AN EXPERIMENTAL ANALYSIS OF A COMPACT GRAPH REPRESENTATION

Daniel Blandford, Guy Blelloch, Ian Kash

CMU

Nellie Wu

MOTIVATION

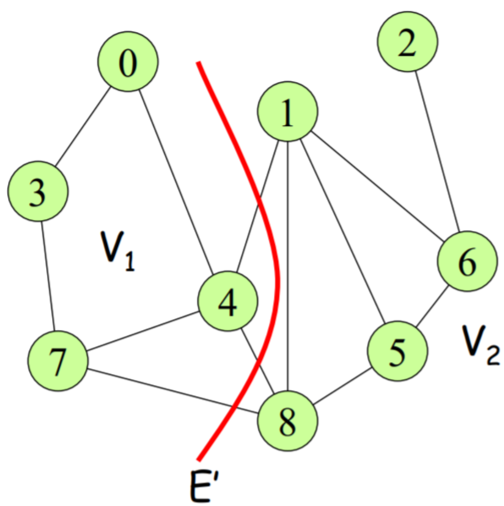
- Graphs are diverse
  - **We want to have a uniform way to represent them**
- Graphs are large and sparse
  - **We want the uniform representation to be compact to save storage**
- Graphs are important data structures to operation on in many algorithms
  - **We want the compact representation to support efficient queries and updates**

This Paper

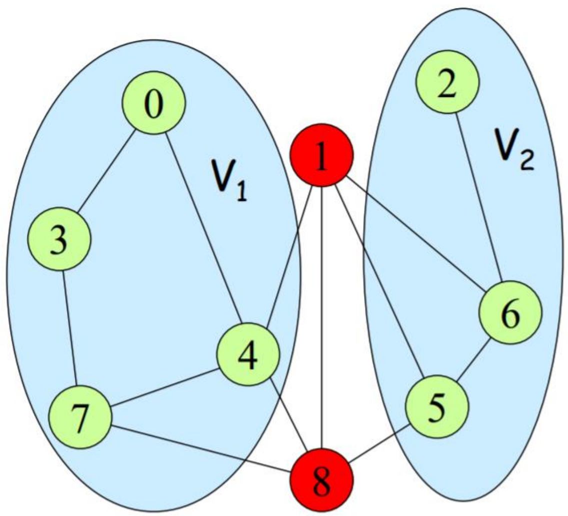
- Proposes the graph separator based representation as a general compact, and efficient representation for various graphs
- Shows the performance of the representation via a comprehensive set of experiments, *e.g., up to 3.5x faster comparing to adjacency arrays for DFS*

GRAPH SEPARATORS

- **Edge separator**
- a set of edges $E' \subset E$ that, when removed, partitions the graph into two almost equal sized parts V_1, V_2.



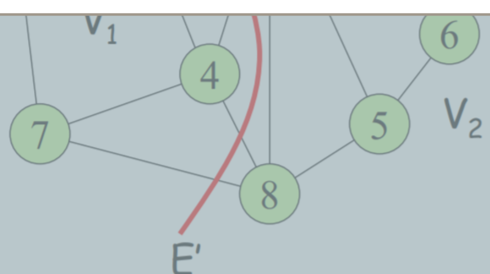
- **Vertex separator**
- a set of vertices $V' \subset V$ that, when removed, partitions the graph into two almost equal sized parts V_1, V_2.



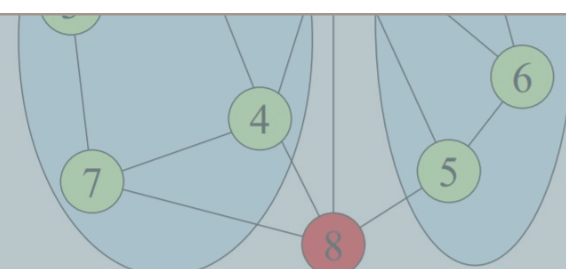
Minimum Separator: the separator that minimizes the number of edges/vertices removed

GRAPH SEPARATORS

- **Edge separator**
- a set of edges $E' \subset E$ that, when removed, partitions the graph into two almost equal sized parts V_1, V_2.



- **Vertex separator**
- a set of vertices $V' \subset V$ that, when removed, partitions the graph into two almost equal sized parts V_1, V_2.



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

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

REAL WORLD GRAPHS HAVE GOOD SEPARATORS

- Good separators allows clean representations of graphs with a set of separators and their associated subtrees
  - Social networks: people form hierarchal communities
  - Scholarly articles: co-authors are usually from similar research areas
  - VLSI circuit design: circuit components usually are laid out in 2D and have just a few metal layers
  - etc.

