Graph Compression

Data compression: lecture 5

15-853, Spring 2018
Outline

• What is a graph?
• Graph representations
• Compressing and reordering graphs
• Examples
What is a graph?

• $G(V, E)$, usually $n$ for #vertices, $m$ for #edges
• Vertices model “objects”
• Edges model relationship between objects
What is a graph?

- Edges can be undirected or directed

**Undirected**
- Coauthorship network
- Social networks (Facebook)
- Protein-protein interaction

**Directed**
- Hyperlink graphs
- Email graphs (enron)
- Follower graphs (twitter)
## Graph sizes in 2018

<table>
<thead>
<tr>
<th>Graph</th>
<th>IVI</th>
<th>IEI (symmetrized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>com-Orkut</td>
<td>3M</td>
<td>234M</td>
</tr>
<tr>
<td>Twitter</td>
<td>41M</td>
<td>1.46B</td>
</tr>
<tr>
<td>Friendster</td>
<td>124M</td>
<td>3.61B</td>
</tr>
<tr>
<td>Hyperlink2012-Host</td>
<td>101M</td>
<td>2.04B</td>
</tr>
<tr>
<td>Facebook (2011) [1]</td>
<td>721M</td>
<td>68.4B</td>
</tr>
<tr>
<td>Hyperlink2012 [2]</td>
<td>3.5B</td>
<td>225B</td>
</tr>
<tr>
<td>Facebook (2018)</td>
<td>&gt; 2B</td>
<td>&gt; 300B</td>
</tr>
<tr>
<td>Google (2018)</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

- Green circle: Publicly available graphs
- Red circle: Private graph datasets

[1] The Anatomy of the Facebook Social Graph, Ugander et al. 2011
Graph compression in industry

NetflixGraph Metadata Library: An Optimization Case Study
by Drew Koszewnik

**Problem:** running into memory issues when storing the movie property graph in memory

**Solution:** Compact Encoded Data Representation

We knew that we could hold the same data in a more memory-efficient way. We created a library to represent directed-graph data, which we could then overlay with the specific schema we needed.

**Results**

When we dropped this new data structure in the existing NetflixGraph library, our memory footprint was reduced by 90%. A histogram of our test application from above, loading the exact same set of data, now looks like the following:

Source: Netflix Tech Blog
Graph compression in industry

Compressing Graphs and Indexes with Recursive Graph Bisection

Abstract

Graph reordering is a powerful technique to increase the locality of the representations of graphs, which can be helpful in several applications. We study how the technique can be used to improve compression of graphs and inverted indexes.

Our experiments show a significant improvement of the compression rate of graph and indexes over existing heuristics. The new method is relatively simple and allows efficient parallel and distributed implementations, which is demonstrated on graphs with billions of vertices and hundreds of billions of edges.
Operations on graphs

- Static graphs:
  - scanning the whole graph (i.e. the storage cost)
  - get_neighbors(v) (in/out neighbors for digraphs)
  - is_edge(u, v) (is the (u, v) edge present in G?)
- Dynamic graphs:
  - insert/delete edges

Source: MIT-6.172 Lecture 21
Graph representations

**Adjacency Matrix**

- Vertices labeled from 0 to n-1
- Entry of "1" if edge exists, 0 o.w.

```
   0 0 0 0
   1 0 1 1
   0 0 0 1
   0 1 1 0
```

**Edge List**

- (1, 0)
- (1, 2)
- (1, 3)
- (2, 3)
- (3, 1)
- (3, 2)

- Space requirements in terms of m and n?
Graph representations

**Adjacency List**

- Array of pointers (one per vertex)
- Each vertex points to a list of its neighbors
- Linked lists: bad cache performance, use arrays instead
  - Tradeoff: hard to insert/delete edges
Graph representations

Compressed Sparse Row (Column)
- Cache-friendly method of storing graph in memory
- Two arrays: Offsets and Edges
- Offsets[i] stores the offset where vertex i’s edges start in Edges

Offsets:

| 0 | 2 | 5 | 5 | 10 | … |

Edges:

| 2 | 13 | 0 | 2 | 3 | 100 | 101 | 102 | 155 | 156 | … |

- How do we calculate the degree of a vertex?
- Space usage?
- Jargon: CSR used for out-edges, CSC for in-edges
## Graph representations: costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>CSR/CSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_graph</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>get_neighbors</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_edge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ins/del neighbor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: [MIT-6.172 Lecture 21](http://example.com)
Graph representations: costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>CSR/CSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_graph</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
<td>$O(m + n)$</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>get_neighbors</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_edge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ins/del neighbor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: MIT-6.172 Lecture 21
## Graph representations: costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>CSR/CSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_graph</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
<td>$O(m + n)$</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>get_neighbors</td>
<td>$O(n)$</td>
<td>$O(m)$</td>
<td>$O(d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>is_edge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ins/del neighbor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: MIT-6.172 Lecture 21
# Graph representations: costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>CSR/CSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_graph</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
<td>$O(m + n)$</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>get_neighbors</td>
<td>$O(n)$</td>
<td>$O(m)$</td>
<td>$O(d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>is_edge</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
<td>$O(d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>ins/del neighbor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Graph representations: costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>CSR/CSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_graph</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
<td>$O(m+n)$</td>
<td>$O(m+n)$</td>
</tr>
<tr>
<td>get_neighbors</td>
<td>$O(n)$</td>
<td>$O(m)$</td>
<td>$O(d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>is_edge</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
<td>$O(d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert edge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete edge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: [MIT-6.172 Lecture 21](https://mit.edu)
## Graph representations: costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>CSR/CSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_graph</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
<td>$O(m + n)$</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>get_neighbors</td>
<td>$O(n)$</td>
<td>$O(m)$</td>
<td>$O(d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>is_edge</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
<td>$O(d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert edge</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$ or $O(d)$</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>delete edge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Adjacency Matrix</td>
<td>Edge List</td>
<td>Adjacency List</td>
<td>CSR/CSC</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------</td>
<td>----------------</td>
<td>----------------</td>
<td>---------------</td>
</tr>
<tr>
<td>scan_graph</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
<td>$O(m + n)$</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>get_neighbors</td>
<td>$O(n)$</td>
<td>$O(m)$</td>
<td>$O(d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>is_edge</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
<td>$O(d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>insert edge</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$ or $O(d)$</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>delete edge</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
<td>$O(d)$</td>
<td>$O(m + n)$</td>
</tr>
</tbody>
</table>

