# Compact Representations of Separable Graphs [*]

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**

We consider the problem of representing graphs compactly while supporting queries efficiently. In particular we describe a data structure for representing $n$-vertex unlabeled graphs that satisfy an $O(n^c)$-separator theorem, $c < 1$. The structure uses $O(n)$ bits, and supports adjacency and degree queries in constant time, and neighbor listing in constant time per neighbor. This generalizes previous results for graphs with constant genus, such as planar graphs.

We present experimental results using many "real world" graphs including 3-dimensional finite element meshes, link graphs of the web, internet router graphs, VLSI circuits, and street map graphs. Compared to adjacency lists, our approach reduces space usage by almost an order of magnitude, while supporting depth-first traversal in about the same running time.

## 1 Introduction

Many applications involve representing very large graphs. In such applications space can be as much of an issue as time. For example, there has been recent interest in compactly representing the link structure of the web for use in various algorithms [4, 1, 25]. Other applications of large graphs include representing the meshes used in finite-element simulations, or 3d models in graphics. For all these applications it is often useful to support dynamic queries efficiently while maintaining a compact representation.

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 [30], 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 graphs in practice have is that they have small separators. 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, for example, have $O(n^{1/2})$ separators [17] 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 [19], as do most nearest neighbor graphs in 3-dimensions. Furthermore many graphs without pre-defined embeddings in low dimensional spaces have small separators [29]. For example, the link structure of the web has small separators, as our experiments show.

In this paper we are interested in compact representations of separable graphs (a graph is separable if it is taken from a class of graphs that satisfies an $n^c$-separator theorem [17] for $c < 1$). We describe a representation that uses $O(n)$ bits and supports constant time adjacency and degree queries, and listing of the neighbors in constant time per neighbor. 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 algorithm is likely to do well since it will compress the components that are separable. Our computational

[*]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.

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.

Our data structure is based on recursively separating a graph and using the separators to renumber the vertices (first numbering one subgraph, then the separator vertices, then the other subgraph). 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. In particular we can store an adjacency list by sorting the neighbor indices, and storing the differences $d$ between adjacent pairs in the list using an $O(\log d)$-bit encoding. For separable graphs with constant in-degree, we show that this idea is sufficient for supporting degree and neighbor queries efficiently. For graphs with non-constant in-degree, this representation is not space-efficient. We show, however, that by using "shadow labels" for the vertices and having the adjacency lists refer to these shadow labels, $O(n)$ bits is sufficient. We achieve constant-time adjacency queries by directing the graph so that all vertices have constant in-degree.

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 [15] gives a polynomial time construction 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 [17] and well shaped meshes [19]. In practice fast heuristics work well for most graphs [13].

We implemented a version of our data structure without shadow labels and present results on several graphs from a variety of sources. We compare several methods for finding separators and for indexing the structure. As a measure of query times we use the time taken for a depth-first search. We compare our space and time to that of adjacency list representations using both linked lists and arrays. Our query time is about the same as for the linked list structure, but slower than arrays. In space we save a factor of up to 10 compared to linked lists, and 5 compared to arrays. The time needed to find a good ordering is equivalent to the time of about 15 depth-first searches.

**Related Work.** There has been considerable previous work on compressing unlabeled graphs. Turan [26] 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 [14], and He, Kao and Lu [11] later describe a technique that is optimal in the first order term. These results generalize to any graph with constant genus [18]. There have also been many results for sub-classes of planar graphs such as trees, triangulated meshes or triconnected planar graphs [14, 10, 23]. For each of these, the constant in front of the $m$ can be improved over general planar graphs. For dense graphs, Naor [22] describes a representations that reduces a lower order term over what is required by an adjacency matrix.