Thus, separator-based graph representations can lead to compact and efficient graph algorithm processing

ENCODING WITH SEPARATORS

High-Level Compression Algorithm

Generate an edge separator tree for the graph

↓

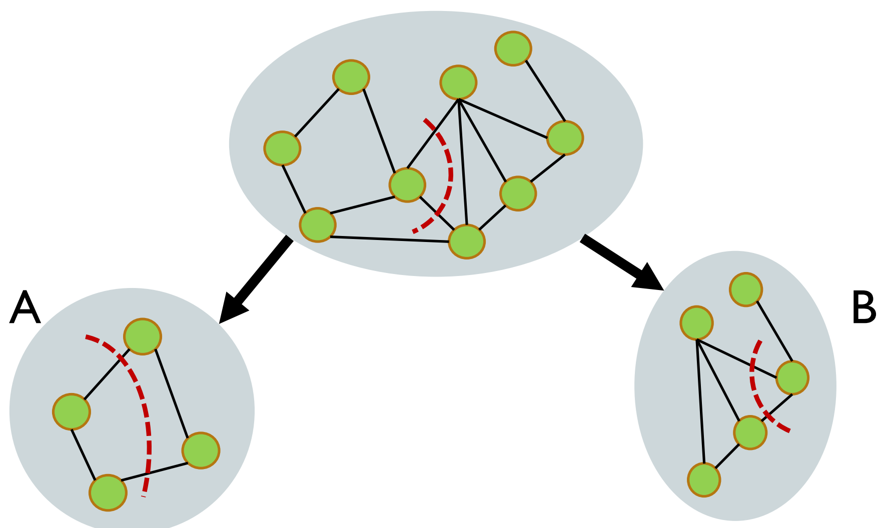
Label the vertices in-order across leaves

↓

Use an adjacency table to represent the relabeled graph

STEP1: BUILD SEPARATOR TREES

Represent Graphs with 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

Heuristic for deciding which edge to collapse

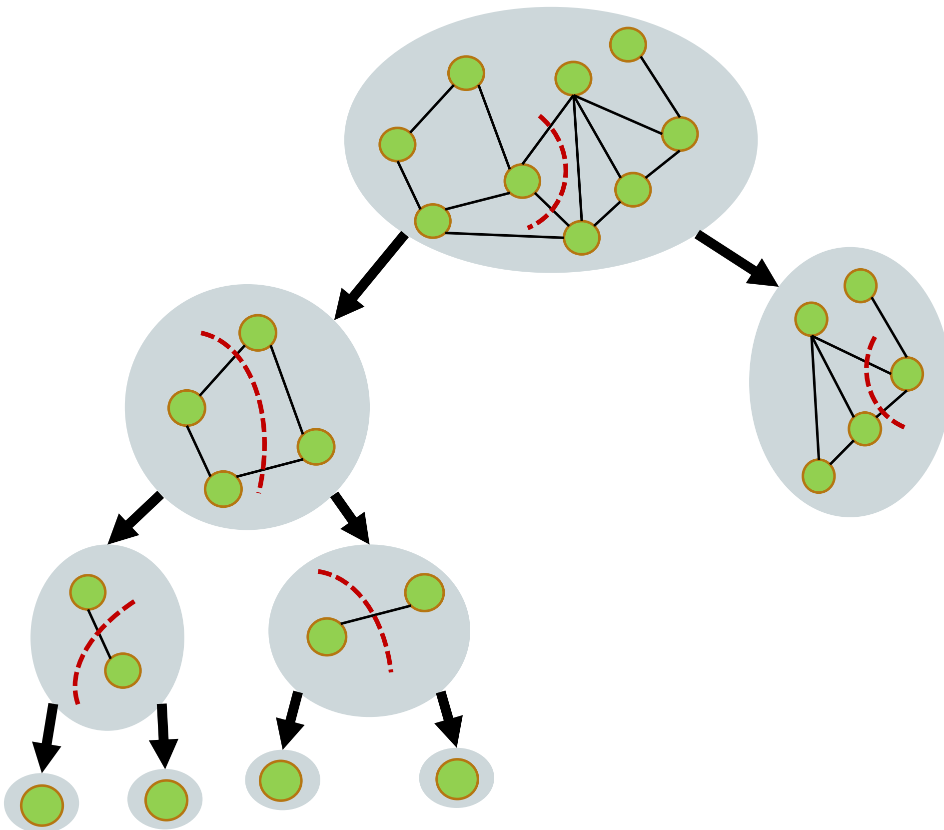
$$priotiy\ metric = \frac{w(E_{AB})}{s(A)s(B)}$$

