **then**
    $I.v|0|.number \leftarrow \ell$
  else
    (I_1, I_2) $\leftarrow$ SPLIT-INDEX(I)
    (I_1, I_2) $\leftarrow$ ORDER-CLUSTERS(I_L, I_1, I_2, I_R)
    ORDER-INDEX(I_1, ℓ, $m - 1$, I_L, I_2)
    ORDER-INDEX(I_2, m, h, I_1, I_R)

Figure 6.2: Our ORDER-CLUSTERS and ORDER-INDEX algorithms.

rotates I_1 and I_2 so as to place similar clusters closer together.

Pseudocode for ORDER-CLUSTERS and for the main body of our algorithm is shown in Figure 6.2.

6.4 Experimentation

Compression Techniques. We tested several common difference codes to see how much improvement our algorithm could provide. The codes we tested include the delta code, Golomb code [54], and arithmetic code. These codes are described in more detail by Witten, Moffat, and Bell in [136]. We also tested the binary interpolative compression method of Moffat and Stuiver [87]. This code was explicitly designed to exploit locality in inverted indices, so it gained the most from our algorithm.

We did not count the cost of storing the sizes of each term since that cost would be invariant across all orderings. We did count the cost of storing an arithmetic table for arithmetic coding, but this cost was negligible compared to the cost of storing the bulk of the data.

Testing. To test our algorithm we used the ad-hoc TREC indexing data, disks 4 and 5 (excluding the Congressional Record). This data contained 527094 documents and 747990 distinct words, and occupied about one gigabyte of space when uncompressed. We tested three different orderings of the data in combination with the difference codes described above. First, we tested a random permutation of the document numbers as a baseline for comparison. Second, we tested the default ordering from the TREC database. We noted that this was already a significant improvement over a random ordering, indicating that there is considerable locality inherent in the TREC database. Third, we tested the ordering produced by our algorithm. Results are shown in Figure 6.3.

Analysis. The Golomb code is near-optimal for the encoding of randomly distributed data, and in fact it was the best code for the Random ordering. However, the Golomb code is not convex, so it does not benefit from locality.

	Random	Identity	Ordered
Binary	20.0	20.0	20.0
Delta	7.52	6.46	5.45
Golomb	5.79	5.77	5.78
Arith	6.82	6.03	5.19
Interp	5.89	5.29	4.53

Figure 6.3: The improvement (in bits per edge) our algorithm offers for different coding schemes using disks 4 and 5 of the TREC database.

Index Size	Random	Identity	Ordered	Improvement over Random	Improvement over Identity
32943	5.73	5.44	4.87	14.9%	10.4%
65886	5.75	5.43	4.78	16.9%	12.0%
131773	5.77	5.41	4.71	18.4%	13.0%
263547	5.78	5.36	4.63	19.9%	13.7%
527094	5.79	5.29	4.53	21.8%	14.4%

Figure 6.4: The improvement offered by our algorithm increases as the size of the index (measured in documents) increases.

The locality inherent in the TREC database made the interpolative code the most efficient code for the identity ordering. Interpolative coding used 5.29 bits per edge, an improvement of about 8.6% over the best encoding with a random document ordering.

Using the ordering produced by our algorithm, however, the interpolative code needed an average of only 4.53 bits per edge to encode the data - a 21.8% improvement over the best coding of a random ordering, and a 14.4% improvement over the best coding of an identity ordering.

Index size. To measure the effect of index size on our algorithm, we tested our algorithm on various subsets of the full index. These subsets were formed by evenly subsampling documents from the full dataset. For each subset we evaluated the best compression using a random, identity, or ordered permutation of the documents. The random permutation was best coded with a Golomb code; the identity and ordered permutations were coded with interpolative codes. Figure 6.4 shows the results of our tests. Interestingly, the improvement offered by our algorithm increases as the size of the index increases.

Parameter Tuning. Our algorithm uses two parameters. The first parameter, τ, is a threshold which determines how sensitive our BUILD-GRAPH algorithm is to term size. If a term t_i has $|t_i| > \tau$, our algorithm will still consider it when calculating cosine measures, but will not add any edges to the similarity graph because of it.

Table 6.5 shows the performance of our algorithm (on a subset of the full dataset containing one-

		τ						
	Rand	1	2	5	10	15	20	40
Time(s)		51.81	81.62	120.7	163.7	196.4	225.0	304.8
Delta	7.44	6.56	6.04	5.95	5.94	5.90	5.89	5.88
Arith	6.73	6.11	5.69	5.62	5.61	5.58	5.57	5.56
Interp	5.81	5.29	4.97	4.89	4.87	4.86	4.86	4.85

Figure 6.5: The performance (in bits per edge) of different values of τ on one-sixteenth of the TREC indexing data.

		ρ				
	Rand	.75	.5	.25	.1	0
Time(s)		70.07	60.59	163.7	454.8	518.9
Delta	7.44	6.27	6.05	5.94	5.83	5.83
Arith	6.73	5.88	5.70	5.61	5.52	5.52
Interp	5.81	5.07	4.95	4.87	4.83	4.84

Figure 6.6: The performance (in bits per edge) of different values of ρ on one-sixteenth of the TREC indexing data. Note that our algorithm's running time is greater with $\rho = .75$ than with $\rho = .5$. This is because the aggressive subsampling results in unbalanced partitions, increasing the recursion depth of the algorithm.

sixteenth as many documents) with different values of τ. Choosing τ to be less than 5 causes too few edges to be included in the similarity graph, but increasing τ beyond that was not beneficial on the index we studied. We chose $\tau = 10$ to be safe.

The second parameter, ρ, determines how aggressively our algorithm subsamples the data. On an index of size n, the algorithm extracts one out of every $\lfloor n^\rho \rfloor$ elements to build a subindex. Table 6.6 shows the performance of our algorithm with different values of ρ. Our algorithm does not perform too badly even with a very large ρ, but there is still a clear tradeoff between time, space and quality. We chose $\rho = .25$ in our experiments as a suitable balance between these concerns.

Graph Compression. Our algorithm can also be used to enhance the performance of difference coding in graph compression. (Chapter 5 discussed separator algorithms for graphs which can be manipulated within main memory. This algorithm, with its subsampling techniques, can be applied to graphs which are much larger; however, the compression produced is weaker.) In graph compression, for each vertex of the graph, an algorithm stores an adjacency list of the vertices that share an edge with that vertex. The vertices are numbered, so it is only natural to apply a difference code to compress each list. If we view the vertices as terms and the adjacency lists as posting lists, we can apply our clustering technique to renumber the vertices of the graph.

To test our clustering technique on graph data we used another TREC dataset: the TREC-8 WT2g web

Code	Random	Identity	Clustered	Ordered
Binary	18.0	18.0	18.0	18.0
Delta	17.5	4.92	4.58	4.52
Golomb	13.3	12.7	12.4	12.6
Arith	14.4	4.32	3.82	3.75
Interp	13.4	5.83	5.66	5.58

Figure 6.7: The performance of our algorithm on the TREC-8 WT2g web track. The "Clustered" column describes the performance of our algorithm without the final rotation step.

data track. That track can be represented as a directed graph on 247428 web pages, where hyperlinks are edges. For best compression, we stored the in-edges (rather than the out-edges) of each vertex in our adjacency lists. The number of in-edges for each vertex was more variable than the number of out-edges, meaning that some adjacency lists were very dense and thus compressed very well. The performance of our algorithm on the in-link representation is shown in Table 6.7.

Chapter 7

Compact Representations of Simplicial Meshes in Two and Three Dimensions

7.1 Introduction

In this chapter we are interested in compressed representations of meshes that permit dynamic queries and updates to the mesh.[1] The goal is to solve larger problems while using standard random-access main-memory algorithms. We present data structures for representing two and three dimensional simplicial meshes. (By a d simplicial mesh we mean a pure simplicial complex of dimension d, which is a manifold, possibly with boundary [49].) The data structures 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 [85] with bounded degree, these operations each take constant time. Although our data structures are not as compact as those designed for disk storage, they still save a factor of between 5 and 10 over standard representations.

Compressed meshes are very important: For many applications the space required to represent large unstructured meshes in memory can be the limiting factor in the size of a mesh. Standard representations of tetrahedral meshes, for example, can require 300-500 bytes per vertex. There has been previous work which deals with larger meshes by maintaining the mesh in external memory. To avoid thrashing, this requires designing algorithms for which the access to the mesh is carefully orchestrated. Although several such external memory algorithms have been designed [55, 45, 43, 83, 128, 7, 124, 5], these algorithms can be much more complicated than their main-memory counterparts, and can be significantly slower.

The field of compressed meshes has received considerable attention [44, 61, 121, 98, 105, 120, 70, 66, 53]. In three dimensions, for example, these methods can compress a tetrahedral mesh to less than a byte per tetrahedron [120]—about 6 bytes/vertex (not including vertex coordinates). 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.

[1] This chapter is based on work with Guy Blelloch, David Cardoze and Clemens Kadow [14].

Our data structures are described in Section 7.3 and 7.4. They take advantage of the separator properties of well-shaped meshes [85] and make use of our results in graph compression (see Chapter 5). In particular our technique uses separators to relabel the vertices so that vertices that share a simplex are likely to have labels that are close in value. Pointers are then difference encoded using variable length codes. For the 2D case we present two mesh representations. One representation is based on storing, for each edge, the triangles that contain that edge. This is described in Section 7.3. The other representation is based on radially storing the neighboring vertices around each vertex. This is described in Section 7.4. Both of our representations generalize readily to 3D and greater dimensions.

Section 7.5 describes an implementation of our data structure using the representation that stores the neighboring vertices for each vertex. Section 7.6 presents experimental results. The implementation 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. We present experiments based on using our representation as part of incremental Delaunay algorithms in both 2D and 3D. We use a variant of the standard Bowyer-Watson algorithm [25, 131] and the exact arithmetic predicates of Shewchuk [111] for all geometric tests. We also present experiments based on a Delaunay refinement algorithm that removes triangles with small angles by adding new points at their circumcenters. All space is reported in terms of the total space including the space for the vertex coordinates and all other data structures required by the algorithm. The results for 1 Gbyte of memory are summarized as follows.

- We can generate a 2D Delaunay mesh with 110 million triangles (.47 Gbytes for the mesh, .44 Gbytes for the vertex coordinates, and about .1 Gbytes for auxiliary data used by the algorithm). Compared to the Triangle code [110] (the most efficient we know of) our algorithm uses a factor of 3 less memory. It is about 10% slower than Triangle's divide-and-conquer algorithm and much faster than its incremental algorithm.
- We can generate a 3D Delaunay mesh with 100 million tetrahedra (.75 Gbytes for the mesh, .17 Gbytes for the vertex coordinates, and .08 Gbytes for auxiliary data). Compared to the Pyramid code [109], our algorithm uses a factor of 3.5 less memory, and is about 30% faster.
- We can generate a refined 2D Delaunay mesh with 80 million triangles with no angle less than 26%. This version dynamically generates new labels, and uses an extra level of indirection in our data-structure.

Our data structure can be used in conjunction with external memory algorithms. Also, although we describe our implementation only for 2D and 3D simplicial meshes, the ideas extend to higher dimensions. These topics are discussed, briefly, in Section 7.7.

7.2 Standard Mesh Data Structures

There have been numerous approaches for representing unstructured meshes in 2 and 3 dimensions. Some are specialized to simplicial meshes and others can be used for more general polytope meshes. For the purpose of comparing space usage, we review the most common of these data structures here. A more complete comparison for 2D structures can be found in a paper by Kettner [73].

In two dimensions most approaches are based on either triangles or edges. The simplest data structure is based on triangles. Each triangle has three pointers to the neighboring triangles, and three pointers to its vertices. Assuming no data needs to be stored on triangles or edges, this data structure uses 6 pointers per triangle. Storing data requires extra pointers. Shewchuk's Triangle code [110] and the CGAL 2D triangulation data structure [21] both use a triangle-based data structure. To distinguish the three neighbors/vertices of a triangle, a handle to a triangle typically needs to include an index from 1 to 3. The data structure used by Triangle, for example, includes such an index in the pointer to each neighbor (in the low 2 bits) so that a neighbor query not only returns the neighbor triangle, but returns in which of three orders it is held.

There are many closely related data structures based on edges, including the doubly connected edge list [89], winged-edge [9], half-edge [133], and quad-edge [59] structures. In addition to triangulated meshes, these data structures can all be used for polygonal meshes. In these data structures each edge maintains pointers to its two neighboring vertices and to neighboring edges cyclically around the neighboring faces and vertices. Each edge might also maintain pointers to the neighboring faces and to edge data. The most space efficient of these data structures can maintain for each edge a pointer to the two neighboring vertices and to just two neighboring edges, one around each face and vertex. Assuming no data needs to be stored on a face or edge, this requires 4 pointers per edge, which for a manifold triangulation is equivalent to the 6 pointers per triangle used by the triangle structure ($|E| = 3/2|T|$). The half-edge data structure [133], used by CGAL [73], LEDA [84] and HGAM [58], maintains two structures per edge, one in each direction. These half-edges are cross referenced, requiring an extra two pointers per edge. The winged-edge and quad-edge structures maintain pointers to all four neighboring edges, requiring 6 pointers per edge (9 per triangle).