None of this work considers implementing fast queries. Jacobson [12] first showed how planar graphs can be represented using $O(n)$ bits while permitting adjacency queries in $O(\log n)$ time. Munro and Raman [21] improved the time for adjacency queries to $O(1)$ time. The constants on the high order term for the space bound has been improved by Chuang et. al. [6], and further by Chiang et. al. [5]. All of these techniques are 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 [7] showed that separators can be used to compress graphs with bounded genus to $O(n)$ bits. He, Kao and Lu [11] use planar-graph separators to compress planar graphs to the optimal number of bits within a low-order term. Neither of these techniques, however, support queries. In our previous work [3] we showed how separators can be used to compress the bipartite graph representing the mapping of terms to documents in a document database, such as used by web search engines. Our use of a separator tree to number the vertices is similar to the Gaussian elimination order generated by nested dissection [16]. In fact, in the special case of constant in-degree we could use the nested dissection ordering directly—it numbers the separator vertices after the two recursive calls, but this does not affect our bounds.

## 2 Preliminaries

**Graph Separators.** Let $S$ be a class of graphs that is closed under the subgraph relation. We say that $S$ satisfies 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 $\alpha n$ vertices each [17].

In this paper 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. Our results will apply to other classes as well: for example, Miller et al. [19] demonstrated that every well-shaped mesh in $\mathbb{R}^d$ has a separator of size $O(n^{1-1/d})$. We will say a graph is *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 [16] 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.* We say that 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 $\alpha n$ 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. We will mostly deal with vertex separators, but we will show simplified results for graphs with good edge separators.

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.

**Queries.** We will consider three kinds of queries: adjacency queries, neighborhood queries, and degree queries. An adjacency query tests whether two vertices are adjacent. A neighborhood query lists all the neighbors of a given vertex. A degree query returns the degree of a vertex.

**Rank and Select.** Our algorithm will require the use of data structures that support the *rank* and *select* operations [12]. Given $S \subset 1 \dots n$,

`Rank(j)` returns the number of elements of $S$ that are less than or equal to $j$, and

`Select(i)` returns the $i$th smallest element in $S$.

These operations can be implemented to run in constant time with $O(n)$-bit data structures [20]. In Section 5 we describe a simpler implementation of a *select* data structure which matches these bounds.

**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 $v_1, v_2, v_3, \dots, v_d$ in sorted order, then the data structure encodes the differences $v_1 - v$, $v_2 - v_1$, $v_3 - v_2$, $\dots$, $v_d - v_{d-1}$ contiguously in memory as a sequence of bits. The differences are encoded using the *gamma code* [8], which uses $1+2\lfloor \log d \rfloor$ 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 gamma 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 $O(\log(n))$ pointer for each vertex would exceed our space bound; instead, we use a *select* data structure to store the start locations using $O(n)$ bits ($S$ is the set of start locations).

Lemma 2.1. *An adjacency table supports degree queries in $O(1)$ time, and neighborhood queries in $O(d)$ time, where $d$ is the degree of the vertex.*

*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 $O(\log(n))$-bit words (for example with table lookup), so it takes $O(d)$ time to decode the contents of the list.

Lemma 2.2. *If the adjacency list for $v$ is gamma-coded, then adding a neighbor $v'$ to the list adds $O(\log(|v-v'|))$ bits, changing a neighbor's position in the list from $v'$ to $v''$ adds $O(\log(|v' - v''|))$ bits, and deleting a neighbor from the list adds $O(1)$ bits to the structure.*

*Proof.* Follows from the fact that gamma codes require $\Theta(\log(d))$ bits, and since the logarithm function is concave. We note that deleting a neighbor from an adjacency list can actually increase the number of bits by 1.

## 3 Separators and Neighborhood Queries

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 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 [16]. 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 2.

```
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($T_a, V_{sep}, T_b$)
```

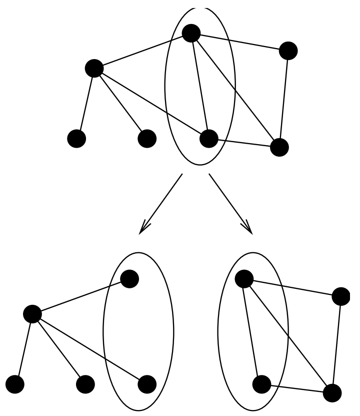

Figure 1: The BuildTree algorithm, and an example of the partition it produces.