$w(E_{AB})$ → # of edges between the multivertices A, B

$s(A)s(B)$ → # of vertices in multivertices A, B

STEP1: BUILD SEPARATOR TREES

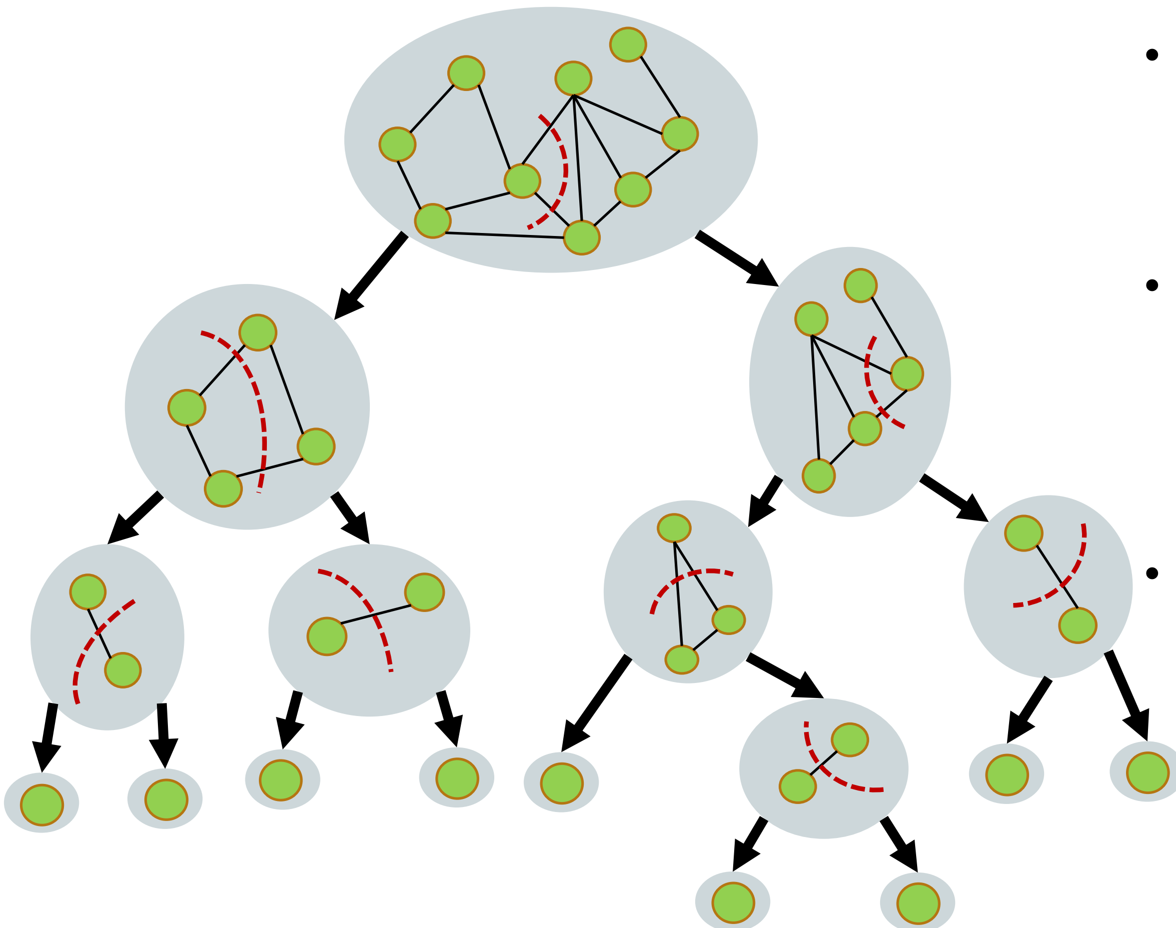
Represent Graphs with Separator Trees



- 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
- Split repeatedly until a single vertex is reached

