An Experimental Analysis of a Compact Graph Representation *

Daniel K. Blandford Guy E. Blelloch Ian A. Kash
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{blandford,blelloch,iak}@cs.cmu.edu

Abstract

In previous work we described a method for compactly representing graphs with small separators, which makes use of small separators, and presented preliminary experimental results. In this paper we extend the experimental results in several ways, including extensions for dynamic insertion and deletion of edges, a comparison of a variety of coding schemes, and an implementation of two applications using the representation.

The results show that the representation is quite effective for a wide variety of real-world graphs, including graphs from finite-element meshes, circuits, street maps, router connectivity, and web links. In addition to significantly reducing the memory requirements, our implementation of the representation is faster than standard representations for queries. The byte codes we introduce lead to DFT times that are a factor of 2.5 faster than our previous results with gamma codes and a factor of between 1 and 9 faster than adjacency lists, while using a factor of between 3 and 6 less space.

1 Introduction

We are interested in representing graphs compactly while supporting queries and updates efficiently. The goals is to store large graphs in physical memory for use with standard algorithms requiring random access. In addition to having applications to computing on large graphs (e.g. the link graph of the web, telephone call graphs, or graphs representing large meshes), the representations can be used for medium-size graphs on devices with limited memory (e.g. map graphs on a handheld 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 this.

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

Many methods have been proposed for compressing various specific classes of graphs. There have been many results on planar graphs and graphs with constant genus [32, 15, 13, 18, 14, 21, 8, 6]. These representations can all store an n-vertex unlabeled planar graph in $O(n)$ bits, and some allow for $O(1)$-time neighbor queries [21, 8, 6]. By unlabeled we mean that the representation is free to choose an ordering on the vertices (integer labels from 0 to $n-1$). To represent a labeled graph one needs to additionally store the vertex labels. Other representations have been developed for various other classes of graphs [15, 12, 24, 30]. A problem with all these representations is that they can only be used for a limited class of graphs.

In previous work we described a compact representation for graphs based on graph separators [5]. For unlabeled graphs satisfying an $O(n^c)$, $c < 1$ separator theorem, the approach uses $O(n)$ bits and supports neighbor or adjacency queries in $O(1)$-time per edge. A property of the representation, however, is that it can be applied to any graph, and the effectiveness of compression will smoothly degrade with the quality of the separators. For random graphs, which don't have small separators (in expectation), the space requirement asymptotically matches the informational theoretical lower bound. This smooth transition is important in practice since many real-world graphs might not strictly satisfy a separator theorem, but still have good separator properties. In fact, since many graphs only come in a fixed size, it does not even make sense to talk about separators theorems, which rely on asymptotic characteristics of separators. As it turns out, most "real world" graphs do have small separators (significantly smaller than expected from a random graph). This is discussed in Section 1.1.

This paper is concerned with the effectiveness of the separator-based representation in practice. We extend the previous approach to handle dynamic graphs (edge insertions and deletions) and present a more complete set of experiments, including a comparison of different

prefix codes, comparison on two machines with different cache characteristics, a comparison with several variants of the adjacency-list representation, and experimental results of two algorithms using the representation. 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 2 we review our previous representation as applied to edge-separators. The representation uses a separator tree for labeling the vertices of the graph and uses difference codes to store the adjacency lists. In Section 3 we describe our implementation, including a description of the prefix codes used in this paper. In Section 4 we describe an extension of the separator-based representation that supports dynamic graphs, *i.e.*, the insertion and deletion of edges. Our original representation only supported static graphs. The extension involves storing the bits for each representation in fixed-length blocks and linking blocks together when they overflow. A key property is that the link pointers can be kept short (one byte for our experiments). The representation also uses a cache to store recently accessed vertices as uncompressed lists.

In Sections 5 and 6 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 use four codes for encoding differences: gamma codes, snip codes, nibble codes, and byte codes (only gamma codes were reported in our previous paper). The different codes present 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 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 7 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 Google page-rank algorithm. 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.