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 vertex. (The labeling of representatives is sparse, but we can use the *select* and *rank* data structures 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 3.1. *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 an adjacency list for each vertex, which is accessed using the root label of the vertex. If vertices $u$ and $v$ have a 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 3.1. *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 $\pm 1$. 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 3.1), so the gamma 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 $\beta n^c$ separator that guarantees that each side of the partition will contain at most $\alpha n$ 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 $O(n)$.

Before we describe the root-find structure, we will discuss two special cases in which it is not needed. For these cases we only need a single label for each vertex, and need not generate any shadow labels. The adjacency lists point directly to the "representative" label of the vertex, and we can directly support neighborhood queries in constant time per neighbor.

The first case is for directed graphs with bounded in-degree. (We will consider a separator on a directed graph to be equivalent to that for the corresponding undirected graph.) In this case, we build an *out-edge adjacency table* that, for any vertex $v$, lists the root label $u$ corresponding to each out-edge $(v, u)$.

Lemma 3.2. *For a class of directed graphs with bounded in-degree and satisfying an $n^c$-separator theorem, any $n$-vertex member can be encoded in $O(n)$ bits using an out-edge adjacency table.*

*Proof.* A shadow adjacency table uses $O(n)$ bits on this graph. Discarding the in-edges produces a "shadow out-edge adjacency table" also using $O(n)$ bits. By Lemma 2.2 the cost of replacing any shadow label $v'$ with its root label $v$ is $O(\log(|v - v'|))$ bits. If $v$ appears as a separator in a graph with $s$ vertices, then this cost is $O(\log(s))$. Each vertex is charged once for each in-edge it has. Since a vertex's in-degree is bounded, this cost is $O(\log(s))$ per vertex. This gives the same recurrence as in Lemma 3.1 and hence $O(n)$ additional bits.

The second special case is for undirected graphs with good edge separators. To label the vertices we use an *edge separator tree* rather than the vertex separator tree produced above. Each vertex appears in one leaf of the tree and is assigned a label based on an in-order traversal.

Lemma 3.3. *For a class of graphs satisfying an $n^c$-*edge *separator theorem, any $n$-vertex member can be encoded in $O(n)$ bits using an adjacency table.*

*Proof.* For each edge $(u, v)$ in a separator in a graph with $s$ vertices, the difference between $u$ and $v$ is $O(s)$, so the contribution of that edge to the adjacency lists of $u$ and $v$ is $O(\log(s))$ bits. As above, we charge $O(\log(s))$ to every edge in a separator of a graph with $s$ vertices; the total is $O(n)$ bits.

**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, gamma 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 that 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 3.1). We charge the two child pointers to the parent, resulting in the same recurrence as in Lemma 3.1 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 go 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 go 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 go in the third category. Rather than encoding these vertices explicitly, we will encode the graphs they appear in. 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

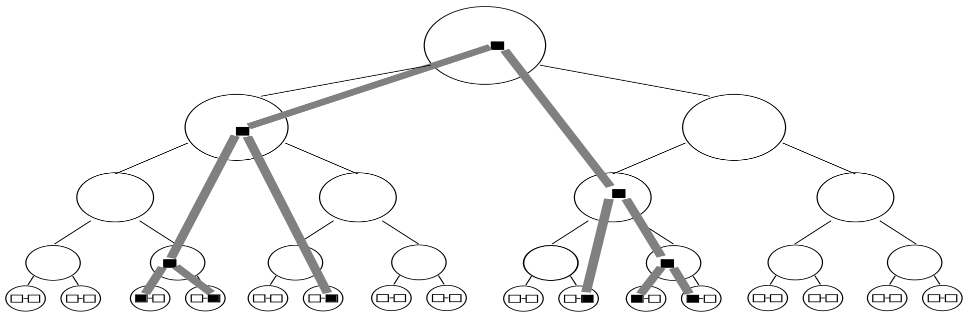

Figure 2: The separator tree, and a shadow tree corresponding to a vertex of degree 6.

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 3.4. *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.