In three dimensions there are analogous data structures based either on tetrahedra or on faces and edges. Again the simplest data structure is to use a structure per tetrahedron. Each tetrahedron has 4 pointers to adjacent tetrahedra, and 4 to its corner vertices. Assuming no data this requires 8 pointers per tetrahedron. This data structure is used by Pyramid [109] and CGAL [21]. The face and edge data structures are often called boundary representations (b-reps). Such boundary representations are more general than the tetrahedron data structures, allowing the representation of polytope meshes, but tend to take significantly more space. Dobkin and Laszlo [48] suggest a data structure based on edge-face pairs, which in general requires 6 pointers per edge-face. For tetrahedral meshes this data structure can be optimized to 9 pointers per face (6 to the adjacent faces rotating around its 3 edges, and 3 to the corner vertices). This corresponds to 18 pointers per tetrahedron. Weiler's radial-edge representation [134], Brisson's cell-tuple representation [26], and Lienhardt's G-map representation [79] all take more space.

In summary, the most efficient standard data structures of simplicial meshes use 6 pointers per triangle in 2D and 8 pointers per tetrahedron in 3D. At least one extra pointer is required to store data on triangles in 2D or tetrahedra in 3D.

7.3 Representation Based On Edges

In this section we discuss our 2D representation based on edges. The representation is very similar to the graph representation from Section 5.4, although its 3D generalization is somewhat different. We begin with an uncompressed representation, and then describe how to compress it.

Each edge (a, b) in a 2D mesh M can be a part of at most two faces (triangles). If the faces are (a, b, c) and (b, a, d), then our representation stores the (key, data) pair $((a, b), (c, d))$ in a dictionary structure. (This is similar to the winged-edge structure of Baumgart [9], except that all references are to vertex labels rather than pointers.) If the orientation of the mesh needs to be maintained, then the vertices c, d are kept in a consistent order; otherwise the order does not matter. If one of the faces is missing, the vertex for that face is replaced with a special token 0: $((a, b), (c, 0))$.

To save space, edges are kept in a consistent direction: if the dictionary stores (a, b), then it does not also store (b, a). The proper direction for an edge is determined by some simple test (for example, all edges are stored as (a, b) where $a < b$).

Our structure supports the operations `search`, `insert`, and `delete`, as follows:

`search`(a, b)**:** finds all vertices c such that (a, b, c) form a face in M. This is a single dictionary lookup.

`insert`(a, b, c)**:** adds the face (a, b, c) to M. This requires updating the dictionary entries for (a, b), (b, c), and (c, a). If an entry is already in the dictionary, its 0 token is replaced with the appropriate vertex; otherwise, the entry is created with a 0 token.

`delete`(a, b, c)**:** deletes the face (a, b, c) from M. This requires updating the dictionary entries for (a, b), (b, c), and (c, a) by replacing the appropriate vertices with 0 tokens. If an entry has 0 tokens in both its data slots, it is deleted.

This interface supports traversing the mesh by repeated invocations of `search`. The variable-bit-length dictionary structure allows us to support `search` in $O(1)$ time and `insert` and `delete` in $O(1)$ expected amortized time. Data can be stored on the mesh by including it in the dictionary entries.

To compress this data structure we use difference coding to encode b, c, and d relative to a. That is, in our dictionary we store tuples of the form $((a, b-a), (c-a, d-a))$. The differences $b-a$, $c-a$, and $d-a$ are gamma coded (as described in Section 2.3); a sign bit indicates whether each difference is negative.

We use a variable-bit-length dictionary to store the encoded entries, as described in Section 3.3. The dictionary absorbs the $O(\log |V|)$-bit cost of representing a. It remains to account for the gamma coded differences $b-a$, $c-a$, $d-a$. We charge the cost of storing $a-b$ to the edge (a, b), the cost of storing $a-c$ to the edge (a, c), and the cost of storing $a-d$ to the edge (a, d). Each edge (a, b) is charged at most five times, and the cost in each case is $O(\log |a-b|)$. This gives us:

Theorem 7.3.1 *Our 2D simplicial mesh representation using our variable-bit dictionary uses $O(\sum_{(a,b)\in E} \log |a-b|)$ bits where E is the $1-$skeleton (that is, the set of edges) of the mesh.*

If the vertices of the mesh are given a k-compact labeling (as described in Section 5.2), then the representation of the mesh will use $O(|V|)$ bits. We note that well-shaped meshes (of fixed dimension) with bounded degree have small separators [85], and so the separator-tree algorithm from Section 5.2 is guaranteed to find a k-compact labeling. Further, 2D meshes are planar and thus have small vertex separators [81]; 2D meshes with bounded degree have small edge separators as well.

Although the representation permits dynamic insertions and deletions, the $O(|V|)$-bit space bound depends on the labeling remaining k-compact. For this reason we describe our representation as *semidynamic*, similar to our graph representations from Sections 5.3 and 5.4.

In practice for well-shaped meshes, it is possible to find a good labeling by taking advantage of the spatial embedding of the vertices. Rather than edge separators, our algorithms make use of $x - y$ cuts to partition the vertices for relabeling. Since edges in most meshes have high locality, this gives a labeling which is very good in practice. More details are given in Section 7.5.

Generalization to 3D. Our representation has a natural generalization to 3D based on storing faces (triangles) of the mesh. Each face (a, b, c) in M can be a part of at most two tetrahedra. If the tetrahedra are (a, b, c, d) and (a, c, b, e), then our representation stores the tuple ((a,b,c), (d,e)) in a dictionary structure. To save space, each face (a, b, c) is stored using only one ordering—for example, the ordering in which $a < b < c$.

The operations `search`, `insert`, and `delete` are supported just as in the 2D case. The data structure is compressed by difference coding: rather than ((a,b,c), (d,e)), the structure stores ((a,b-a,c-a), (d-a,e-a)) in a variable-bit-length dictionary.

We now examine the space usage of our 3D representation. The dictionary absorbs the $\log |V|$-bit cost of representing a using quotienting. 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). Each face is charged $O(1)$ times, and each time the charge is $O(\max(\log |a-b|, \log |a-c|, \log |b-c|)) = O(\log |a-b| + \log |a-c| + \log |b-c|)$. This gives a bound of $O(\sum_{(a,b,c)\in F} \log |a - b| + \log |a - c| + \log |b - c|)$, where F is the 2-skeleton (that is, the set of faces) of the mesh.

In fact we can prove a stronger bound on the space usage:

Lemma 7.3.1 *Let F be the $2-$skeleton (the set of faces) of a 3D simplicial mesh. Let E be the $1-$skeleton (the set of edges) of the mesh. Then any mesh representation with space usage $O(\sum_{(a,b,c)\in F} \log |a - b| + \log |a - c| + \log |b - c|)$ bits also has space usage $O(\sum_{(a,b)\in E} \log |a - b|)$ bits.*

Proof. We wish to charge the cost of each face $(a, b, c) \in F$ to one of its adjoining edges. An edge (a, b) can be assigned a charge of $\log |a - b|$. We define the *strongest edges* of a face (a, b, c) to be the two edges with the greatest difference between their vertices. For example, if $a < b < c$ and $|a - b| > |b - c|$, then (a, b) and (a, c) are the strongest edges. Note that $\log |a - b| > \log |b - c|$ and $\log |a - b| \simeq \log |a - c|$, so we can charge the $O(\log |b - a| + \log |c - a| + \log |c - b|)$ cost of the face to either of its strongest edges. However, we must ensure that no edge is charged more than $O(1)$ times.

Let the *star* of a vertex $S(v_i)$ be the set of simplices (edges, faces, and tetrahedra) containing v_i. Let the *closure* of the star $C(S(v_i))$ be the set of simplices in $S(v_i)$ together with the lower-dimensional simplices contained in them. Then we define the *link* of the vertex $L(v_i) = C(S(v_i)) - S(v_i)$ to be the set of faces, edges, and vertices that share tetrahedra with v_i but do not contain v_i.

Let $V(L(v_i))$ and $E(L(v_i))$ be the vertices and edges in the link of v_i (these correspond to the set of edges and faces containing v_i in the mesh). $L(v_i)$ is a two-dimensional surface, so Euler's rule applies: we know that $|E(L(v_i))| < 3|V(L(v_i))|$.

We will now direct all of the edges in $E(L(v_i))$ in such a way as to ensure that no vertex of $L(v_i)$ has indegree greater than 5. This can be done iteratively: At each step, find a vertex $v \in V(L(v_i))$ of degree 5 or less. (Euler's rule guarantees that this is possible.) Direct all edges containing v into v, and then delete v and all its edges from $L(v_i)$. At termination, all edges have been directed, and no vertex has received indegree greater than 5.

Now, recall that vertices in $V(L(v_i))$ correspond to edges containing v_i in the 3D mesh, and that edges in $E(L(v_i))$ correspond to faces. For each face (v_i, v_j, v_k), if the face's strongest edges are (v_i, v_j) and (v_i, v_k), then examine the corresponding edge $(v_j, v_k) \in E(L(v_i))$. If the edge is directed towards v_j, charge the cost of the face to the edge (v_i, v_j); otherwise, charge it to the edge (v_i, v_k).

Each edge is charged at most five times at each of its endpoints, so each edge is charged by $O(1)$ faces. The charge in each case is $\log |a-b|$ for an edge (a, b). Thus the total space used is $O(\sum_{(a,b)\in E}(\log |a-b|))$ bits.

As in the 2D case, if the 1-skeleton of the mesh has a k-compact labeling, then the representation of the mesh will use $O(|V|)$ bits.

7.4 Representation Based On Vertices

In this section we discuss our 2D mesh representation based on vertices. Our representation is based on storing the cycle of neighbors, in order, around each vertex in the mesh. (The cycle of neighbors of a vertex is also known as its *link*.) This is similar to the half-edge structure of Weiler [133]. We note, however, that all references are to vertex labels instead of pointers to other higher-dimensional simplex structures, allowing us to compress based on vertex labels. We begin with an uncompressed representation, and then describe how to compress it.

For each vertex in a 2D mesh M our representation stores the cycle of neighboring vertices. The cycle is ordered radially around the vertex in the orientation of the complex, *e.g.*, clockwise.

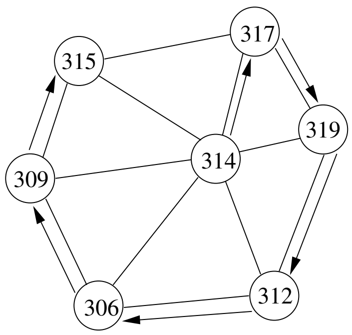
If there are holes in the mesh, then the cycle for vertex a may be split into multiple "paths" of connected vertices. Each path is stored separately.

The entry for each vertex a begins with a gamma code for $|a|$, the degree of that vertex.

Our structure supports the operations `search`, `insert`, and `delete`, as follows:

`search`(a, b)**:** finds all vertices c such that (a, b, c) form a face in M. This requires searching the cycle of neighbors of a for the predecessor and successor of b.

`insert`(a, b, c)**:** adds the face (a, b, c) to M. This requires updating the entry for each vertex. For example, the entry for vertex a is searched for the path P_b ending with b and the path P_c beginning with c. If no P_b or P_c are found, then new paths of length 1 are created to contain the missing vertices. The paths P_b and P_c are concatenated. The same process is applied to the cycle of neighbors for vertices b and c.



6	3	5	−2	−8	−5	1

Figure 7.1: The neighborhood and corresponding difference code data for vertex 314. The first entry, 6, is the degree of the vertex. Other entries are the offsets of the neighbors.

`delete`(a, b, c)**:** deletes the face (a, b, c) from M. This requires updating the entry for each vertex. For example, the entry for vertex a is searched for the path containing b and c. The path is split in two between b and c. If either of the resulting paths has length 1, it is deleted. The same process is applied to the cycle of neighbors for vertices b and c.

This interface supports traversing the mesh by repeated invocations of `search`. The time required for an operation on a is $O(|a|)$; in bounded-degree meshes this is $O(1)$.

We compress this data structure using difference coding: rather than store $b, c, d, e, \ldots$ in the entry for a, we store $b - a, c - a, d - a, e - a, \ldots$, as shown in Figure 7.1. The differences are gamma coded (using a sign bit) and concatenated. If the entry for a vertex contains multiple paths, the paths are concatenated in the entry; each gamma code is followed by a flag to indicate when one path ends and the next begins. The entry for each vertex is stored in a variable-bit-length array structure, as described in Section 3.2.