STEP1: BUILD SEPARATOR TREES

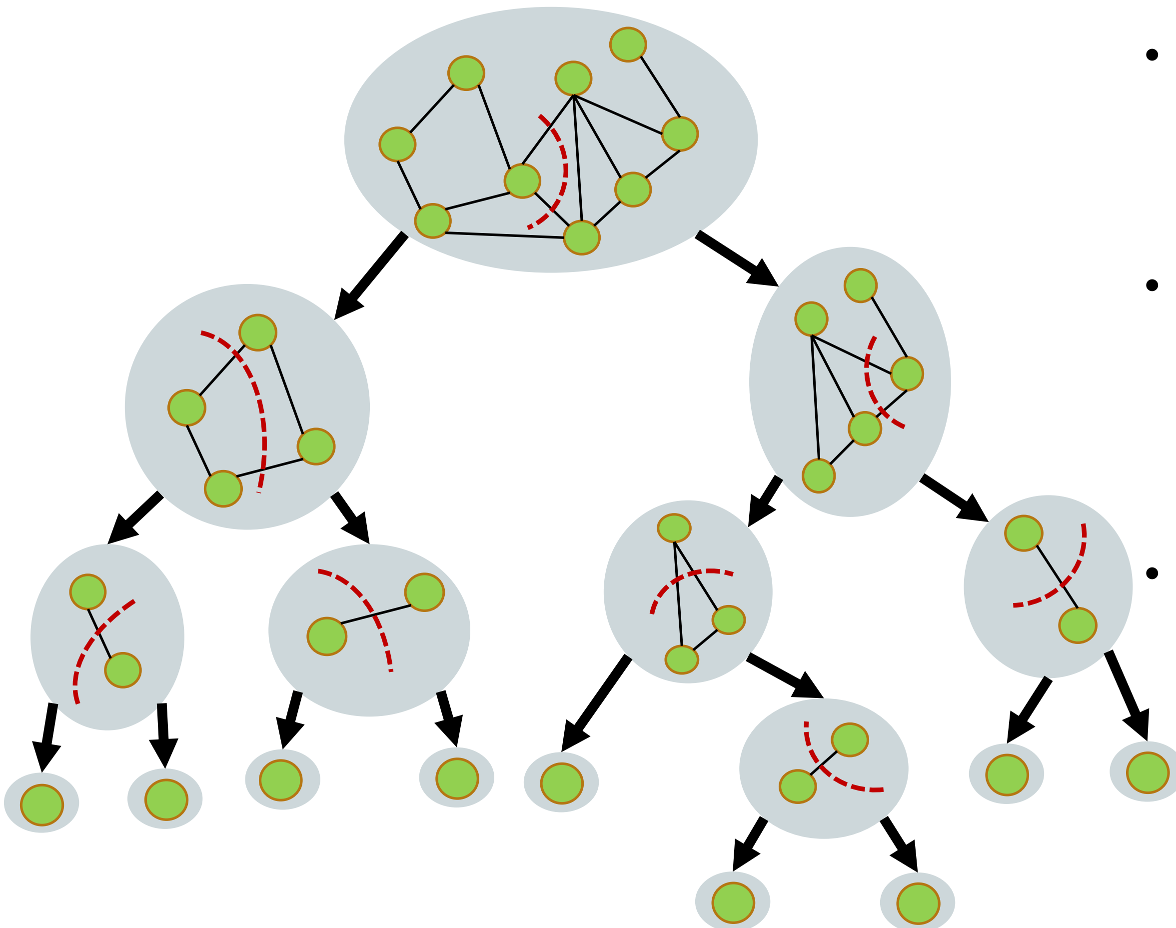
Represent Graphs with Separator Trees



- 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
- Split repeatedly until a single vertex is reached

STEP1: BUILD SEPARATOR TREES

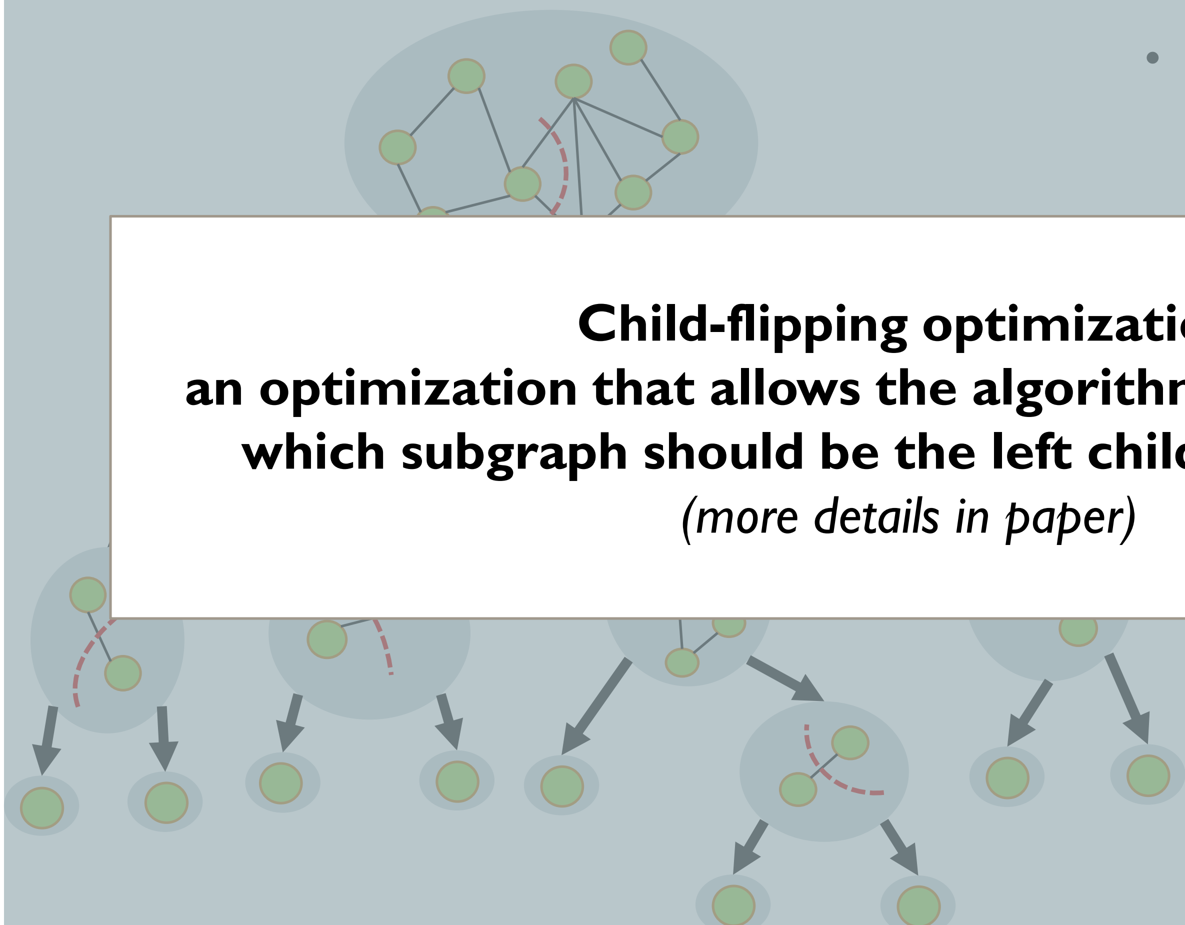
Represent Graphs with Separator Trees



- 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
- Split repeatedly until a single vertex is reached