## 4 Adjacency Queries

Using the structures 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 convert the target graph to a directed graph with bounded in-degree.

Lemma 4.1. *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 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 $2b$.

Theorem 4.1. *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.* We direct our graph for bounded in-degree using Lemma 4.1. For neighborhood queries on undirected graphs we need to store an edge in both directions. We therefore partition the edges into two sets: those that point in the direction of bounded in-degree and those that point in the other direction—these will have bounded out-degree. For the first set we can use Lemma 3.2 and skip the root-find structure, if desired. For the second set we need the root-find structure but the adjacency lists are constant length. (By Lemma 3.1, discarding entries can increase the space of the table by at most $O(1)$ bit per edge.) For neighborhood queries we search both tables. This takes $O(d)$ time (Lemmas 2.1 and 3.4). 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. Degree queries are handled by Lemma 2.1.

## 5 Experimental Setup

We were interested in analyzing the effectiveness of our approach and exploring various space time tradeoffs. We decided to implement the version of our data structure based on edge separators (see Lemma 3.3) since the graphs we test have reasonably good edge separators, and this version avoids the need for the somewhat complicated root-find structure. In this section we describe the experimental setup including the algorithms we used to find separator trees, a heuristic for improving the ordering given a separator tree, the select data structures used to index into the adjacency table, and the graphs used in our experiments. In the next section we discuss the actual results.

**Separator Algorithms.** 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 [13] 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 [28]. We apply Metis recursively to produce a separator tree.

Our third algorithm, *bu*, 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 contains 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.

**Child-flipping.** 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.

**Codes and Decoding.** We use table lookup to decode multiple gamma codes at once. For a fixed number of bits $k_g$ we create a table of size $2^{k_g}$. In each entry we place the number of values that the $k_g$ bits decode to (starting at one end), the actual values, and the number of bits ($\leq k_g$) to skip over to find the next codeword (since the next codeword could start in the current $k_g$ bits). If the number of values is 0, then the codeword does not fit into $k_g$ bits and it is decoded explicitly. In our implementation we limit the number of words decoded at once to 3, and the magnitude of each value to fit in 8 bits. This makes it possible to fit each table entry into 32 bits.

There are several other codes that could be used instead of gamma codes, including delta [8], Huffman, and arithmetic codes. We expect some of these could improve compression, but we leave this for future work.

**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 three 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 used $\Theta(\log(n))$ 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. The

| Graph | Vtxs | Edges | Max Degree | Source |
|---|---|---|---|---|
| auto | 448695 | 3314611 | 37 | 3D mesh [28] |
| feocean | 143437 | 409593 | 6 | 3D mesh [28] |
| m14b | 214765 | 1679018 | 40 | 3D mesh [28] |
| ibm17 | 185495 | 2235716 | 150 | circuit [2] |
| ibm18 | 210613 | 2221860 | 173 | circuit [2] |
| CA | 1971281 | 2766607 | 12 | street map [27] |
| PA | 1090920 | 1541898 | 9 | street map [27] |
| googleI | 916428 | 5105039 | 6326 | web links [9] |
| googleO | 916428 | 5105039 | 456 | web links [9] |
| lucent | 112969 | 181639 | 423 | routers [24] |
| scan | 228298 | 320168 | 1937 | routers [24] |

Figure 3: The graphs used in our experiments.

second structure, *semi-direct*, uses a one-word pointer per four vertices and fits three offsets into a second word. If the offsets don't all fit, they are stored elsewhere, and the second word is a pointer to them.

We also implemented a structure, *indirect*, that uses optimal $O(n)$ bits and constant time. This is significantly simpler than the structure of Munro [20]. 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.

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.

**Graphs.** We drew test graphs for this project from several sources: 3D Mesh graphs from the online Graph Partitioning Archive [28], street connectivity graphs from the Census Bureau Tiger/Line [27], graphs of router connectivity from the SCAN project [24], graphs of webpage connectivity from the Google [9] programming contest data, and circuit graphs from the ISPD98 Circuit Benchmark Suite [2]. 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 Figure 5. For the google graphs we list the number of directed edges, and we take the degree of a vertex to be the number of elements in its adjacency list. The I and O postfix refers to representing the incoming and outgoing edges respectively.