To analyze the space usage of this representation, observe that each edge (a, b) is stored twice: once in the entry for a and once in the entry for b. In each case the cost of the gamma code is $O(\log |a - b|)$. The cost of storing a gamma code for the degree of each vertex is less than two bits per edge. Thus the total space used is $O(\sum_{(a,b)\in E} \log |a - b|)$ bits.

As in Section 7.3, if the vertices of the mesh are given a k-compact labeling (as described in Section 5.2), then the representation of the mesh will use $O(n)$ bits.

Generalization to 3D. In 2D our representation mapped each vertex to the cycle of neighbors around that vertex. In 3D our representation maps each edge to the cycle of neighbors around that edge. In other words, for each edge (a, b) in the mesh, the representation stores the set of vertices that share a face with both a and b. (This is known as the *link* of the edge.) The cycle is ordered radially around the edge in the orientation of the complex, *e.g.*, clockwise. This is similar to the Dobkin and Laszlo [48] mesh structure.

To save space, edges are kept in a consistent direction: if the dictionary stores (a, b), then it does not also store (b, a). The proper direction for an edge is determined by some simple test (for example, all edges are stored as (a, b) where $a < b$).

The operations `search`(a, b, c), `insert`(a, b, c, d), and `delete`(a, b, c, d) are implemented as in the 2D case, except that searches and updates are now performed on edges rather than vertices. For example, when inserting (a, b, c, d) into the mesh, the entries for the edges (a, b), (a, c), (a, d), (b, c), (b, d), and (c, d) need to be updated. The updates themselves are still just a matter of joining paths (for inserts) or splitting paths (for deletes).

Note that there is considerable redundancy in the storage we have described. To resolve a search for (a, b, c), our structure could query the edge for (a, b), (b, c), or (c, a), and get the same result in any case. We can decrease the space requirement of our structure with a simple optimization: our structure stores only a *representative subset* of the edges in the mesh. Specifically, it stores only those edges (a, b) for which the labels a and b are either both odd, or both even. (These edges are called "representative edges".) This still permits the structure to resolve queries since any triangle (a, b, c) must contain vertices with either two even or two odd labels.

As in the 2D case, we compress the structure using difference coding. For each representative edge (a, b) with associated cycle of k vertices $c, d, e, f, \ldots$, we store the table entry $((a, b-a), (k, c-a, d-a, e-a, f-a, \ldots))$ in a variable-bit-length dictionary structure (as described in Section 3.3). The a is stored using a $\log |V|$-bit representation; the other values are gamma coded and concatenated. The dictionary absorbs the cost of storing a; it remains to account for the other differences.

Every face (a, b, c) in M contributes at most three gamma coded values to the representation: an entry for c in the cycle of the edge (a, b), an entry for b in the cycle of (a, c), and an entry for a in the cycle of (b, c). Each of those gamma coded values has size $O(\log |a-b| + \log |b-c| + \log |a-c|)$. The space used for the cycles of vertex labels is thus $O(\sum_{(a,b,c)\in F} \log |a-b| + \log |a-c| + \log |b-c|)$ bits, which by Lemma 7.3.1 is $O(\sum_{(a,b)\in E} \log |a-b|)$ bits. The table entry for each edge (a, b) stores an additional $b-a$ term which requires another $\log |a-b|$ bits, which is within our space bound. Storing a gamma code for k, the number of vertices in the cycle, requires $O(1)$ bits per edge. Thus the total space usage is $O(\sum_{(a,b)\in E} \log |a-b|)$ bits. Again, using a k-compact vertex labeling this reduces to $O(n)$ bits.

Data can be added to the mesh by including it in the dictionary entries. For example, if data is associated with a tetrahedron (a, b, c, d), the data is stored between the vertices c and d in the table entry for the edge (a, b). Each tetrahedron will have multiple representative edges with which to associate data; the data needs to be stored only with one of the representative edges (chosen in a fixed manner to make lookup easy). We make use of this in the compressed data structure. Specifically, if data is to be stored on (a, b, c, d) where $a < b < c < d$, then at least one of the edges (a, b), (a, c), or (b, c) must be a representative edge; our structure stores the data on one of those edges, in that order of preference.

7.5 Implementation

To decide which of our structures to implement we examined the space usage of each structure. We assume that the average cost of a difference code is the same in either structure, and count the number of difference codes used by each structure.

In 2D our representation based on edges uses a table entry $((a, b-a), (c-a, d-a))$ per edge. Encoding a is free (because of quotienting), but there are three codes stored per edge. The 3D generalization of that structure uses a table entry $((a, b-a, c-a), (d-a, e-a))$ per face. Again the a is free, and the cost is four codes stored per face.

In 2D our representation based on vertices uses two codes for each edge (a, b): it stores b in the entry for a and a in the entry for b. The 3D generalization potentially stores each face (a, b, c) three times (once each in the entries for (a, b), (b, c), and (c, a)), but each edge has a 50% chance of not being stored, so the expected space usage is 1.5 codes stored per face. (There is some small overhead per edge stored, but this is negligible compared to the expense per face.)

Our representation based on edges has stronger time bounds (it runs in $O(1)$ time rather than $O(|a|)$ time where $|a|$ is the degree of the vertex being examined) but uses more space, particularly for 3D. Accordingly we chose to implement the representation based on vertices.

Generating Labels. Our space bounds for our compressed structure are $O(\sum_{(a,b)\in E} \log |a-b|)$ bits. Achieving good compression with this structure relies on having a k-compact ordering, as described in Section 5.2. To achieve this, our algorithm relabels the vertices using a technique based on x-y cuts. Given a set of points, the technique first finds which of the x and y axes has the greatest diameter. It finds the approximate median in that coordinate and partitions the points on either side of that median. The points on one side are labeled first, then the points on the other side. This is done recursively to produce a labeling in which points that are near each other have similar labels. This is similar to the separator-tree based labeling scheme from Section 5.2 except that it is based on the coordinates of the vertices rather than on the edges.

If not all vertices are known before the algorithm begins, our algorithm can assign a sparse labeling to the initial vertices. When a new vertex is added, it is assigned a label that is close to the labels of its neighbors.

2D Triangulation. Our 2D compressed data structure is implemented as follows.

For difference encoding our structure uses the *nibble code* to store values, as described in Section 2.3.

It is sometimes necessary to store an extra bit b with a value v. This is accomplished with a shift operation: $v' \leftarrow 2v + b$. In particular, if any value might be negative, our difference coder stores its absolute value plus a sign bit: $v' \leftarrow 2|v| + \mathtt{sign}(v)$.

A vertex is represented with a nibble code for the degree of the vertex, followed by nibble codes for the differences to each of the vertex's neighbors. Our implementation stores two additional "special-case" bits with each neighbor to provide information about the triangle that precedes it in the link. One bit is set to indicate a gap in the link; it indicates that there is no triangle preceding that neighbor in the mesh. The other bit is set when data is associated with the triangle preceding that neighbor. In this case, the code for that neighbor is followed with a nibble code representation of the data.

As an optimization, note that for many vertices none of the special-case bits will be set. Our implementation stores a bit with the degree of each vertex to indicate if none of its special-case bits are set; if this is so, those bits are omitted in the encoding of that vertex.

Block Size	Blocks Needed	Total Space
5	745,151	10,086,381
6	475,263	9,998,531
7	283,559	9,920,446
8	164,660	10,101,104
9	94,105	10,537,195
10	53,399	11,179,987
11	30,496	11,974,072

Figure 7.2: The number of extra blocks needed for 2^{20} vertices on a uniform distribution in 2D, and the total space required if we allocate 30% more blocks than are needed.

The variable-bit-length array used in our implementation is somewhat different from that described in Section 3.2. The version of Section 3.2 was developed for extremely short bitstrings, and it is able to pack multiple bitstrings into one array slot to avoid wasted space due to underfull buckets. For our application, each bitstring represents a vertex, and underfull blocks are not a problem. Instead, the concern is efficiently allocating additional storage for overfull buckets.

Our array implementation stores the nibble codes for each vertex in an array containing one seven-byte block per vertex. If a block overflows (that is, if the storage needed is greater than seven bytes), additional space is allocated from a separate pool of seven-byte blocks. The last byte of the block stores a pointer to the next block in the sequence. Our implementation uses a hashing technique to ensure that the pointer never needs to be larger than one byte. This requires a hash function that maps (address, i) pairs to addresses in the spare memory pool. Our implementation tests values of i in the range 0 to 127 until the result of the hash is an unused block. It then uses that value of i as the pointer to the block. Under certain assumptions about the hash function, if the memory pool is at most 75% full, the probability that this technique will fail to find an $i \leq 127$ is at most $.75^{128} \simeq 10^{-16}$.

If the vertices are labeled sparsely (so that new labels can be generated dynamically), our implementation also makes use of a hash mapping between labels and vertex data blocks. One byte of memory is allocated per label; if the label is in use, this byte contains a hash pointer to the first data block for that vertex.

One bit is stored with each block to indicate whether the current block is the last in the sequence. For the first block this bit is stored with the degree of the vertex; for subsequent blocks it is stored as the eighth bit of the one-byte pointer to that block.

There is a tradeoff in the sizes of the blocks used. Large blocks are inefficient since they contain unused space; small blocks are inefficient since they require space for pointers to other blocks. In addition, there is a cost associated with computing hash pointers by searching for unused blocks in the memory pool. Figure 7.2 shows the tradeoff between these factors for our Delaunay triangulation algorithm run on 2^{20} uniformly distributed points in the unit square. We chose a block size of 7 since it gives the most efficient use of space.

To improve the efficiency of lookups our implementation uses a caching system. When a query or update

Block Size	Blocks Allocated	Blocks Used		
		2^{10}	2^{15}	2^{20}
2	$0.55n$	59%	67%	70%
4	$1.3n$	90%	90%	88%
6	$1.55n$	90%	90%	87%
8	$1.3n$	78%	73%	75%
10	$1.8n$	30%	51%	63%

Figure 7.3: The number of blocks of each size that are allocated for an n-vertex 3D mesh, and the percentage of blocks that were used for $n = 2^{10}$, 2^{15}, and 2^{20}.

is made, the blocks associated with the appropriate vertex are decoded. The information is represented in uncompressed form as a list with one vertex in the link per element of the list. The lists are kept in a FIFO cache with a maximum capacity of 2000 nodes. Update operations may affect the lists while they are in the cache. The lists are encoded back into blocks when they are flushed from the cache.

3D Tetrahedralization. The main difference between our 3D structure and our 2D structure is the need to keep track of edges rather than vertices. For the 2D structure it sufficed to keep an array slot for each vertex; for the 3D structure we need to allocate space for each edge stored by the representation. We do not use a true hashing-based dictionary structure to keep track of the edges. Instead our 3D data structure keeps a map from each vertex a to all of its representative out-edges. This is stored as a difference coded list of the corresponding neighbors. The code for each neighbor v' is followed by a code for the number of nibbles in the encoding of the representative edge (v, v'), and a pointer to the first block containing the data for that edge. (The pointer is stored using the same hash trick as above to keep pointer sizes small.) Every representative edge has its own block allocated from the memory pool, with the capability to allocate additional blocks if needed.

When an edge is queried, our implementation loads only the list for one vertex and for the edge itself into the cache. It does not need to decompress the other edges adjoining that vertex.

Since the number of nibbles needed per representative edge is quite variable, our data structure allocates from pools of 2, 4, 6, 8, or 10-byte blocks to reduce wasted space. The number of blocks in each pool was determined experimentally and is shown in Figure 7.3. The data structure ensures that each pool always has at least 10% free space; if a block cannot be allocated from a given pool, the data structure looks for a larger one. The initial block for each vertex comes from a separate array containing blocks of size 7.

Dynamic point generation. To support dynamic point generation we use an expanded label space. If a total of n vertices are to be generated, we allow for $2n$ possible labels. Each label receives a one-byte hash pointer which, if the label is in use, points to the initial data block for the corresponding vertex. The initial vertices are spread evenly across the label space.

Incremental Delaunay Algorithm. We implemented a Delaunay triangulation algorithm in two and three dimensions using our compressed data structure. We employ the well-known Bowyer-Watson kernel [25, 131] to incrementally generate the mesh. During the course of the algorithm a Delaunay triangulation of the current pointset is maintained. An incremental step inserts a new vertex into the mesh by determining the faces (or, in 3D, tetrahedra) that violate the Delaunay condition. Those faces form the Delaunay *cavity*. The edges (or, in 3D, faces) that bound the cavity are called the *horizon*. The mesh is modified by removing the faces in the cavity and connecting the new vertex to the horizon.