STEP1: BUILD SEPARATOR TREES

Represent Graphs with Separator Trees



- Each **node** contains a subgraph and a separator for that subgraph

nts

he

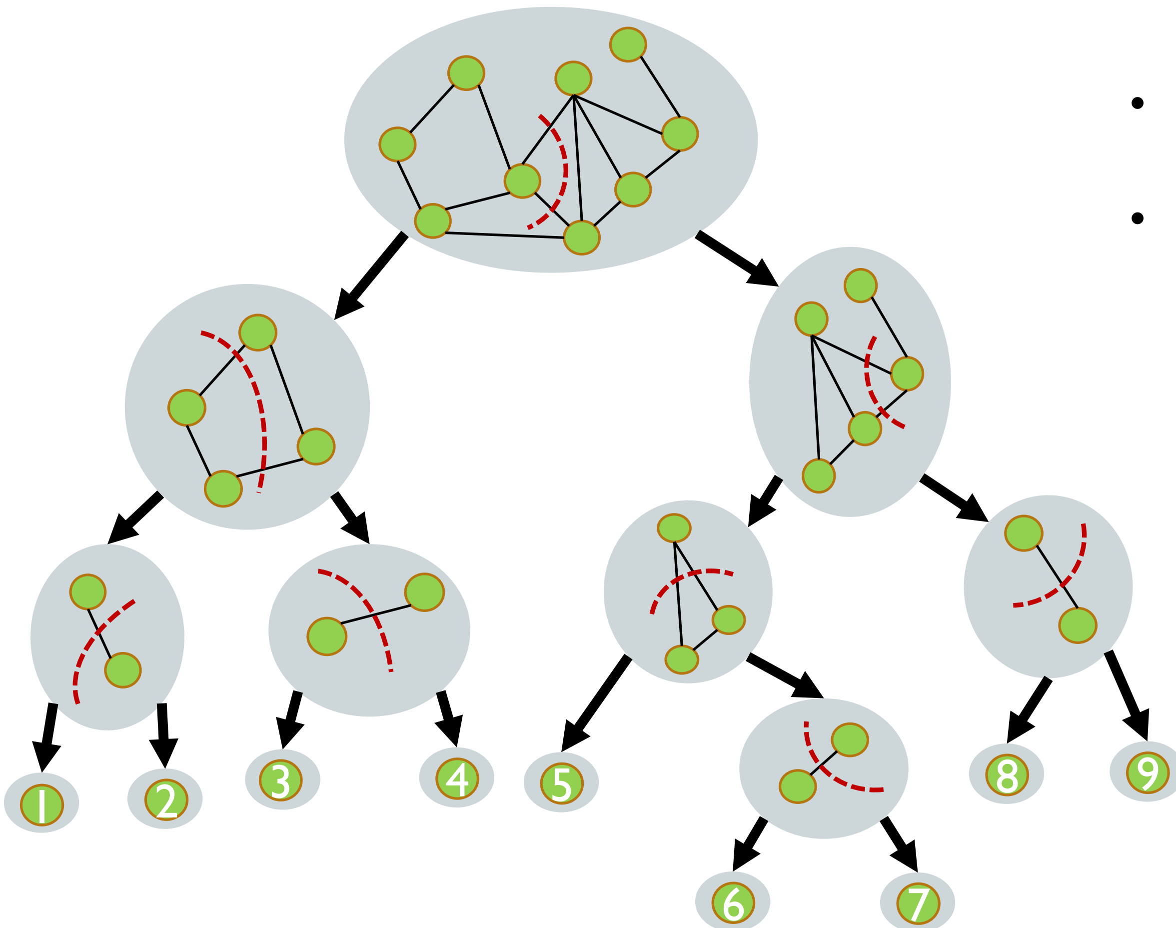
gle

vertex is reached

Child-flipping optimization
an optimization that allows the algorithm to better decide which subgraph should be the left child and right child
(more details in paper)

