An Experimental Analysis of a Compact Graph Representation

Abdel Kareem Dabbas

Authors

Daniel
Blandford

Guy
Blelloch

Ian Kash

Motivation

Real graphs are large & sparse: we want them in RAM for algorithms that need random access.

Even without RAM pressure, layout ⇒ cache locality ⇒ speed.

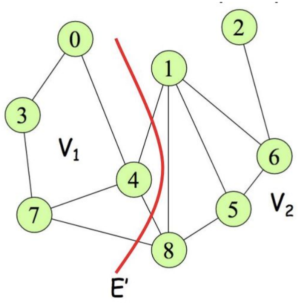
Goal: one representation that’s compact and supports fast traversal/queries; also a dynamic variant for updates

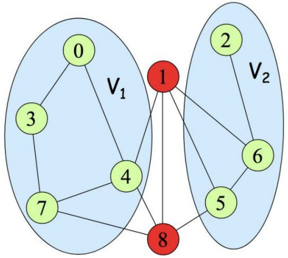
Separators

Edge separator: a set of edges whose removal splits the graph into two balanced parts

Vertex separator: a set of vertices whose removal splits the graph into two balanced parts

Minimum Separator: the separator that minimizes the number of edges/vertices removed

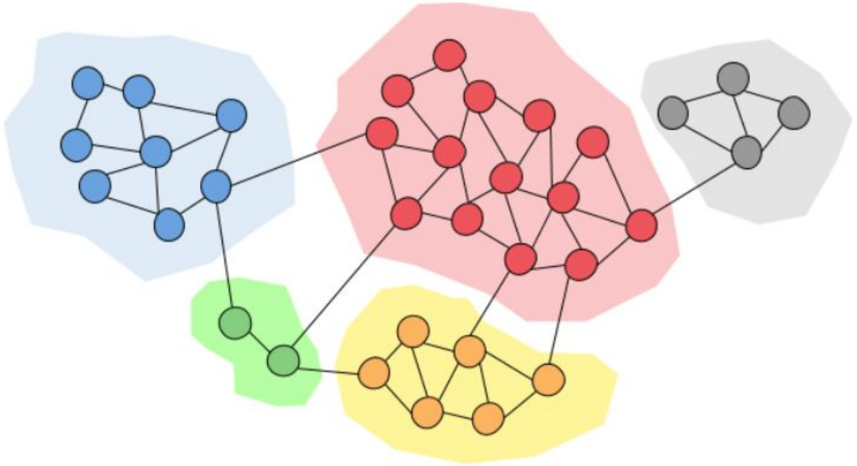




Why real graphs have good separators

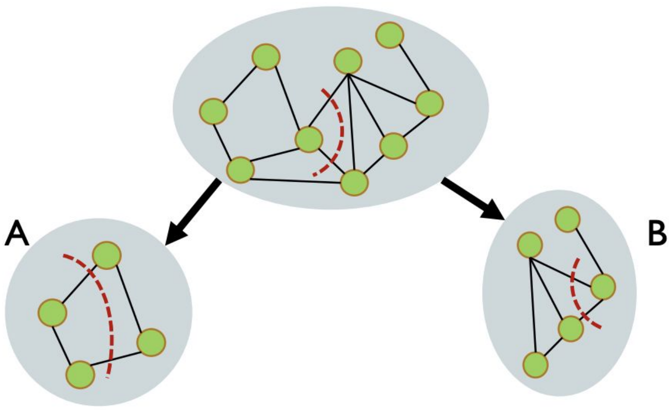
Community structure (web/social/citation): lots of local links ⇒ good recursive cuts

Low-dimensional embeddings (meshes, maps, circuits): known separator theorems



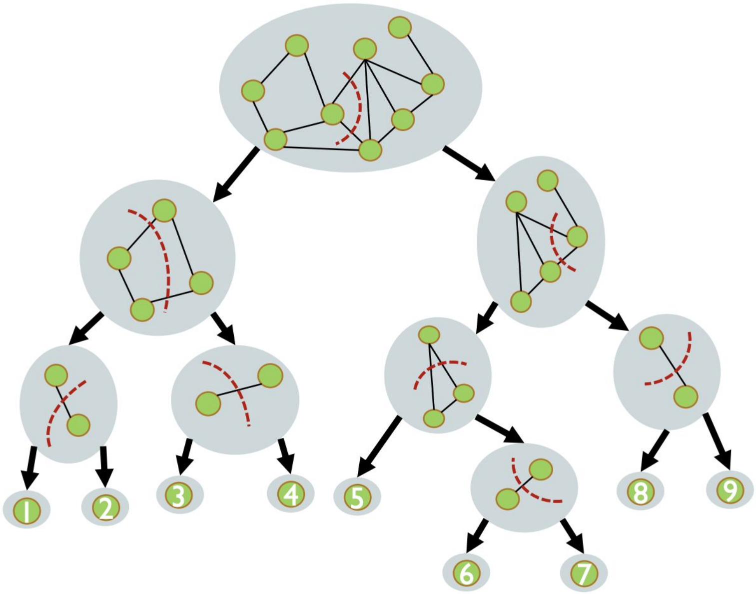
Separator Trees

- Each node contains a subgraph and a separator for that subgraph
- The children of a node contain the two components of the graph induced by the separator
- Priority Metric: w(E_AB) / s(A) s(B)
- Child Flipping Optimization: decide which subgraph should be the left or right child



Compression Algorithm

1. Generate an edge separator tree for the graph
2. Label the vertices in-order across leaves
3. Use an adjacency table to represent the relabeled graph



Why this works?