The cavity is connected, so it can be found by a local search on the current mesh. When a point p is inserted, the cavity is determined by a search starting from the face that contained p. To achieve optimal runtime bounds we use the idea of Clarkson and Shor [41] and maintain an association of every point p not yet inserted into the mesh with the face t_p that contains p. The search for the cavity of p will start at t_p. Their algorithm keeps the history of the mesh and uses that history to locate the t_p for each p as it is inserted. In contrast we do not keep the mesh history but maintain the association of noninserted points p to containing faces t_p on the current mesh.

At each incremental step all points on faces that were in the cavity have to be reassociated with new faces using lineside tests (or, in 3D, planeside tests); this accounts for the dominant cost of the algorithm. We have carefully implemented the *bulldozing* idea described in [18] and extended it to three dimensions.

Our implementation does not require extra memory for the lists of points since at any time a point is either a vertex in the mesh or in one such list. The memory that will be used to store the vertex in the mesh can first be used as a list node.

The algorithm maintains a work queue of faces whose interiors contain points. When no faces contain points (*i.e.*, all have been added to the mesh), the algorithm terminates.

In this scenario all points are known at the beginning. We generate labels for the input points using cuts along coordinate directions as described earlier. The runtimes reported in the next section include this preprocessing step.

Delaunay Refinement. To test our implementation's performance for the case when new points are dynamically generated at runtime, we implemented a 2D Delaunay refinement code in the style of Ruppert [106]. We augment a Delaunay triangulation by adding circumcenters of badly shaped triangles while maintaining the Delaunay property. When the initial triangulation is built we walk through the mesh once and check the quality of each face, queuing the ones not satisfying a preset minimum angle bound. The same work queue used in the triangulation phase of the algorithm is used to store the list of triangles to be split.

Whenever a new point p is generated the algorithm assigns a new label by considering the *horizon* vertices H of the cavity created by p and calculating the value v that minimizes the sum of the $\log$ norms to H. It then finds the closest label to v that is not yet used.

In the pure triangulation code, all vertices are known at the beginning, so we can store the point coordinates and the first level vertex arrays densely. In the refinement code we can only fill these arrays up to about 85% before the open address hashing takes prohibitively long. We also require extra memory for the additional map from the label space to the vertices.

Distribution	# Pts	# Extra Blocks	Time(s)
uniform	2^{18}	$70,823$	3.16
normal	2^{18}	$72,239$	3.52
kuzmin	2^{18}	$72,917$	4.36
line	2^{18}	$66,297$	3.64
uniform	2^{20}	$288,255$	13.25
normal	2^{20}	$292,580$	14.41
kuzmin	2^{20}	$292,709$	21.34
line	2^{20}	$276,124$	15.86

Figure 7.4: The number of extra 7-byte blocks needed to store triangular Delaunay meshes for various point distributions using our structure and the runtime of our 2D implementation.

7.6 Experimentation

We report experiments on a Pentium 4, 2.4GHz system, running RedHat Linux Kernel 2.4.18, GNU C/C++ compiler version 3.0.1. For all geometric operations (lineside, planeside, incircle, and insphere tests) we use Shewchuk's adaptive precision geometric predicates [111]. We use single-precision floating-point numbers to represent the coordinates. For every problem setting and size the results of our experiments were very consistent over multiple runs. Therefore we do not report ranges of results for identical runs.

2D Delaunay. We tested our 2D implementation on data drawn from several distributions to assess its memory needs for non-uniform data sets. We ran tests on the following distributions: Uniformly random, normal, kuzmin, and a line singularity. Details on these distributions can be found in [19]. In Figure 7.4 we report the number of extra (overflow) 7-byte blocks used to store Delaunay meshes of various point distributions and the runtime of our implementation. It can be seen that the runtime varies by about 40% while the number of extra blocks varies by about 10%. Furthermore the number of extra blocks used comes to only about 28% of the number of default blocks needed, which is one per vertex. In our experiments we set the number of extra blocks available to 35% of the number of default blocks. The extra blocks therefore fill to about 80% of capacity. Given this setting, the total space we require for the mesh is 1.35×7 bytes/vertex, which is 4.725 bytes/triangle.

Next, we compare runtime and memory usage of our implementation to Shewchuk's Triangle [110] code which is the most efficient code reported by Boissonnat et. al. [21]. In Figure 7.5 we report the runtime of our (incremental) code vs. Triangle's divide-and-conquer and its incremental implementation. We report the total memory use of both codes in Figure 7.6 and break down our memory use for the simplicial mesh, point coordinates and the work queue in Figure 7.7. While using just about a third of the memory our code runs about 10% slower than Triangle's divide-and-conquer implementation and is about an order of magnitude faster than Triangle's incremental implementation. In our code 50% of the memory is used to represent the mesh, 40% to store the coordinates, and 10% for the work queue.

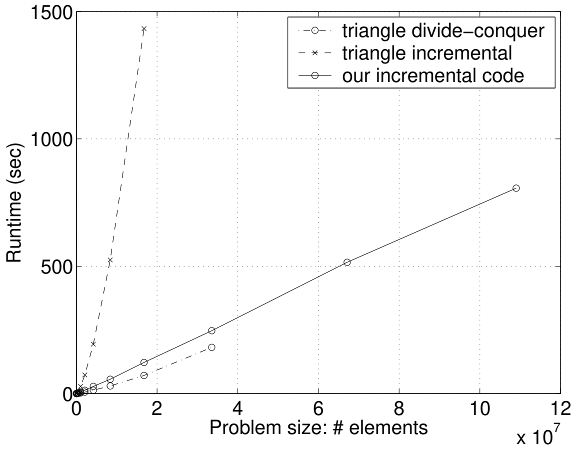


Figure 7.5: Runtime in 2D, uniformly random points

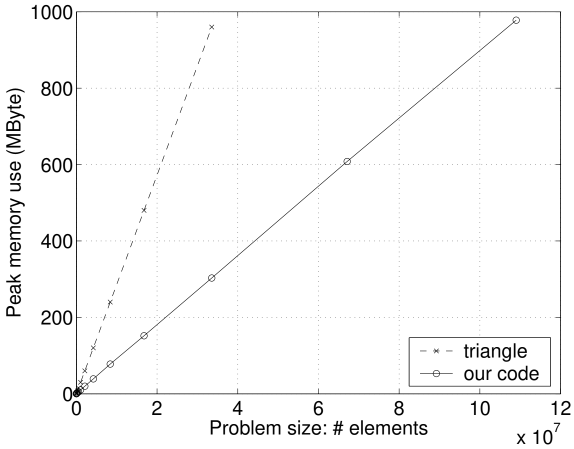


Figure 7.6: Memory use in 2D, uniformly random points

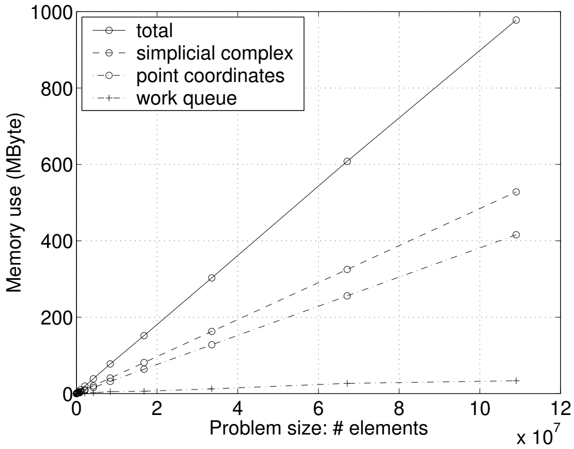


Figure 7.7: Breakdown of memory use in 2D, uniformly random points

Distribution	# Pts	# Bytes used	Time(s)
uniform	2^{16}	$2,525,309$	9.26
normal	2^{16}	$2,572,659$	9.38
kuzmin	2^{16}	$2,571,769$	11.23
line	2^{16}	$2,264,465$	8.77
uniform	2^{18}	$10,135,321$	39.59
normal	2^{18}	$10,463,761$	41.89
kuzmin	2^{18}	$10,444,195$	45.04
line	2^{18}	$9,372,669$	38.97

Figure 7.8: The number of bytes needed for occupied blocks to store tetrahedral Delaunay meshes for various point distributions and the runtime of our 3D implementation.

3D Delaunay. As in 2D we tested our 3D implementation on the same four point distributions. In our 3D structure we allocate memory blocks of different size. To compare the memory needs for various point distribution, we report the number of bytes used to store occupied blocks in Figure 7.8. As in 2D the runtimes differ, but the memory needed is nearly independent of the distribution.

For the distributions we tested we found that our meshes contained roughly 6.5 tetrahedra per vertex.

We compare our 3D implementation with uniform random data to Shewchuk's Pyramid code [109][2]. Figures 7.9 and 7.10 show the runtime and the memory usage. Figure 7.11 breaks down the memory usage of our code.

In comparison our implementation runs slightly faster and uses only about one third of the memory. In 3D the representation of the mesh uses about 75% of the total memory (about 7.5 bytes per tetrahedron, which is slightly under 50 bytes per vertex for the distribution we tested); point coordinates and work queue account for 18% and 7%, respectively.

2D Delaunay refinement. For our 2D Delaunay refinement code we compare runtime and memory use to our pure 2D Delaunay code, as shown in Figures 7.12 and 7.13. The figures show problem size in terms of the final number of faces in the mesh. In the pure Delaunay code, all n points are known initially; in the refinement code, only $n/2$ points are known initially and the other $n/2$ are generated and labeled on the fly as described in Section 7.5. We refine the mesh up to a minimum angle of $26.85°$.

The runtimes for the two versions are almost identical. We need about 30% more memory in the refinement code. Additional memory is needed for the map from labels to vertices and for slack in the point coordinate array and the first level vertex array needed for our hashing technique.

[2]We note that the version of Pyramid we are using is a Beta release.

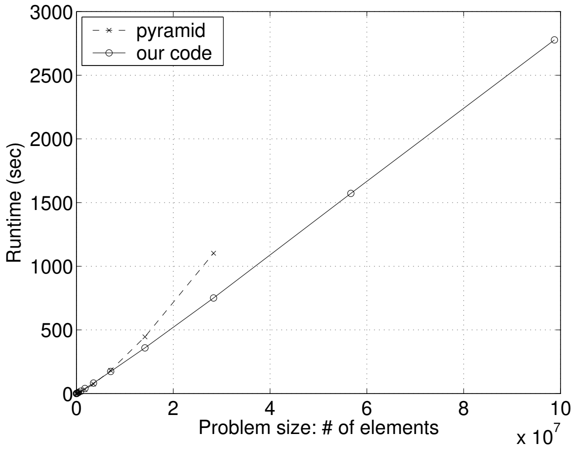


Figure 7.9: Runtime in 3D, uniformly random points

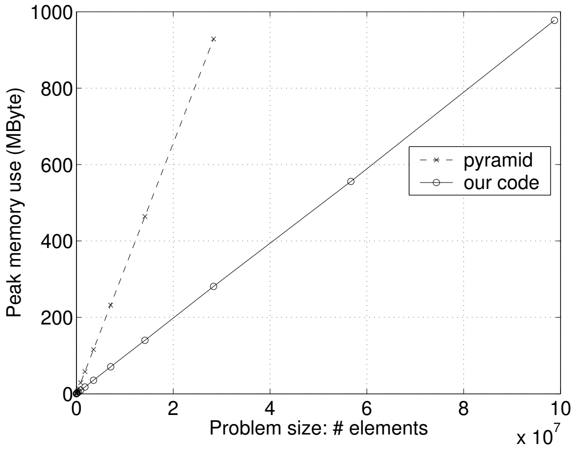


Figure 7.10: Memory use in 3D, uniformly random points

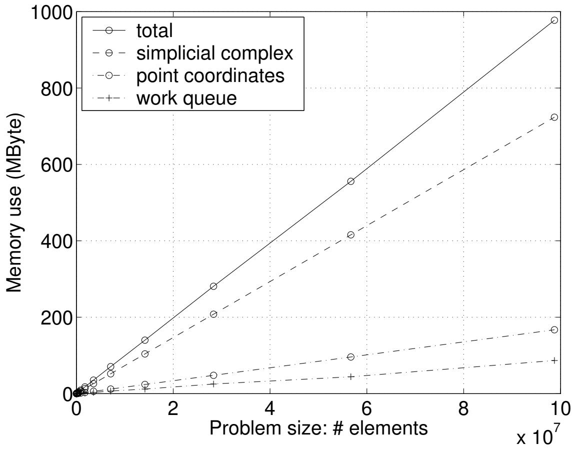


Figure 7.11: Breakdown of memory use in 3D, uniformly random points

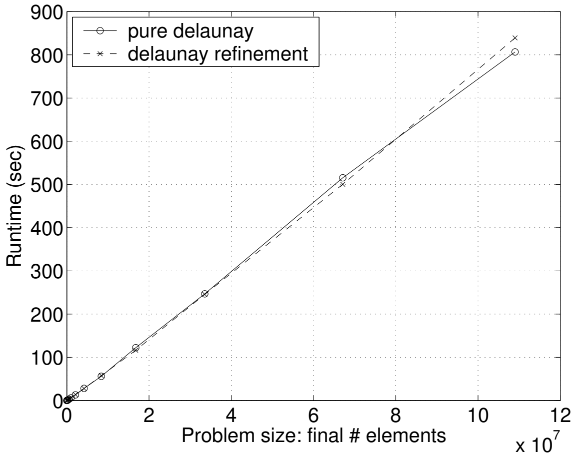


Figure 7.12: Runtime in 2D, pure Delaunay vs. Delaunay refinement

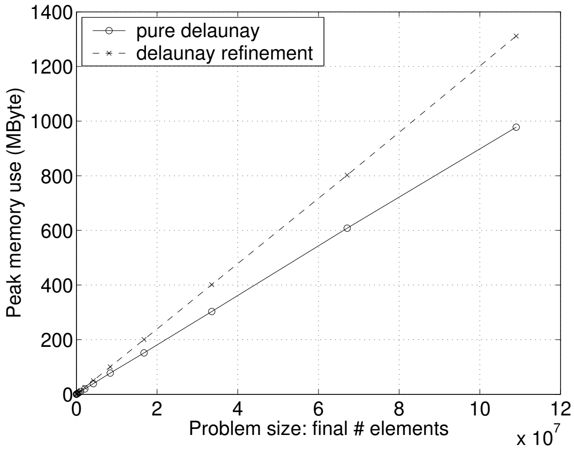


Figure 7.13: Memory use in 2D, pure Delaunay vs. Delaunay refinement

7.7 Discussion

The representation we described can be used as an alternative to external memory (out-of-core) representations, when the mesh is within a factor of five or so of fitting in memory relative to a standard representation. Our representation has the advantage that it allows random access to the mesh without significant penalty, and can therefore be used as part of standard in-memory algorithms (or even code) by just exchanging the mesh interface.