STEP2: ASSIGN LABEL TO LEAVES

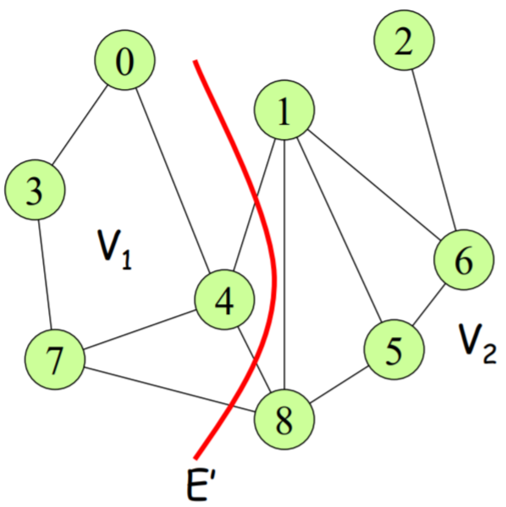
Represent Graphs with Separator Trees



- Assign labels to leaves in an increasing order
- Adjacent labels belong to the same subgraph, allowing more efficient encoding of the representation in the next step

STEP3: CONSTRUCT ADJACENCY TABLES

- For each vertex in the graph, its neighbors are stored in a **difference-encoded adjacency list**.

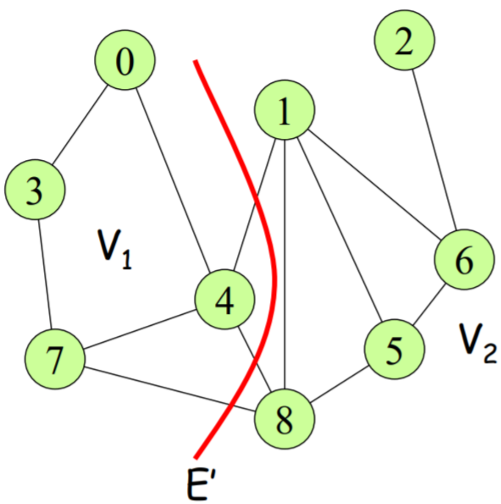


For vertex v, the associated list is: $v_1 - v, v_2 - v_1, \ldots$

example: vertex 0: 3, 1

STEP3: CONSTRUCT ADJACENCY TABLES

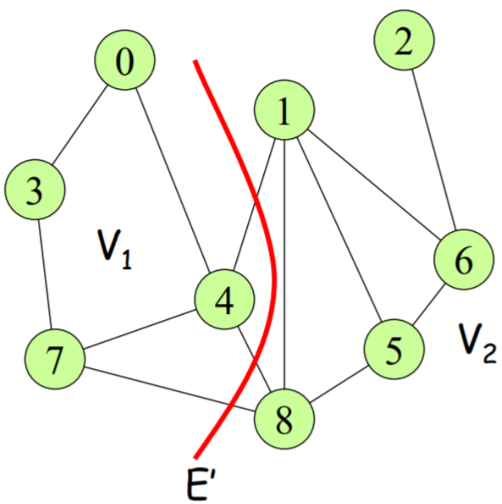
- Difference values are encoded using **logarithmic code**, a prefix code that uses O(logd) bits to encode a difference of size d



Implemented codes:

- gamma code
  - unary code for $\lceil logd \rceil$
  - binary code for $d \ - 2^{\lceil logd \rceil}$
  - total: $1 \ + \ 2\lceil logd \rceil$ bits
- snip, nibble, byte codes
  - 2-, 4-, 8-bit version of the more general k-bit code, which encodes integers as a sequence of k-bit blocks
    - i^{th} bit represents whether the integer is greater than 2^i
  - designed as memory accesses are usually aligned, with fetch with of 2, 4, 8 bits

STEP3: CONSTRUCT ADJACENCY TABLES



For vertex v, the associated list is: $v_1 - v, v_2 - v_1, \ldots$

- Each adjacency list also carries metadata:
  - A signed bit is included in the first entry to account for negative difference
  - The start of the list also stores # of entries in the list
    - Helps with efficiency lookup
- All adjacency lists are concatenated to form the adjacency table for the graph