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 [23] 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 [2], nested dissection for solving linear systems [16], partitioning graphs on to parallel processors [27], clustering [29], and computer vision [26]. Although finding a minimum separator for a graph in NP-hard, there are many algorithms and codes that find good approximations [23]. Here we briefly review why graphs have good separators.

Many graphs have good separators 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 [36] conjecture that locality is one of the main properties of graphs based on social networks.

Many graphs have good separators because 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. Two dimensional meshes are often planar (although not always) and hence satisfy an $O(n^{1/2})$ vertex-separator theorem [17]. Well shaped three dimensional meshes are known to satisfy an $O(n^{2/3})$ vertex-separator theorem [20]. 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 [33].

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

2 Encoding with Separators

In previous work [5] we described an $O(n)$-bit encoding with $O(1)$ access time for graphs satisfying either an n^c, $c < 1$ edge or vertex separator theorem. In this paper we only consider the simpler version based on edge separators. Here we review the algorithm for edge-separators.

Edge Separators. Let S be a class of graphs that is closed under the subgraph relation. We say that 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 [17].

Given a graph G it is possible to build a *separator tree*. Each node of the tree contains a subgraph of G and a separator for that subgraph. The children of a node contain the two components of the graph induced by the separator. The leaves of the tree are single nodes.

Without loss of generality we will consider only graphs in which all vertices have nonzero degree. We will also assume the existence of a graph separator algorithm that returns a separator within the $O(n^c)$ bound.

Adjacency Tables. Our data structures make use of an encoding in which we store the neighbors for each vertex in a difference-encoded adjacency list. 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 as a sequence of bits. The differences are encoded using any *logarithmic code*, that is, a prefix code which uses $O(\log d)$ bits to encode a difference of size d. 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. Finding these locations efficiently (in time and space) requires a separate *indexing structure* [5]. For the experiments in this paper we use an indexing structure (see Section 3) that is not theoretically optimal, but works well in practice and is motivated by the theoretically optimal solutions.

Graph Reordering. Our compression algorithm works as follows:

1. Generate an edge separator tree for the graph.

2. Label the vertices in-order across the leaves.

3. Use an adjacency table to represent the relabeled graph.

LEMMA 2.1. *[5] For a class of graphs satisfying an n^c-edge separator theorem, and labelings based on the separator tree satisfying the bounds of the separator theorem, the adjacency table for any n-vertex member requires $O(n)$ bits.*

3 Implementation

Separator trees. There are several ways to compute a separator tree from a graph, depending on the separator algorithm used. In our previous paper we tested three separator algorithms and described a "child-flipping" postprocessing heuristic which could be used to improve their performance. Here we use the "bottom-up" separator algorithm with child-flipping. This algorithm gave the best performance on many of our test graphs while being significantly faster than the runner-up for performance. Generating the separator tree and labeling with this algorithm seems to take linear time and takes about 15 times as long as a depth-first search on the same graph.

The bottom-up algorithm 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. We do not know of any theoretical bounds on this or similar separator algorithms.

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 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 structure. Our static algorithms use an indexing structure to map the number of a vertex to the bit position of the start of the appropriate adjacency list. In our previous paper we tested several types of indexing structure and demonstrated a tradeoff between space used and lookup speed. Here we use a new structure called *semi-direct-16* which 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.

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.

This indexing method saves about six bits per vertex over our previous *semidirect* index while causing almost no slowdown.

Codes and Decoding. We considered several logarithmic codes for use in our representations. In addition to the *gamma code* [9], which we used in our previous experiments, we implemented *byte codes*, *nibble codes*, and *snip codes*—three closely related codes of our own devising.

Gamma codes store an integer d using a unary code for $\lceil \log d \rceil$ followed by a binary code for $d - 2^{\lceil \log d \rceil}$. This uses a total of $1 + 2\lfloor \log d \rfloor$ bits. Assuming the machine word size is at least $\log d$ bits, gamma codes can be decoded in constant time using table lookup.