In conjunction with external-memory techniques. For very large problems our representation can be used in conjunction with external-memory techniques. Since in our representation the ordering of the vertices is designed to be local (it is based on the quad/oct tree decomposition), and the blocks of memory for vertices are laid out in this ordering, nearby vertices in the mesh will most likely appear on the same page. (One problem is that, if the data for a vertex overflows, our dictionary structure assigns a new block for the overflow data using a hash, which has no locality. In Chapter 8 we correct this by, essentially, breaking the large dictionary over $|V|$ vertices into an array of smaller ones for 16 vertices each.) Using this representation, algorithms that have a strong bias to accessing the mesh locally (e.g., see the recent work of Amenta, Choi and Rote [5]) will tend to have good spatial locality and work well with virtual memory when it does not fit into physical memory.

Generalizations to d-dimensions. The idea of storing the link of every $d-2$ dimensional simplex generalizes to arbitrary dimension. The compression technique also generalizes to arbitrary dimension, but is likely to be ineffective for large dimensions. This is because the size of the difference codes depends on the separator sizes [15], which in turn depends on the dimension. Choosing an effective way to select the representative subset of the $d-2$ dimensional simplices will depend on the dimension and would need to be considered to use our representation on dimensions greater than three. We have not done any experimentation to analyze the effectiveness of our techniques on dimensions greater than three, or to compare our representations to other representations.

Vertex Relabeling Schemes. Our system of x–y–z cuts for relabeling vertices is effective, but crude. It could be improved using a *child-flipping* optimization as discussed in Chapter 5. Essentially, when making a cut, the relabeling algorithm can use information from past cuts to decide which side of the cut should receive the higher part of the label space.

A further improvement might involve labeling the vertices using a Hilbert curve. We experimented briefly with using a Hilbert curve library to relabel the vertices, but found that the relabeling time required was too great compared to the increase in compression provided. Other authors, such as Papadomanolakis et al. [99], have been able to successfully use a Hilbert curve to relabel the vertices of a tetrahedral mesh for locality.

Point Location. In addition to the use of Hilbert curves, Papadomanolakis et al. describe a technique for rapid point location based on locality of vertex labels. To find the tetrahedron containing a vertex, they locate the nearest point in Hilbert space and walk through the mesh to the destination. Since our mesh structure has locality of vertex labels as well, it might be possible to combine this technique with our work to produce a savings in both time (in bulldozing the points to allow for our point location) and space (since the work queue could be replaced with a more compact structure).

Acknowledgements

We are grateful to Jonathan Shewchuk for commenting on the paper and letting us use a pre-release version of Pyramid. This work was done as part of the Sangria [67] project, and several project members have contributed ideas.

Chapter 8

Compact Parallel Delaunay Tetrahedralization

8.1 Introduction

In Chapter 7 we presented a compact data structure for representing 2D and 3D meshes, accompanied by a sequential algorithm using the structure for Delaunay triangulation. In this chapter we show how to parallelize the algorithm. We describe changes that were made to the algorithm and data structure.

For the 3D case we present experimental results. When we increase the problem size and number of processors by a factor of 64, the vertex insertion rate increases by a factor of 37.17. However, we cannot call this a "speedup" measurement since the runs are on different job sizes and the algorithm performs $O(n \log n)$ work for the distributions we test. The amount of work performed by the 64-processor run is more than 64 times greater than the amount performed by the one-processor run, so the ratio 37.17 underestimates somewhat the actual speedup of the algorithm.

These results could be useful for many applications dealing with large 3D meshes. As an example, the Quake project [123] makes use of hexahedral meshes of size up to 1.37 billion grid points. That application uses an out-of-core algorithm to generate the mesh, and uses hexahedra to decrease the number of elements in the mesh. Our structure can manipulate tetrahedral meshes of that size in main memory: with 64 processors it generated a mesh containing 1.51 billion vertices and 10 billion tetrahedra using 5512 seconds and 197GB of RAM. The vertices were chosen uniformly at random from the unit cube.

The compactness of our data structure is preserved: where the structure of Chapter 7 used 50 bytes per vertex for $3D$ mesh data, our structure here uses 77.5 bytes per vertex counting overhead for faster decoding and for synchronization. With the addition of space for vertex coordinates (24 bytes per vertex) and temporary storage for the meshing algorithm (28 bytes per vertex), the total memory footprint of our algorithm is 130 bytes per vertex. This is a significant improvement over standard representations, which can require 300 to 500 bytes per vertex just for meshing data.

The algorithm as presented is only used to construct a Delaunay mesh over a given set of points; however, the generalization to Delaunay refinement described for the sequential version would also apply in the parallel case.

We have tested our algorithm on the uniform, gaussian, kuzmin, and line singularity distributions (see [19] for details). For those distributions its time and space usage are nearly independent of the distribution used. We also tested our algorithm on real-world data: a set of 133 million grid points from the Quake project [123]. The fact that the vertices were at grid points posed some challenges to our application, but we were able to overcome this using small random perturbations.

Our algorithm is a shared-memory parallel version of the incremental insertion algorithm from Chapter 7. To review, during the course of the algorithm a Delaunay triangulation of the current pointset is maintained. An incremental step inserts a new vertex into the mesh by determining the elements that violate the Delaunay condition. We use the idea of Clarkson and Shor [41] and maintain an association between uninserted vertices and the tetrahedra containing those vertices. We keep a work queue of tetrahedra whose interiors contain points; threads draw tetrahedra at random from the queue for processing. Threads lock pieces of the mesh as they prepare to insert a vertex; if a thread is unable to obtain a lock, it aborts the insertion and draws a different job from the queue instead.

We have implemented several optimizations to improve the parallel performance of the algorithm. In particular, we bootstrap the algorithm by growing the mesh sequentially (using the Pyramid algorithm of Shewchuk [109]) until it is sufficiently large to avoid excessive contention between threads. We then run parallel point-location to associate all uninserted vertices with simplices in the mesh. The main incremental algorithm begins once this point-location step is complete.

Information in our data structure is stored by vertex label, using difference coding for compression. To improve the quality of difference coding, we preprocess the input vertices, relabeling them using x–y–z cuts so that vertices that are close spatially have similar labels.

Our basic data structure is a variant on the 3D representation from Section 7.4. The structure is modified to improve data locality: it is divided into groups of G vertices, and hashing is only performed within each group. (That is, an edge (v, v') is stored in the hashtable corresponding to the group containing v.) Groups can be locked individually to prevent concurrent access by multiple threads.

In this chapter we describe the changes made to parallelize our algorithm. We discuss our locking mechanism and several means of decreasing contention between threads for locks. In our experimental results section we present results from running the algorithm with varying queueing disciplines and varying point distributions. We analyze the speedup and the time and space efficiency of the algorithm for up to 64 processors.

Related Work. There has been significant previous work on parallel Delaunay algorithms, using three main approaches to avoid conflicts between threads.

The first approach is that of divide-and-conquer: The mesh is (recursively) partitioned in two regions, with each partition built by a separate processor. The border between the regions must be constructed separately. Aggarwal [2] described an algorithm which constructed the border by joining the regions after they were built. Chen et al. [35] described an algorithm which assigned certain points to both regions; this resulted in some duplicate work but meant that joining the regions involved only discarding duplicate triangles. Hardwick [63], Blelloch et al. [19], and Lee et al. [77] describe an algorithm that projects the 2D points to a paraboloid in 3D, compute the lower convex hull, and use that to derive a border before building the regions. These algorithms only work in two dimensions.

The second technique for parallel Delaunay meshes involves incremental insertion (using the Bowyer-Watson kernel [25, 131]). Most algorithms for incremental insertion avoid collisions by assigning a region of the mesh to each processor. Operations involving multiple regions of the mesh are handled by message-passing between processors. For this technique it is necessary to perform load-balancing between regions while still ensuring that each region's border is small. In 2D this was done by Okusanya and Peraire [93] and Chrisochoides and Sukup [39].

The region-per-processor technique was also used by Chrisochoides and Nave [37, 38] to produce a 3D parallel algorithm for a message-passing architecture. Much of the detail in that work involved minimizing the latency from interprocessor communication, a problem which we can avoid since our concern is with a shared-memory machine. Also, our work is on a greater scale: our largest mesh is 5000 times larger than theirs.

Kohout et al. [76, 75] describe a 2D incremental insertion algorithm which does not assign a region to each processor; instead, all processors draw from a global queue, similar to our own work. They report a speedup of up to 5.84 on eight processors. Their algorithm uses a DAG data structure for point location (whereas our algorithm associates points with tetrahedra to save memory). Kohout et al. also give a good survey of related work.

None of these consider space-efficiency of their representations. The only compact dynamic mesh representation we know of is our own, described in Chapter 7.

Compressed Meshes. There has been considerable work involving compressed meshes [44, 61, 121, 98, 105, 120, 70, 66, 53]. In three dimensions these methods can compress a tetrahedral mesh to less than a byte per tetrahedron [120]—about 6 bytes/vertex (not including vertex coordinates). 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.

Another option for handling larger meshes is to maintain the mesh in external memory. To avoid thrashing, this requires designing algorithms for which the access to the mesh is carefully orchestrated. Several such external memory algorithms have been designed [55, 45, 43, 83, 128, 7, 124]. Of particular note is the bucketed randomized insertion order scheme of Amenta et al. [5], which improves the memory locality of an out-of-core tetrahedralization algorithm by altering the insertion order of the vertices. This insertion order might combine with our own work to form an improved out-of-core algorithm using compressed data structures with very strong memory locality. We discuss this further in Section 8.5.

8.2 The Algorithm

The sequential version of our algorithm is described in detail in Chapter 7; we will summarize it here.

Locality. For several purposes, involving both compression quality and locality of memory access, we found it important to ensure that vertices that were close spatially (eg, those likely to share edges in the mesh) had similar labels. To ensure this, as a preprocessing step we relabeled the vertices using x–y–z cuts.

Given a set of points, our algorithm first finds which of the x, y, and z axes has the greatest diameter. It finds the approximate median of that diameter and partitions the points using that median. The points on one side are labeled first, then the points on the other side. This is done recursively (and in parallel) to produce a labeling in which points that are near each other have similar labels. This is similar to the separator-based technique for graph relabeling from Section 5.2 except that it occurs before any edges have been added to the graph.

If not all vertices are known before the algorithm begins, our algorithm can assign a sparse labeling to the initial vertices. When a new vertex is added, it is assigned a label that is close to the labels of its neighbors. In previous work [14] we presented results for a Delaunay refinement algorithm that made use of this technique; this algorithm could be made parallel in a straightforward fashion.

Sequential Insertion. We employ the well-known Bowyer-Watson kernel [25, 131] to incrementally generate the mesh. The algorithm maintains a Delaunay triangulation of the current pointset at all times. An incremental step inserts a new vertex into the mesh by determining the elements that violate the Delaunay condition. Those elements form the Delaunay *cavity*. The faces that bound the cavity are called the *horizon*. The mesh is modified by removing the elements in the cavity and connecting the new vertex to the horizon.