## 6 Experimental Results

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 and the time needed for queries (using more space for the indexing structure gives faster query times). Our experiments demonstrate these tradeoffs.

Our experiments were conducted on a MicroPro, with an Intel Pentium III 1Ghz CPU, and 512MB PC133 Memory with 32-bit word size. For the undirected graphs we represent the edge in both directions. The bits/edge are reported for each direction. The $k_g$ for table lookup is 16.

Figure 4 illustrates the tradeoff between the time needed to generate an ordering, and the space needed by the compressed adjacency lists that use that ordering. In addition to the separator schemes discussed in Section 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.

Figure 5 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 in-

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

Figure 4: The performance (time used and compression achieved) of several of our ordering algorithms. Space is in bits per edge for encoding the edges; $T_d$ 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).

|  | List | Array | Direct |  | Semi |  | k=1 |  | k=16 |  |
|---|---|---|---|---|---|---|---|---|---|---|
|  | $T_\ell$ | $T/T_\ell$ | $T/T_\ell$ | Space | $T/T_\ell$ | Space | $T/T_\ell$ | Space | $T/T_\ell$ | 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: The performance of various direct and indirect indexing schemes. Space is measured in bits per edge; $T_\ell$ is in seconds and the other times are normalized to it.

dices of each vertex contiguously in one large array with the lists for the vertices placed one after 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 if it has been visited.

The results show that for the direct and semi-direct 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 spacial 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 semi-direct indexing structure saves a factor of two in space over the direct structure while requiring little extra time. The indirect indexing structure ($k$ = subblock size) further improves the space overhead, but significantly increases the running time, especially for the large $k$.

| Graph | Array | | List | | bu-cf/semi | |
|---|---|---|---|---|---|---|
| | time | space | time | space | time | space |
| auto | 0.24 | 34.2 | 0.61 | 66.2 | 0.51 | 7.17 |
| feocean | 0.04 | 37.6 | 0.08 | 69.6 | 0.09 | 11.75 |
| m14b | 0.11 | 34.1 | 0.29 | 66.1 | 0.24 | 6.70 |
| ibm17 | 0.15 | 33.3 | 0.40 | 65.3 | 0.34 | 7.72 |
| ibm18 | 0.14 | 33.5 | 0.38 | 65.5 | 0.32 | 7.33 |
| CA | 0.34 | 43.4 | 0.56 | 75.4 | 0.58 | 11.66 |
| PA | 0.19 | 43.3 | 0.31 | 75.3 | 0.32 | 11.68 |
| googleI | 0.24 | 37.7 | 0.49 | 69.7 | 0.45 | 7.86 |
| googleO | 0.24 | 37.7 | 0.50 | 69.7 | 0.51 | 9.90 |
| lucent | 0.02 | 42.0 | 0.04 | 74.0 | 0.05 | 11.87 |
| scan | 0.04 | 43.4 | 0.06 | 75.4 | 0.08 | 12.85 |

Figure 6: A summary of space and time performance. Space is in bits per edge and includes both the adjacency lists and indexing structure. Time is in seconds for a depth-first search.

Overall when the time for creating the ordering is not critical, *bu-cf* ordering with semi-direct indexing seems to present the best tradeoff between time and space. Otherwise *bu-bpq* is likely better. Table 6 summarizes the results for the *bu-cf* ordering.

## References

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] Deo and Litow. A structural approach to graph compression. In *MFCS Workshop on Communications*, 1998.

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

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

[10] 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.

[11] 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.

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

[13] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, 1995.

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

[15] 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.

[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] 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.

[20] J. I. Munro. Tables. In *16th FST & TCS*, volume 1180 of LNCS, pages 37–42. Springer-Verlag, 1996.

[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] M. Naor. Succinct representation for general unlabeled graphs. *Discrete Applied Mathematics*, 28:303–308, 1990.

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

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

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

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

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

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

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

[30] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufman, 1999.