Previous work:
For graphs satisfying an n^c ($c<1$) separator theorem and using the separator-tree labeling, the adjacency table for any n-vertex member uses $O(n)$ bits, per-edge neighbor access is $O(1)$

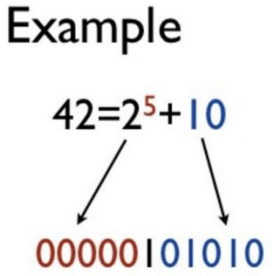
Idea:

- Separator tree gives short ranges for most edges.
- Sum of encoded gaps over all lists is dominated by edges crossing small separators at each level.
- Geometric decrease in subproblem size ⇒ linear total bits.

Gamma Codes

Store an integer d: by using a unary code for ceil(log d) + binary code for d - 2^ceil(log d) = 1 + 2ceil(log d) bits

number	2^n	output
1	2^0+0	1
2	2^1+0	010
3	2^1+1	011
4	2^2+0	00100
5	2^2+1	00101
6	2^2+2	00110
7	2^2+3	00111
8	2^3+0	0001000
9	2^3+1	0001001
10	2^3+2	0001010
11	2^3+3	0001011
12	2^3+4	0001100
13	2^3+5	0001101
14	2^3+6	0001110
15	2^3+7	0001111
16	2^4+0	000010000
17	2^4+1	000010001



K-bit codes

Encode values in chunks of k-bits

Use k-1 bits for data, and 1 bit as the continue bit

Snip, nibble, and byte versions are used

Adjacency Table

For every vertex v, neighbours are sorted into v_1, v_2, …

The associated neighbour list is represented by v_1 - v, v_2 - v_1, …

Metadata:

Sign bit stored in the first entry to deal with negative differences

Start of the list stores the number of entries

These adjacency lists are concatenated to form an adjacency table

Indexing Structure

Each vertex needs a pointer to the start of its adjacency list in the encoded bitstream.

The design balances lookup speed and space efficiency.

Semi-Direct-16 groups 16 consecutive vertices and stores their start offsets in five 32-bit words:

- Word 1: absolute start offset of vertex 0.
- Word 2: three 10-bit deltas from vertex 0 to vertices 4, 8, and 12.
- Words 3 – 5: twelve 8-bit deltas from those reference vertices to the remaining vertices in the group.

This layout cuts index storage by ≈ 6 bits per vertex with negligible impact on lookup time.

Dynamic Representation

Supports incremental insertion and deletion of edges.

Because a vertex's degree can change, its adjacency data must be stored in dynamically allocated memory.

Uses fixed-size memory blocks to simplify allocation and reuse:

- Initially, each vertex owns one block from a pre-allocated array.
- When a block fills, the vertex acquires additional blocks from a shared pool of spare memory blocks.

Blocks belonging to the same vertex are linked together:

- Each block stores an 8-bit nonce i.
- The address of the next block is computed as hash(current_address, i).

To preserve spatial locality, a separate contiguous memory pool is reserved for every 1024 vertices in the graph.

Caching

Repeatedly encoding and decoding neighbor lists is expensive.

When a vertex is accessed, its decoded neighbors are stored in a small LRU-based cache.

Subsequent queries to the same vertex can reuse this uncompressed data directly.

If a cached vertex is modified and later evicted, its updated neighbor list is re-encoded and written back to the main structure in compressed form.

Machine Setup

Two machines, each with 32-bit processors.

A .7GHz Pentium III processor with .1Ghz bus and 1GB of ram (cache line 32 bytes).

2.4Ghz, Pentium 4, with 4 processors, .8GHz bus, and 1GB of ram (cache line 128 bytes). Supports quad vectorization, and hardware prefetching.

Good at contiguous memory accesses

Experiments

Evaluates performance on depth-first search (DFS) while varying edge insertion order.

DFS explores vertices in a non-trivial, data-dependent order.

A simple character array tracks whether each vertex has been visited.

Three edge-insertion strategies are tested:

- Linear: insert all out-edges of vertex 0, then vertex 1, and so on.
- Transpose: insert all in-edges of vertex 0, then vertex 1, etc.
- Random: insert edges in a fully random order.

Static Results

	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

3-4× smaller than arrays (Byte ~12.5 vs Array ~38 b/edge).

Byte is the fastest compressed format; Nibble/Snip trade speed for space.

Ordering drives time for arrays (up to 8× swing); compressed is more stable.

Dynamic Results

	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

Space-opt: ~11.6 b/edge (vs lists ~76.4); Time-opt: ~18.8.

Often faster than lists for DFS, insert is slower.

Insertion/label order can change list performance by 7-11×, dynamic is insensitive.

Block size: larger = faster, smaller = tighter space (pick by average degree).

CPU Performance

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

Table 5: Summary of space and normalized times for various operations on the Pentium 4.

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

Table 6: Summary of space and normalized times for various operations on the Pentium III.

P4: Array(Sep) is fastest; Byte close at ~⅓ the space.

PIII: Byte overtakes arrays for DFS (cache lines/throughput favor compression).

Cached dynamic improves locality-friendly inserts (linear/transpose).

Applications

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

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

PageRank: PIII → Byte fastest; P4 → Array(Sep) fastest, Byte close at 3× smaller space.

Matching: Array(Sep) slightly faster; Byte gives ~2.5× space reduction with minimal slowdown.

Ordering consistently matters for arrays; compression gives a robust baseline

Strengths and Weaknesses

Strengths

The paper presents a clear motivation for using a separator-based representation.

The three-step compression algorithm is intuitive and modular.

Its design supports multiple encoding schemes

The authors evaluate their method on a diverse set of datasets, showing that the representation is compact across many graph types.

Weaknesses

Most additions are experimental.

Evaluation focuses only on DFS, sequential traversal, and edge insertions, broader algorithmic testing would strengthen the claims.

The approach assumes free vertex relabeling; it is less useful when label order is fixed by the application.

The paper lacks NUMA considerations