To achieve optimal runtime bounds we use the idea of Clarkson and Shor [41] and maintain an association of every point p not yet inserted into the mesh with the tetrahedron t_p that contains p. The search for the cavity of p starts at t_p. With each tetrahedron we keep data indicating which uninserted points are contained in it. We maintain a work queue of tetrahedra which contain points.

At each step, the algorithm draws a tetrahedron from the front of the queue. The algorithm checks that the tetrahedron is still in the mesh (that is, that an update has not deleted that tetrahedron since it was added to the queue). If so, the algorithm extracts a point p from the tetrahedron and performs the insertion. It uses the *bulldozing* idea described in [18] to reassociate points from the cavity with new tetrahedra. Any new tetrahedra that contain points are added to the back of the work queue.

This algorithm has an expected $O(n \log n)$ runtime if the elements for insertion are picked at random.

8.2.1 Parallel version.

The parallel version of the algorithm is the same as the sequential version except that every thread draws work from the queue. To avoid overlapping reads and writes between threads we use data locks in two ways: on the mesh and on the work queue. All data locks are "test-locks" rather than "wait-locks": if a thread fails to acquire a lock, it aborts the operation rather than waiting for the lock to become free.

The meshing data structure stores edges in a variable-bit-length dictionary structure, as described in Section 7.4. To improve memory locality, vertices are divided into groups of size G, and each vertex group has its own hashtable in which to store edge data. (In our experiments we use $G = 16$.) With each vertex group we store a lock, so that only one thread may access that group at a time. The space cost of the lock is amortized over the G vertices in the group.

As a thread explores the cavity for a point p, it secures the lock on each vertex it encounters. (Recall that the vertices have been relabeled so that vertices with similar labels are close together; it is likely that many of the vertices for a cavity may share the same few locks.) If a thread encounters a vertex that is locked by another thread, it aborts the insertion: it releases all of its locks and returns the tetrahedron to the work queue. Otherwise, once the thread has secured the locks on all of the cavity, it performs the insertion as normal and releases the locks when finished.

The work queue is also secured by locks to prevent concurrent access. In the parallel version for p processors the work queue contains $10p$ subqueues. (We experimented with several queue configurations—see Section 8.4 for details.) Each subqueue has its own separate lock; when a thread accesses the work queue, it probes the subqueues at random until it acquires the lock on one. The thread operates on the queue (adding a number of tetrahedra to be processed, or randomly extracting a tetrahedron for processing) and then releases the lock.

In rare cases it may be necessary for a thread to allocate more memory using calls to `malloc`. (For example, this is needed if a hashtable overflows.) To do this a thread must wait until it acquires a global lock. This is the only time in our algorithm when a thread waits to acquire a lock.

Contention. When the mesh is very small compared to the number of threads operating on it, there is danger of contention: multiple threads may all compete for the same few vertices, such that for a long time, no thread is able to acquire enough vertex locks to perform an insertion in a certain area of the mesh. This may result in a few very large tetrahedra remaining untouched, with many uninserted vertices on them, while other areas of the mesh are tetrahedralized to a fine resolution. When a thread finally acquires enough locks to handle the tetrahedron, the associated cavity is very large. In addition to the obvious inefficiencies, the space required to hold the full cavity in the cache is considerable; this places strain on the caching and memory allocation structures, which is undesirable.

An easy solution to the contention problem is to hold some threads back at the start of the algorithm. Experimentally we find that restricting the density of threads to one per 2^{14} vertices in the mesh is sufficient to eliminate contention almost entirely. Unfortunately, this causes other types of slowdown: for the initial 2^{14} vertex insertions, only one thread is active in the mesh.

To see why this is a problem, consider the time complexity of a vertex insertion. We assume that finding the cavity for a vertex requires constant time per insertion. (This is the case for bounded-degree meshes in the absence of contention. With random data, for example, each cavity contains an average of 26 tetrahedra). However, our algorithm must also perform planeside tests for the uninserted vertices that lay in the deleted tetrahedra. If k of the n vertices have been inserted, then there are an expected $\Omega(n/k)$ vertices per insertion that require planeside tests. (In particular, the first insertion performed requires $\Omega(n)$ planeside tests for the uninserted vertices.) Performing all of these tests with one thread is inefficient.

Bootstrapping via Pyramid. To run our algorithm in parallel, we need to build the mesh sufficiently large that all threads can use it at once. To do this we make use of a separate tetrahedralization algorithm—the serial Pyramid algorithm of Shewchuk. That algorithm is different from ours in that it does not associate uninserted vertices with tetrahedra; instead, to insert a vertex v, it walks through the mesh using plane-side tests to locate the tetrahedron that should contain v.

Our bootstrapping algorithm works as follows. Given n vertices and p processors, we first relabel the vertices using x–y–z cuts, as in the standard algorithm. We then sample k vertices for insertion via Pyramid. (We could perform the sampling at random; however, since the labels are assigned using x–y–z cuts, we instead sample at evenly spaced intervals. This produces a more evenly spaced distribution.) Once the Pyramid mesh data structure is built, we perform point location on the remaining vertices to associate them with tetrahedra in the mesh.

Each processor performs point location on a contiguous block of vertices. Since this does not involve modifying the mesh it produces no conflicts between threads. Shewchuk's point location routine allows us to begin the walk from any tetrahedron in the mesh. Since the vertices have high spatial locality (due to our reordering via x–y–z cuts), we begin the walk for each vertex v from the tetrahedron that contained vertex $v-1$. The cost for this point location is thus quite low.

When all the vertices have been mapped to tetrahedra, the Pyramid mesh structure is deallocated. The work queue is allocated, and the tetrahedra are inserted into it. (The space used for Pyramid is reused for the work queue, so that it does not add to the total space cost of the algorithm.) Our parallel insertion algorithm then begins as normal.

There is a tradeoff between insertion of vertices using Shewchuk's mesh-walking code and our bulldozing code. If there are n total points, and k points have been inserted into the mesh, then inserting a vertex using our code requires $\Theta(n/k)$ work (spent using planeside tests to reassociate the points in the cavity with new tetrahedra). The cost of the same insertion using Pyramid is $\Theta(k^{\frac{1}{4}})$ serial time, which is equivalent to $\Theta(pk^{\frac{1}{4}})$ work.

To optimize performance we must select a k such that these costs are balanced. Solving the expression $\frac{n}{k} = pk^{\frac{1}{4}}$ yields $k = (\frac{n}{p})^{\frac{4}{5}}$. For our experimental setup, however, we always use $n = 2^{24.5}p$: the same k should be valid throughout. Experimentally we find that k should be between 2^{19} and 2^{20} for best performance. With 64 processors, our initial mesh needs roughly 2^{20} vertices to avoid contention. Accordingly we use bootstrapping of 2^{20} vertices for all of our tests.

Cleanup. As the algorithm nears termination, it may occur that only one region of the mesh still contains uninserted vertices. In this case, the algorithm may encounter contention. To prevent this, threads leave the mesh as the number of remaining uninserted vertices decreases: thread k leaves the mesh when fewer than $2048k$ uninserted vertices remain. Since the last insertions are quite rapid (as they involve almost no planeside tests), this does not cause significant slowdown.

8.3 Data Structure

Here we summarize the data structure we use to represent our 3D meshes. The structure is adapted from that of Chapter 7. We have made several modifications. First, we hash the edges of our structure explicitly. The previous implementation stored a pointer to each edge from the bucket corresponding to its first vertex.

Our new data structure hashes on the full edge as described in Section 7.4. Second, we divide the hashtable into *groups* containing data for $G = 16$ vertices each, and assign a data lock to each group. Third, we use a simplified memory-allocation system. Our representation from Chapter 7 used a system that allocated memory blocks of size 2, 4, 6, 8, or 10 bytes, depending on the space required. Our new representation allocates only fixed-length blocks of size six bytes. Fourth, we take special care to avoid thread contention for memory pages by allocating all data for a vertex group in contiguous memory.

The data structure supports the following operations:

- `add`(v_1, v_2, v_3, v_4, d) adds the (oriented) tetrahedron (v_1, v_2, v_3, v_4) to the mesh, with associated data d. For our application d is the label of an uninserted point contained within the tetrahedron.
- `find`(v_1, v_2, v_3) searches the mesh for the tetrahedron containing the (oriented) triangle (v_1, v_2, v_3). It returns v_4 and the associated d.
- `delete`(v_1, v_2, v_3, v_4) deletes the tetrahedron from the mesh.
- `lock`(v) attempts to lock the vertex v and returns a boolean indicating success or failure.
- `unlock`() releases all locks owned by the calling thread.

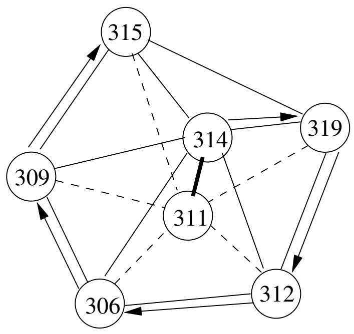
As in the sequential version of our algorithm, our structure stores the *link* for a set of edges (1-simplices). The link of an edge is the oriented cycle of vertices that connect to both endpoints of the edge (see Figure 8.1 for an example). Not all edges are stored: only edges between vertices having the same even-odd parity are kept in the dictionary structure. This still permits us to resolve `find` queries since any triangle must contain at least one such edge. Also, edges are only stored in one direction, determined according to a hash function.

All vertices in the link for an edge (v, v') are stored by vertex label, and compressed via *difference coding* [136] with respect to v. The difference code we use is the *byte-aligned code* from Section 2.3. We chose this code because it is rapid to encode and decode (as described in Section 5.7). The data d for a tetrahedron is stored as described in Section 7.5. For our application, this data is the label of an uninserted point that is contained in the tetrahedron. This data is also difference encoded with respect to v.

The codes for all of the data and vertices in an edge's link are concatenated. The resulting bit-string is stored in a variable-bit dictionary structure as described in Section 7.4.

Uninserted Points. It may occur that a tetrahedron contains more than one uninserted point. We represent these points using a linked list. We keep an array `next`$[0..N-1]$ such that, if point p is contained within a tetrahedron, then `next`$[p]$ is the index of another point within the same tetrahedron (or -1 if there is no such point). The first point in the list is stored with the tetrahedron in the mesh data structure.

Memory Locality. In an environment in which multiple threads are accessing a data structure, it is important to ensure that memory accesses involved in a query go to a small set of cache lines. Hashtables have notoriously poor memory locality; to address this, we divide the vertices into vertex groups of size G. (We used $G = 16$ in our experiments.) Each vertex group is allocated with its own hashtable (*i.e.*, its



5	319	312	306	309	315

5	5	−2	−8	−5	1

Figure 8.1: The neighborhood and corresponding difference code data for the edge $314 \rightarrow 311$. The first entry, 5, is the degree of the vertex. Other entries are the offsets of the neighbors from 314.

own variable-bit-length dictionary); all data associated with the hashtable is kept in the same contiguous block. (In rare cases the hashtable may require resizing, in which case the additional memory must be allocated elsewhere. For the settings we chose this happens roughly 15% of the time.) Edges are stored in the hashtable corresponding to their first vertex. Along with the hashtable data we keep a data lock, shared by the G vertices of the vertex group; a thread must acquire the lock in order to read from or write to the hashtable.

Caching. To improve the efficiency of lookups our implementation uses a caching system. When a query or update is made, the codes associated with the appropriate edge are decoded. The information is represented in uncompressed form as a linked list with one listnode per vertex in the link of the edge. The lists are kept in a cache which is specific to the thread performing the query or update. Update operations may affect the lists while they are in the cache. As part of an update, the application may delete simplices, producing holes in the mesh; however, we maintain the invariant that edge links that are written out of the cache must be full cycles. Thus the cache is only flushed after a new vertex insertion is complete.

8.4 Experimentation

Experimental Setup. We ran our experiments on `rachel.psc.edu` [101], a pair of HP GS 1280 SMP machines with 64 1.15-Ghz EV67 processors each. The operating system was Tru64 Unix. We used the OpenMP [94] library to provide parallel functionality. Our code was written in C and C++; we compiled using the command `cxx -O -fast -arch ev7 -tune ev7 -omp`.

There were 4 Gbytes of RAM available per processor. Given our space usage (discussed below) this was sufficient to build a mesh of about $2^{24.5}$ (about 23 million) vertices per processor.

processors:	1	2	4	8	16	32	64
Total runtime	3202s	3769s	4352s	4435s	4686s	5090s	5512s
Parallel loop	2995s	3553s	4063s	4159s	4416s	4725s	5064s
s_0, vtxs/p/sec	7130	5388	4221	3990	3454	3170	2853
s_{10}, vtxs/p/sec	7768	6809	6183	6043	5779	5439	5101
s_{15}, vtxs/p/sec	7678	7152	6596	6510	6263	5935	5712
Init Fails	0	3.8M	5.4M	6.2M	6.5M	6.7M	6.7M
Dig Fails	0	40K	65K	80K	90K	99K	106K
Rep Fails	0	1.1M	2.2M	2.9M	3.1M	3.3M	3.2M

Table 8.1: Performance measurements per processor for our algorithm. We inserted $2^{24.5}$ (about 23 million) vertices per processor.