Decoding the gamma codes is the bottleneck in making queries. To reduce the overhead we devised three codes, snip, nibble, and byte codes, that better take advantage of the fact that machines are optimized to manipulate bytes and words rather than extract arbitrary bit sequences. 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 if is less 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 (byte code) is particularly fast to encode and decode since all memory accesses are to bytes. The 4-bit version (nibble code) and 2-bit version (snip code) are somewhat slower since they require more accesses and require extracting pieces of a byte.

We also considered using Huffman and Arithmetic codes which are based on the particular distribution at hand. However, we had used these codes in previous work [3] and found that although they save a little space over gamma codes (about 1 bit per edge for arithmetic codes), they are more expensive to encode and decode. Since our primary goal was to improve time performance, we did not implement these codes.

4 Dynamic Representation

Here we present a data structure that permits dynamic insertion (and deletion) of edges in the graph. 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 size of a vertex can change with each update, so it is necessary to dynamically assign memory to vertices.

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. Under certain assumptions about the hash function, if the memory pool is at most 80% full, then the probability that this technique will fail is at most $.80^{128} \simeq 4 * 10^{-13}$.

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.

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

Graph	Vtxs	Edges	Max Degree	Source
auto	448695	6629222	37	3D mesh [35]
feocean	143437	819186	6	3D mesh [35]
m14b	214765	3358036	40	3D mesh [35]
ibm17	185495	4471432	150	circuit [1]
ibm18	210613	4443720	173	circuit [1]
CA	1971281	5533214	12	street map [34]
PA	1090920	3083796	9	street map [34]
googleI	916428	5105039	6326	web links [10]
googleO	916428	5105039	456	web links [10]
lucent	112969	363278	423	routers [25]
scan	228298	640336	1937	routers [25]

Table 1: Properties of the graphs used in our experiments.

5 Experimental Setup

Graphs. We drew test graphs for our experiments from several sources: 3D Mesh graphs from the online Graph Partitioning Archive [35], street connectivity graphs from the Census Bureau Tiger/Line data [34, 28], graphs of router connectivity from the SCAN project [25], graphs of webpage connectivity from the Google [10] programming contest data, and circuit graphs from the ISPD98 Circuit Benchmark Suite [1]. 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 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 spacial locality (even more than the factor of 3.4 in processor speed would indicate), but not particularly well on the experiments without spacial locality. All code is written in C and C++ and compiled using g++ (3.2.3) using 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 them 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 [19] 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 spacial locality.

In our experiments we measured DFS runtimes after inserting the edges in three orders: linear, transpose, and random. These insertion orders are describe above. The insertion orders have a major affect 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 spacial 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 spacial 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.

	Array			Our Structure									
	Rand	Sep		Byte		Nibble		Snip		Gamma		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	1.063	7.18	0.399	12.33
feocean	0.048s	0.312	37.60	0.312	12.79	0.604	10.86	0.791	11.12	1.0	11.97	0.374	13.28
m14b	0.103s	0.388	34.05	0.349	10.01	0.728	7.10	0.970	6.55	1.320	6.68	0.504	11.97
ibm17	0.095s	0.536	33.33	0.536	10.19	1.115	7.72	1.400	7.58	1.968	7.70	0.747	12.85
ibm18	0.113s	0.398	33.52	0.442	10.24	0.867	7.53	1.070	7.18	1.469	7.17	0.548	12.16
CA	0.920s	0.126	43.40	0.146	14.77	0.243	10.65	0.293	10.55	0.333	11.25	0.167	14.81
PA	0.487s	0.137	43.32	0.156	14.76	0.258	10.65	0.310	10.60	0.355	11.28	0.178	14.80
lucent	0.030s	0.266	41.95	0.3	14.53	0.5	11.05	0.566	10.79	0.700	11.48	0.333	14.96
scan	0.067s	0.208	43.41	0.253	15.46	0.402	11.84	0.477	11.61	0.552	12.14	0.298	16.46
googleI	0.367s	0.226	37.74	0.258	11.93	0.405	8.39	0.452	7.37	0.539	7.19	0.302	13.39
googleO	0.363s	0.250	37.74	0.278	12.59	0.460	9.72	0.556	9.43	0.702	9.63	0.327	13.28
Avg		0.287	38.202	0.302	12.501	0.561	9.357	0.696	9.07	0.909	9.424	0.380	13.662

Table 2: 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.