BOUNDS FOR STORAGE REQUIREMENT

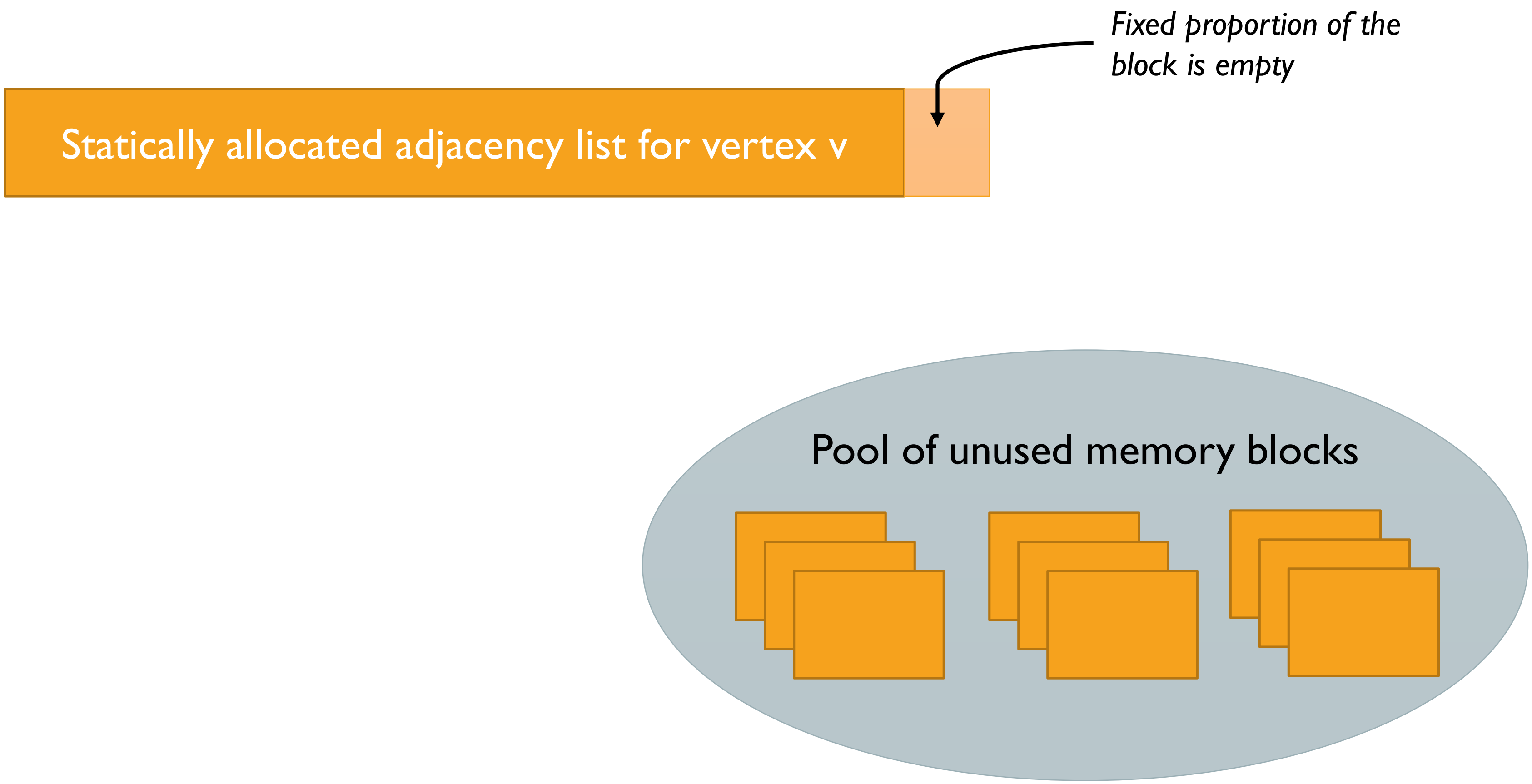
- Lemma (proved in a previous work)

- For a class of graphs satisfying an $n^c (c < 1)$-edge separator theorem, and labeling based on the separator tree satisfying the bounds of separator theorem, the adjacency table for any n-vertex member requires O(n) bits