We used the exact arithmetic predicates of Shewchuk [111] for all geometric tests. Additionally we used the beta version of Shewchuk's Pyramid code [109] to bootstrap our main parallel algorithm.

Main Results. We ran our algorithm on points with the uniform distribution using between 1 and 64 processors. In all cases we used $2^{24.5}$ (about 23 million) points per processor. We used a fixed amount of bootstrapping (2^{20} vertices) for each run. In the one-processor case our algorithm took 3202 seconds, for an average of 7410 vertices/second. In the 64-processor case our algorithm averaged 4305 vertices/second per processor. The vertex insertion rate increased by a factor of 37.17. This ratio actually underestimates the speedup of our algorithm since the amount of work per vertex inserted is $\Theta(\log n)$ (for the distributions we test). After accounting for this discrepancy we get a speedup of 46.27.

We decompose the runtime of our algorithm into several factors (see Table 8.1). The total runtime listed includes all steps of the algorithm from x–y–z reordering to termination. The next time measurement given includes only the parallel loop (that is, without the bootstrapping, reordering, or initialization phases). For convenience of analysis we divide the parallel loop into *stages* $s_0 \dots s_{15}$, each of which involves inserting $1/16$ of the total pointset. (Note that s_0 involves slightly fewer insertions than the others since it does not include the 2^{20} vertices used in bootstrapping.) For each of s_0, s_{10}, and s_{15} we give the number of vertices inserted per processor per second. The higher cost of earlier steps is due to the large amount of point-location work performed during these steps.

Finally, we give three measures of contention. A lock failure is classed as an *initialization failure* if the thread fails to obtain the lock on one of the vertices in the initial tetrahedron, or a *dig failure* if the thread fails to obtain the lock on a subsequent vertex while performing the insertion. If the failure occurs immediately after a previous failure, it is instead classed as a *repeat failure*. We give the average number of each type of failure per processor.

The 64-processor run inserted $1,518,041,200$ points, producing $10,274,246,916$ tetrahedra. As far as we know this is the largest tetrahedral Delaunay mesh that has been generated.

Discipline	Init Fails	Rep Fails	Maximum Queue Size	Runtime (main loop)
FIFO (2p)	113M	234M	181M	4620s
FIFO (10p)	51M	23M	160M	4321s
QR (2p)	68M	19K	257M	4165s
QR (10p)	37M	11K	200M	4234s
RAND (2p)	32K	107	595M	4249s
RAND (10p)	32K	47	589M	4300s

Table 8.2: Impact of various queueing disciplines on our algorithm using $2^{27.5}$ (about 190M) vertices and 8 processors.

Queueing Disciplines. In our algorithm there is a central work queue from which all threads draw tetrahedra for processing. To avoid concurrency issues, the queue is divided into a number of subqueues; when a thread wishes to access the queue, it chooses randomly from the subqueues until it finds one that is not in use. Here we discuss the issues involved in design of the work queue.

We considered three possible queueing disciplines for our work queue. The first we considered was the standard FIFO queueing discipline. A concern with this algorithm is that, on completion of an insertion, our threads may add to the queue a large number of tetrahedra that all share the same vertex (the newly inserted point). If two or more threads attempt to handle the tetrahedra resulting from a single push, then most (or all) of the threads will encounter locked vertices and abandon the job.

A second discipline we considered was the random queue (RAND): tetrahedra are added to the tail of the queue but extracted at random from any point within the queue. This ensured that our threads' access patterns were random. Unfortunately we found experimentally that it led to larger queue sizes than the FIFO queue: large numbers of "garbage" tetrahedra (those that no longer existed in the mesh) collected in the queue and were not removed until near the end of the algorithm.

The third option we considered was the "queue-random" discipline (QR), a compromise between the first two disciplines. A thread would initially attempt to draw a tetrahedron from the front of the queue; if work on that tetrahedron failed due to contention, the thread would next draw from a random point within the queue.

In addition to experimenting with various queueing disciplines, we performed experiments with varying the numbers of subqueues in the work queue. The FIFO queueing discipline problem occurred when multiple threads accessed the same subqueue within a short amount of time; by increasing the number of subqueues we hoped to make this event less likely. For p processors we experimented with using $2p$ and $10p$ subqueues.

The results of our experiments are shown in Table 8.2. We classify lock failures as *initial failures* or as *repeat failures* depending on whether the thread had encountered a lock failure just prior to the current one. The increase in failures for the FIFO queueing discipline is quite dramatic, and the increase in queue size for the random disciplines equally so. However, the corresponding increase in runtime was fairly small since most of the failures occurred before significant work was performed. Thus, we chose to minimize the space

One processor, $2^{24.5}$ vertices		
Distribution	Time	Additional Bytes/Vtx
uniform	3136s	4.30
normal	3164s	4.67
kuzmin	3182s	4.56
line	3147s	2.80

Eight processors, $2^{27.5}$ vertices		
Distribution	Time	Additional Bytes/Vtx
uniform	4296s	4.69
normal	4301s	4.79
kuzmin	4478s	4.80
line	4301s	3.67

Table 8.3: Space used and time required by our algorithm for various distributions.

used by the work queue: for our experiments we use the FIFO queueing discipline with $10p$ subqueues.

Space usage. Our algorithm allocates space for several purposes. The vertex coordinates use 24 bytes per vertex (three eight-byte floating-point values). The array `next`$[p]$, used to link together vertices in the same tetrahedron, uses 4 bytes per vertex. For the work queue we allocate two entries per vertex, each containing three integers a, b, c, for a total of 24 bytes per vertex. (To find the fourth vertex of a tetrahedron, the algorithm performs a lookup on (a, b, c).)

The mesh structure divides vertices into groups of $G = 16$; for each group it allocates a structure of 1160 bytes, or 72.5 bytes per vertex. This includes blocks of memory for storing difference codes (one 7-byte block and ten 6-byte blocks per vertex, as described in Section 7.5), structures to handle allocation of the memory, and a pointer to an additional block of memory if necessary. It also includes a data lock.

When the hashtable for a group overflows, additional memory is allocated from the heap. We chose settings such that this occurs on $13.8\% - 15.4\%$ of groups in the tests for Table 8.1, with the overflow becoming more likely for larger input sizes. The cost is $4.30 - 4.77$ additional bytes per vertex.

The algorithm allocates some fixed-size structures as well (caches and pools of linked list nodes), but the memory for these is negligible. The total space cost for our algorithm, then, is less than 130 bytes per vertex, meaning that our 10-billion-tetrahedron computation used 197GB of RAM.

Point Distributions. We tested our algorithm on several different distributions of data, including uniform, Gaussian, kuzmin, and line-singularity distributions. Details on these distributions can be found in [19]. For each distribution we ran $2^{24.5}$ vertices on one processor and $2^{27.5}$ vertices on eight processors. We computed the total runtime required and the number of additional bytes of memory per vertex allocated. (The numbers given are in addition to the 124.5 bytes per vertex required in all cases.) Results are shown in Table 8.3.

We also ran tests on real-world data: a set of grid points based on an octree decomposition generated by the Quake project [123]. The problem of computing a tetrahedral mesh over grid points proved difficult, as our algorithm was not designed to handle the perfectly flat tetrahedra that result when four vertices lie at the vertices of a square. To handle this we introduced small random perturbations: we added a small random value to each coordinate of each vertex.

Even after doing this, though, we encountered some difficulty with the tetrahedralization. Our insertion

Random Perturbation	Boundary Size	Contention (Rep Fails)	Time (sec)
(fully random)	9K	16M	3132
0—1	99K	20M	3054
0—.5	138K	62M	3230
0—.2	203K	316M	3895
0—.1	274K	836M	3954
0—.05	370K	2207M	7397
0—.02	525K	7976M	16544
0—.01	(too much contention, aborted)		

Table 8.4: Performance of our algorithm on 2^{27} fully random points (from the unit cube) versus 2^{27} points derived from the Quake project [123]. The points provided have a very large boundary, resulting in contention for the lock on the four bounding vertices. Adding randomness to the point locations makes less of the boundary "visible" to the boundary vertices, making the problem more tractable.

algorithm begins with a single tetrahedron on four artificial "boundary" vertices $v_1 \dots v_4$, chosen such that the tetrahedron contains all of the points to be inserted. As the points are inserted, the vertices $v_1 \dots v_4$ connect to the boundary of the mesh. For the random distributions we tested this did not pose a problem: the boundary of the mesh was no more than a few thousand vertices at most. For our octree-decomposition data, however, the boundary of the mesh was much larger. The degree of the vertices $v_1 \dots v_4$ grew large enough that there was significant contention between processors attempting to perform insertions near the boundary of the mesh. This decreased the performance of our algorithm considerably.

We were able to solve the problem by adding more randomness to the points. The smallest distance between any two points in our octree-decomposition data was 6 units; we added a random value between 0 and 1 to every vertex coordinate. By doing this we decreased the boundary of the mesh to a reasonable size. Results are shown in Table 8.4.

One interesting feature of our octree-decomposition data was its labeling. For random data, as a preprocessing step our algorithm relabels the points using x–y–z cuts (as described in Section 8.2). For our octree-decomposition data we found this step was unnecessary: the points came preordered, with a labeling that produced compression superior to what our relabeling algorithm provided. (For uniformly-random points with our reordering, the mesh data for 2^{27} vertices required 1.13G blocks for edge data. For the octree points with our reordering, mesh data required 1.09G blocks for edge data. For the octree points without reordering, mesh data required 0.97G blocks, which left the hashtables somewhat underfull.) Accordingly, the results in Table 8.4 use the labeling provided by the data.

8.5 Future Work

Out-Of-Core Algorithms. Our algorithm and data structure could be extended to work in an out-of-core setting: because the vertices are relabeled for locality, most of the memory accesses for an insertion should be very close together. Keeping the representation compressed means that more of it could fit in RAM at once. Unfortunately, our algorithm as described performs insertions in a random order. This could be improved by using a BRIO ("biased randomized insertion order") [5] to provide some locality between insertions. The work queues of our algorithm would need to be replaced with a series of $O(\log n)$ groups of work queues, one for each level of the BRIO.

High-Degree Meshes. We have shown that the algorithm behaves well on several distributions as long as the maximum degree of a vertex is bounded. When the mesh contains vertices of high degree (as for the octree-decomposition data, as discussed in Section 8.4), the competing threads suffer from contention for the high-degree vertices. Experimentally it seems that our code is tolerant of four vertices with total degree $138K$ (out of a total of $128M$ vertices shared among 8 processors), but performance suffers when the degree grows larger.

Part of this problem is from our coarse-grained locking mechanism: we divide our data structure into hashtables, and force threads to lock each hashtable they access. We allocate one hashtable per $G = 16$ vertices, and store the data for each edge in a hashtable corresponding to one of its vertices. We require that a thread acquire the lock on all of the vertices adjoining the cavity before performing an update. For correctness it is only necessary to lock the *edges* adjoining the cavity, not the vertices. A more conservative locking mechanism might be able to exploit this to tolerate high-degree vertices. However, to do this it would be necessary to distribute the edges of a high-degree vertex evenly among many hashtables, which might sacrifice the good memory-locality properties of the representation.

Also, even with this improvement, there exist 3D meshes in which *all* vertices have high degree. It is not clear how any parallel incremental-insertion algorithm could handle such meshes efficiently.

Chapter 9

Bibliography

References

[1] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Data Compression Conference (DCC)*, pages 203–212, 2001.

[2] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaig, and C. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1998.

[3] C. J. Alpert. The ISPD circuit benchmark suite. In *ACM International Symposium on Physical Design*, pages 80–85, Apr. 1998.

[4] C. J. Alpert and A. Kahng. Recent directions in netlist partitioning: A survey. *VLSI Journal*, 19(1–2):1–81, 1995.

[5] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proc. ACM Symposium on Computational Geometry*, June 2003.

[6] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 540–545, 1989.

[7] L. Arge. External memory data structures. In *Proc. European Symposium on Algorithms*, pages 1–29, 2001.

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

[9] B. Baumgart. A polyhedron representation for computer vision. In *Proc. National Computer Conference*, pages 589–596, 1975.

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

[11] R. Blanco and A. Barreiro. Document identifier reassignment through dimensionality reduction. In *ECIR*, pages 375–387, 2005.

[12] D. Blandford and G. Blelloch. Index compression through document reordering. In *Data Compression Conference (DCC)*, pages 342–351, 2002.

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

[14] D. Blandford, G. Blelloch, D. Cardoze, and C. Kadow. Compact representations of simplicial meshes in two and three dimensions. In *International Meshing Roundtable (IMR)*, pages 135–146, Sept. 2003.