LEDA. We also ran all our experiments using LEDA [19] 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 [19, 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.

6 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 where the user needs control of the ordering of the out-edges. We make use of this in a compact representation of simplicial meshes [4].

6.1 Static representations Table 2 presents results comparing space and DFS times for the static representations for all the graphs on the Pentium 4. Tables 5 and 6 present summary results for a wider set of operations on both the Pentium III and Pentium 4. In Table 2 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 spacial locality has been previously studied for use in sparse-matrix vector multiply [31, 11], 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.

	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 3: 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. .

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 worsens 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 6). 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, later in the section, describes the time cost of simply reading all the edges in a graph (without the effect of cache locality).

6.2 Dynamic representations A key parameter for the dynamic representation is selecting the block size. 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 3 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 4 presents results comparing space and DFS times for the dynamic representations for all the graphs on the Pentium 4. Tables 5 and 6 give summary results for a wider set of operations on both the Pentium III and Pentium 4.

Table 3 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.[1] Space Opt uses the

[1] We actually pick a setting that optimizes T^3S where T is time

	Linked List							Our Structure					
	Random Vtx Order			Sep Vtx Order				Space Opt			Time Opt		
Graph	Rand T_1	Trans T/T_1	Lin T/T_1	Rand T/T_1	Trans T/T_1	Lin T/T_1	Space	Block Size	Time T/T_1	Space	Block Size	Time 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 4: 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.

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 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. [19, page 268] and [7]) 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 a user 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 below.

Summary. Tables 5 and 6 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,

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.

Graph	DFS	Read		Find Next	Insert			Space
		Linear	Random		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: Summary of space and normalized times for various operations on the Pentium 4.

Graph	DFS	Read		Find Next	Insert			Space
		Linear	Random		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 6: Summary of space and normalized times for various operations on the Pentium III.

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 is 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.

7 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 [22]. 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 7 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.

Representation	Time (sec)		Space
	PIII	P4	(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 7: Performance of our PageRank algorithm on different representations.

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 [19, page 372]. We also implemented an optimization that does one level of BFS before the DFS. This improved performance by 40%. Finally we used a strided loop through the left vertices, using a prime number (11) as the stride. This reduces locality, but greatly improved performance since the average since 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 (semi-direct-16) as

Representation	Time (sec)		Space
	PIII	P4	(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 8: Performance of our bipartite maximum matching algorithm on different static representations.

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 8. 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.

8 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 [16]. For nested dissection more sophisticated separators 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. Small separators 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 cost for even a rather simple graph algorithm, DFS. 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 labeling that the separators give 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 on 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 paper can clearly be generalized to other structures beyond simple graphs. A separate paper [4] describes how similar ideas can be used for representing simplicial meshes in two and three dimensions.

References

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

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

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

[4] 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.

[5] 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.

[6] Y.-T. Chiang, C.-C. Lin, and H.-I. Lu. Orderly spanning trees with applications to graph encoding and graph drawing. In *SODA*, pages 506–515, 2001.

[7] 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.

[8] 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.

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

[10] Google. Google programming contest web data. `http://www.google.com/programming-contest/`, 2002.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.

[17] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Applied Mathematics*, 36:177–189, 1979.

[18] H.-I. Lu. Linear-time compression of bounded-genus graphs into information-theoretically optimal number of bits. In *SODA*, pages 223–224, 2002.

[19] K. Mehlhorn and S. Naber. *LEDA: A platfor for combinatorial and geometric computing*. Cambridge University Press, 1999.

[20] 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.

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

[22] 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.

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

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

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

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

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

[28] 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.

[29] A. Strehl and J. Ghosh. A scalable approach to balanced, high-dimensional clustering of market-baskets. In *Proc. of the Seventh International Conference on High Performance Computing (HiPC 2000)*, volume 1970 of *Lecture Notes in Computer Science*, pages 525–536. Springer, Dec. 2000.

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

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

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

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

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

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

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