The adjacency table storage requirement is theoretically bounded

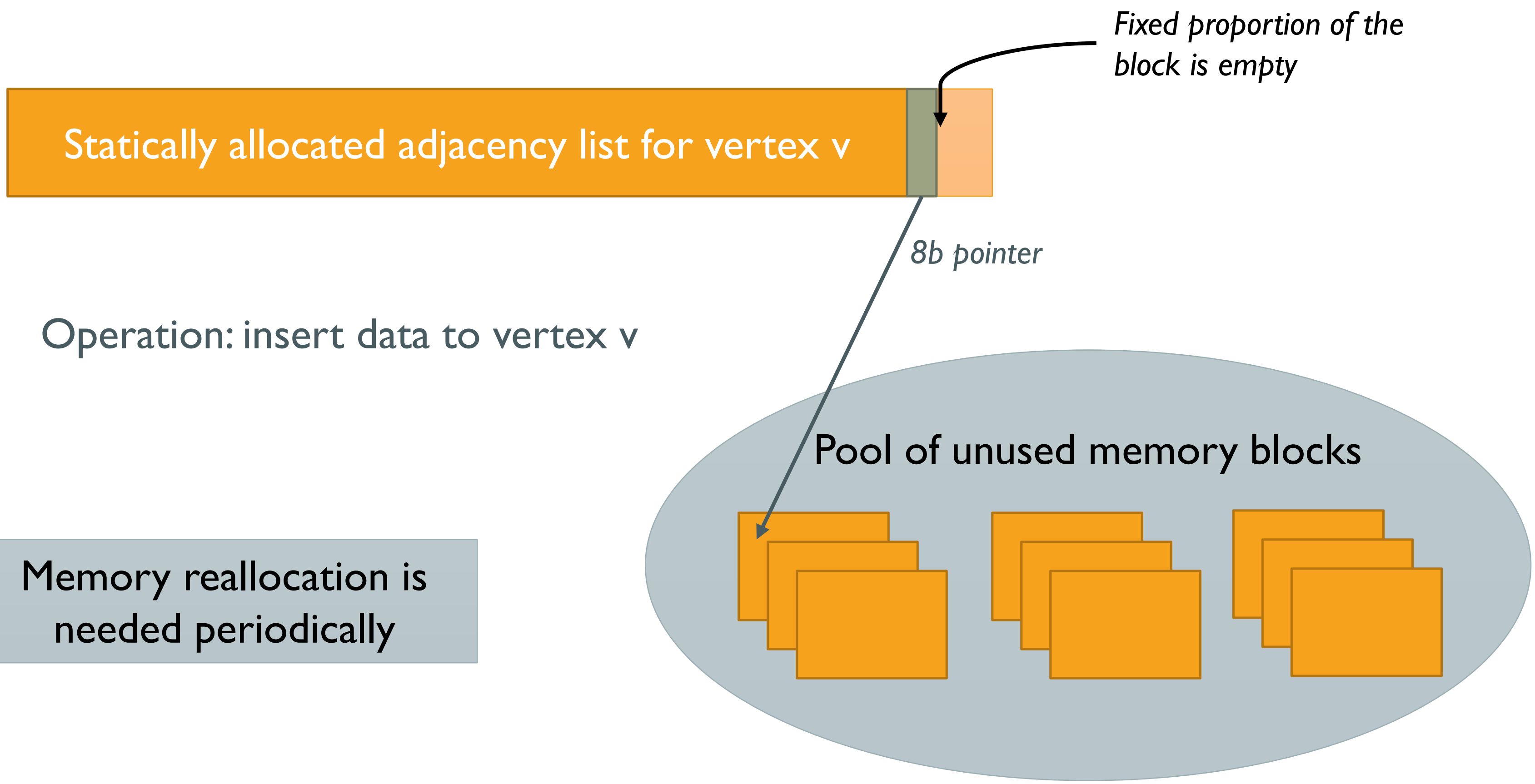
DYNAMIC DATA STRUCTURES

- To allow insertion of new nodes, dynamic allocation of memory (to represent the newly inserted nodes) is necessary



DYNAMIC DATA STRUCTURES

- To allow insertion of new nodes, dynamic allocation of memory (to represent the newly inserted nodes) is necessary



Operation: insert data to vertex v

Memory reallocation is needed periodically

EXPERIMENTAL SETUP

Graphs Used in Experiments

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]

Benchmarks

- DFS time
- time for reading and inserting all edges

COMPARISON TO ADJACENCY ARRAY REPRESENTATION

Machine: Pentium 4 (larger cache line size)

	Array		
	Rand	Sep	
Graph	T_1	T/T_1	Space
auto	0.268s	0.313	34.17
feocean	0.048s	0.312	37.60
m14b	0.103s	0.388	34.05
ibm17	0.095s	0.536	33.33
ibm18	0.113s	0.398	33.52
CA	0.920s	0.126	43.40
PA	0.487s	0.137	43.32
lucent	0.030s	0.266	41.95
scan	0.067s	0.208	43.41
googleI	0.367s	0.226	37.74
googleO	0.363s	0.250	37.74
Avg		0.287	38.202

Rand: vertices are ordered randomly
Seq: vertices are ordered sequentially

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.

COMPARISON TO ADJACENCY ARRAY REPRESENTATION

Machine: Pentium 4 (larger cache line size)

	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.

- Byte encoding is significantly faster than other proposed structures because of the machine's byte-based instruction streams
- Significant space savings compared to baseline
- Always faster than Array-based rand but sometimes slower than array-based seq

BLOCK SIZE SENSITIVITY (DYNAMIC)

- 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

Memory Block Size

	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. .

Storage Space and Processing Time Tradeoff

COMPARISON TO LINKED LIST (DYNAMIC)

- Significant space savings
- Separator-based representations are insensitive to vertex order, so faster than linked list random, but slower than linked list linear

	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 ALGORITHMS RUNNING ON DIFFERENT MACHINES

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.

Machines

- Pentium 3 processor
  - 0.1GHz bus
  - 1GB RAM
  - 32 byte cache line
- Pentium 4 processor
  - 0.8GHz bus
  - 1GB RAM
  - 128 byte cache line

Allows much better performance when the application has spatial locality

SUMMARY

- **Strength**
  - The paper clearly motivates the separator-based representation.
  - The proposed 3-step compression algorithm is easy-to-understand. And the modularity for building adjacency lists based on various encodings to better adapt to the underlying hardware platform allows flexible software-hardware codesign.
  - An extensive set of datasets are used in the evaluation section to show that the representation is indeed compact for various classes of graphs.
- **Weakness**
  - The work is pretty incremental, as it is mostly based on a previously proposed separator-based representation. Most of the new work is just related to run more experiments.
  - The experiment only considers DFS, sequential traversal, and insertion. It would be more convincing if more algorithms are evaluated.
  - The table-based result presentation is really hard to read and find insights.
  - The representation is only useful if the algorithm allows free labeling of vertices.