[15] D. Blandford, G. Blelloch, and I. Kash. Compact representations of separable graphs. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 342–351, 2003.

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

[17] D. K. Blandford and G. E. Blelloch. Dictionaries using variable-length keys and data, with applications. In *Symposium on Discrete Algorithms*, 2005.

[18] G. Blelloch, H. Burch, K. Crary, R. Harper, G. Miller, and N. Walkington. Persistent triangulations. *Journal of Functional Programming (JFP)*, 11(5), Sept. 2001.

[19] G. Blelloch, J. Hardwick, G. L. Miller, and D. Talmor. Design and implementation of a practical parallel delaunay algorithm. *Algorithmica*, 24(3/4):243–269, 1999.

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

[21] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. *Computational Geometry*, 22(1–3):5–19, 2002.

[22] P. Boldi and S. Vigna. The webgraph framework I: Compression techniques, 2003.

[23] A. Bookstein, S. Klein, and T. Raita. Modeling word occurrences for the compression of concordances. In *Proc. IEEE Data Compression Conference*, page 462, Mar. 1995.

[24] A. Borodin, R. Ostrovsky, and Y. Rabani. Subquadratic approximation algorithms for clustering problems in high dimensional spaces. In *ACM Symposium on Theory of Computing*, pages 435–444, July 1999.

[25] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24:162–166, 1981.

[26] E. Brisson. Representing geometric structures in d dimensions: Topology and order. In *Proc. ACM Symposium on Computational Geometry*, pages 218–227, 1989.

[27] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Sixth Int'l. World Wide Web Conference*, pages 391–404, Cambridge, July 1997.

[28] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *WWW9 / Computer Networks*, 33(1–6):309–320, 2000.

[29] A. Brodnik and J. Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[30] A. L. Buchsbaum, R. Sundar, and R. E. Tarjan. Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, pages 40–49, 1992.

[31] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural merge sort. In *Proceedings of the International Symposium on Algorithms SIGAL'90*, pages 251–260, Tokyo, Japan, Aug. 1990.

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

[33] D. Chakrabarti, S. Papadimitriou, D. S. Modha, and C. Faloutsos. Fully automatic cross-associations. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004.

[34] D. Chakrabarti, Y. Zhan, D. Blandford, C. Faloutsos, and G. Blelloch. NetMine: New mining tools for large graphs. In *the SDM 2004 Workshop on Link Analysis, Counter-terrorism and Privacy*.

[35] M. Chen, T. Chuang, and J. Wu. Efficient parallel implementations of 2D Delaunay triangulation with high performance fortran. In *Proceedings of 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[36] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 1–12, 1999.

[37] N. Chrisochoides and D. Nave. Simultaneous mesh generation and partitioning for Delaunay meshes. In *Proceedings of the Eighth International Meshing Roundtable*, pages 55–66, 1999.

[38] N. Chrisochoides and D. Nave. Parallel Delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 58:161–176, 2003.

[39] N. Chrisochoides and F. Sukup. Task parallel implementation of the Bowyer-Watson algorithm. In *Proceedings of the Fifth International Conference on Numerical Grid Generation in Computational Fluid Dynamic and Related Fields*, 1996.

[40] 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.

[41] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4(1):387–421, 1989.
[42] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Comput*, 9:828–834, 1984.
[43] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. In *Proc. ACM Symposium on Computational Geometry*, pages 259–268, June 1998.
[44] M. Deering. Geometry compression. In *Proc. SIGGRAPH*, pages 13–20, 1995.
[45] F. K. H. A. Dehne, D. Hutchinson, A. Maheshwari, and W. Dittrich. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proc. IPPS/SPDP*, pages 14–20, 1999.
[46] Deo and Litow. A structural approach to graph compression. In *MFCS Workshop on Communications*, 1998.
[47] 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.
[48] D. Dobkin and M. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4(1):3–32, 1989.
[49] H. Edelsbrunner. *Geometry and Topology of Mesh Generation*. Cambridge Univ. Press, England, 2001.
[50] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, March 1975.
[51] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. Space efficient hash tables with worst case constant access time. In *STACS*, 2003.
[52] 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.
[53] P.-M. Gandoin and O. Devillers. Progressive and lossless compression of arbitrary simplicial complexes. In *Proc. SIGGRAPH*, 2002.
[54] S. Golomb. Run-length encodings. *IEEE Transactions on Information Theory, IT*, 12:399–401, July 1966.
[55] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 714–723, Nov. 1993.
[56] Google. Google programming contest web data. `http://www.google.com/programming-contest/`, 2002.
[57] 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.
[58] X. Gu. Harvard graphics archive—mesh library. `http://www.cs.deas.harvard.edu/~xgu/mesh/`.
[59] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.
[60] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.
[61] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *Proc. SIGGRAPH*, pages 133–140, 1998.
[62] H. Han and C.-W. Tseng. A comparison of locality transformations for irregular codes. In *Proc. Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 70–84, 2000.
[63] J. Hardwick. Implementation and evaluation of an efficient parallel Delaunay triangulation algorithm. In *Proceedings of Ninth Annual Symposium on Parallel Algorithm and Architectures*, 1997.
[64] X. He, M.-Y. Kao, and H.-I. Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *SIAM J. on Discrete Mathematics*, 12(3):317–325, 1999.
[65] 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.
[66] M. Isenburg and J. Snoeyink. Face fixer: Compressing polygon meshes with properties. In *Proc. SIGGRAPH*, pages 263–270, 2000.
[67] J. F. Antaki et. al. Sangria project. `http://www.cs.cmu.edu/~sangria`, 2002.
[68] G. Jacobson. Space-efficient static trees and graphs. In *30th FOCS*, pages 549–554, 1989.
[69] H. Kaplan and R. Tarjan. Purely functional representations of catenable sorted lists. In *Proc. of the 28th Annual*

ACM Symposium on the Theory of Computing, pages 202–211, May 1996.
[70] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. In *Proc. SIGGRAPH*, pages 279–286, 2000.
[71] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, International Computer Science Institute, 1995.
[72] K. Keeler and J. Westbrook. Short encodings of planar graphs and maps. *Discrete Applied Mathematics*, 58:239–252, 1995.
[73] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry – Theory and Applications*, 13:65–90, 1999.
[74] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, MA, 1973.
[75] J. Kohout, I. Kolingerov, and J. Zara. Practically oriented parallel delaunay triangulation in E2 for computers with shared memory. *Computers and Graphics*, 28:703–718, 2004.
[76] I. Kolingerova and J. Kohout. Optimistic parallel Delaunay triangulation. *The Visual Computer*, 18(8):511–5, 2002.
[77] S. Lee, C. Park, and C. Park. An improved parallel algorithm for delaunay triangulation on distributed memory parallel computers. *Parallel Processing Letters*, 11:341–352, 2001.
[78] F. T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems, with applications to approximation algorithms. In *FOCS*, pages 422–431, 1988.
[79] P. Lienhardt. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal of Computational Geometry and Applications*, 4(3):275–324, 1994.
[80] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.
[81] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Applied Mathematics*, 36:177–189, 1979.
[82] H.-I. Lu. Linear-time compression of bounded-genus graphs into information-theoretically optimal number of bits. In *SODA*, pages 223–224, 2002.
[83] S. McMains, J. M. Hellerstein, and C. H. Squin. Out-of-core build of a topological data structure from polygon soup. In *Proc. Symposium on Solid Modeling and Applications*, pages 171–182, June 2001.
[84] K. Mehlhorn and S. Naher. *LEDA: A platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
[85] 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.
[86] A. Moffat, O. Petersson, and N. C. Wormald. A tree-based mergesort. *Acta Informatica*, 35(9):775–793, 1998.
[87] A. Moffat and L. Stuiver. Exploiting clustering in inverted file compression. In *Proc. Data Compression Conference*, pages 82–91, Mar. 1996.
[88] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, July 2000.
[89] D. Muller and F. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.
[90] J. I. Munro. Tables. In *16th FST & TCS*, volume 1180 of LNCS, pages 37–42. Springer-Verlag, 1996.
[91] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th FOCS*, pages 118–126, 1997.
[92] M. Naor. Succinct representation for general unlabeled graphs. *Discrete Applied Mathematics*, 28:303–308, 1990.
[93] T. Okusanya and J. Peraire. 3D parallel unstructured mesh generation. *AMD*, 220:109–115, 1997.
[94] OpenMP. `http://www.openmp.org/`.
[95] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.
[96] R. Pagh. Low redundancy in static dictionaries with constant query time. *Siam Journal of Computing*,

31(2):353–363, 2001.

[97] R. Pagh and F. F. Rodler. Cuckoo hashing. In *ESA*, 2001.

[98] R. Pajarola, J. Rossignac, and A. Szymczak. Implant sprays: Compression of progressive tetrahedral mesh connectivity. In *Proc. Visualization 99*, pages 299–306, 1999.

[99] S. Papadomanolakis, A. Ailamaki, J. C. Lopez, T. Tu, D. R. O'Hallaron, and G. Heber. Efficient query processing on unstructured tetrahedral meshes, 2006.

[100] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[101] `rachel.psc.edu`. `http://http://www.psc.edu/machines/marvel/rachel.html/`.

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

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

[104] A. L. Rosenberg and L. S. Heath. *Graph Separators, with Applications*. Kluwer Academic/Plenum Publishers, 2001.

[105] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, /1999.

[106] J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1993.

[107] SCAN project. Internet maps. `http://www.isi.edu/ scan/mercator/maps.html`, 2000.

[108] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[109] J. Shewchuk. Pyramid mesh generator software. (`http://www.cs.berkeley.edu/˜jrs/`). Personal Communication.

[110] J. R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Proc. First Workshop on Applied Computational Geometry*, pages 124–133, Philadelphia, PA, May 1996.

[111] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–368, 1997.

[112] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[113] W.-Y. Shieh, T.-F. Chen, J. J.-J. Shann, and C.-P. Chung. Inverted file compression through document identifier reassignment. *Inf. Process. Manage.*, 39(1):117–131, 2003.

[114] I. Silicon Graphics. The C++ standard template library. `http://www.sgi.com/tech/stl/index.html`.

[115] F. Silvestri, R. Perego, and S. Orlando. Assigning document identifiers to enhance compressibility of web search engines indexes. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 600–605, New York, NY, USA, 2004. ACM Press.

[116] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[117] J. Sperling. Development and maintenance of the TIGER database: Experiences in spatial data sharing at the U.S. Bureau of the Census. In *Sharing Geographic Information*, pages 377–396, 1995.

[118] A. Strehl and J. Ghosh. A scalable approach to balanced, high-dimensional clustering of market-baskets. In *HiPC '00: Proceedings of the 7th International Conference on High Performance Computing*, pages 525–536, London, UK, 2000. Springer-Verlag.

[119] T. Suel and J. Yuan. Compressing the graph structure of the web. In *Data Compression Conference (DCC)*, pages 213–222, 2001.

[120] A. Szymczaka and J. Rossignac. Grow & Fold: compressing the connectivity of tetrahedral meshes. *Computer-Aided Design*, 32:527–537, 2000.

[121] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[122] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–726, 1997.

[123] T. Tu and D. R. O'Hallaron. A computational database system for generating unstructured hexahedral meshes with billions of elements. In *Proceedings of SC2004*, 2004.

[124] T. Tu, D. R. O'Hallaron, and J. C. Lopez. ETREE - a database-oriented method for generating large octree meshes. In *Proc. International Meshing Roundtable*, pages 127–138, Sept. 2002.

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

[126] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, MD, 1984.

[127] U.S. Census Bureau. UA Census 2000 TIGER/Line file download page. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html`, 2000.

[128] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[129] E. Voorhees and e. D.K. Harman. Overview of the eighth text retrieval conference (TREC-8). In *Proceedings of the Eighth Text REtrieval Conference (TREC-8)*, pages 1–24, 1999.

[130] C. Walshaw. Graph partitioning archive. `http://www.gre.ac.uk/~c.walshaw/partition/`, 2002.

[131] D. F. Watson. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. *The Computer Journal*, 24:167–172, 1981.

[132] D. Watts and S. Strogatz. Collective dynamics of small-world networks. *Nature*, 363:202–204, 1998.

[133] K. Weiler. Edge based data structures for solid modeling in a curved surface environment. *IEEE Computer Graphics and Applications*, 5(1):21–40, Jan. 1985.

[134] K. Weiler. The radial edge structure: A topological representation for non-manifold geometric boundary modeling. In *Geometric Modeling for CAD Applications*, pages 3–36. North-Holland, 1988.

[135] D. Wishart. Efficient hierarchical cluster analysis for data mining and knowledge discovery. *Computer Science and Statistics*, 30:257–263, July 1998.

[136] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images (Second Edition).* Morgan Kaufmann Publishing, San Francisco, 1999.