Source: MIT-6.172 Lecture 21
Graph representations: summary

- Understand the set of operations before choosing a format
- This lecture: mostly use CSR/CSC
  - Sparse graphs \( m = O(n) \)
  - Static algorithms
  - Need to scan over neighbors of a vertex efficiently

\[ \begin{array}{cccccc}
\text{Offsets} & 0 & 2 & 5 & 5 & 10 \\
\text{Edges} & 2 & 13 & 0 & 2 & 3 & 100 & 101 & 102 & 155 & 156
\end{array} \]

\( n + m \) space

Source: MIT-6.172 Lecture 21
Storing uncompressed graphs

Hyperlink2012 Graph
- $n = 3.6B$, $m = 225B$ (undirected edges)
- Vertex ids fit into 4 bytes
- $> 900Gb$ to store in CSR format

32Gb DRAM: about 300$*

So, about 9000$ of memory just to store the graph. Doesn’t include memory needed to run algorithms on it!

*Source: Hynix HMA84GR7MFR4N-UH 32GB DDR4-2400 ECC REG DIMM Server Memory
Compressing graphs

• Web graphs
• Difference encoding
• Reordering for locality
Web graphs

• Vertices are web pages
• Directed edges represent hyperlinks
• Used:
  • Understand structure of the web
  • Mine communities
  • Prioritize crawling

Entire conferences around the web and web-algorithms
Compressing web graphs

Is the web structured?
Compressing web graphs

Lots of structure!

- **Locality**: Many links stay within the same sub-domain. I.e. most links point close by in the lexicographic ordering.

- **Similarity**: Pages close by in the lexicographic order tend to have similar sets of neighbors.

- Boldi and Vigna (WWW 2004) exploit these observations about the internet in the WebGraph framework:
  - Reference coding
  - Difference coding

Source: Boldi and Vigna, “The WebGraph Framework I: Compression Techniques”, WWW 2004
Compressing web graphs: techniques

Reference coding

Idea: to encode neighbors of $v$
- Find previous vertex, $ref$, which has significant overlap
- Encode edges with respect to $ref$.

Original graph:

vertex 0: [1, 2, 4, 5, 9, 10]

... vertex 6: [1, 2, 4, 5, 9, 13]

Reference coded:

vertex 0: [1, 2, 4, 5, 9, 10]

... vertex 6: ref(0), {10}, {13}

How do you find good references?

Is accessing $N(v)$ efficient? (O(deg(v))?)

Source: Boldi and Vigna, “The WebGraph Framework I: Compression Techniques”, WWW 2004
Compressing web graphs: techniques

**Difference coding**

Neighbor lists exhibit a high degree of *locality*

We want to store a set of integer vertex ids

\[ N(3) = [2, 4, 1, 13, 5, 9] \]

Sort the elements

\[ N(3) = [1, 2, 4, 5, 9, 13] \]

Store gaps instead of the actual integers

\[ [1-3, 2-1, 4-2, 5-4, 9-5, 13-9] \]

\[ = [(-)2, 1, 2, 1, 4, 4] \]

Compress the gaps using *integer codes*

Source: Boldi and Vigna, “The WebGraph Framework I: Compression Techniques”, WWW 2004
Compressing web graphs

WebGraph Framework

Combines:
• Reference coding
• Difference encoding

The WWW paper shows that these two techniques can be used to represent a billion-edge web graph in
• out edges: 3.08 bits/edge
• in edges: 2.89 bits/edge

Why do in edges compress better?

Source: Boldi and Vigna, “The WebGraph Framework I: Compression Techniques”, WWW 2004
Compressing web graphs

Many other systems and techniques:

Fast and Compact Web Graph Representations
  • Grammar-based techniques (Re-Pair, LZ)
  • Similar space as WebGraph, but faster access times

Representing Web Graphs
  • Hierarchical representation of web graphs

Towards Compressing Web Graphs
  • Another early scheme based on copying
Compressing CSR

Use difference coding

- General purpose technique
- Compresses lists with small, regular gaps well
- Easy to see that accessing neighbors is $O(deg(v))$

Offsets

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>5</th>
<th>5</th>
<th>10</th>
<th>...</th>
</tr>
</thead>
</table>

Edges

| 2 | 13 | 0 | 2 | 3 | 100 | 101 | 102 | 155 | 156 | ... |

Compressed edges

| 2 | 11 | -1 | 2 | 1 | 97 | 1 | 1 | 53 | 1 | ... |

Source: Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+
Variable length codes

k-bit codes

• Most gaps are small; want to avoid wasting 4 bytes/gap

Gaps:

| 8 | 1 | 1 | 53 | 1 |

• Ex: byte-code.

encoding(7):

| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

continue bit

encoding(129):

| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

= \(2^0\)*(block1) + \(2^7\)*(block2)

Source: Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+
Variable length codes

**k-bit codes**

- First gap could be negative, so first block is encoded specially (6 data bits, 1 continue bit, 1 sign-bit)
- Decoding:

```c
int read_byte_code(uint8_t* start) {
    int gap = 0;
    int shift = 0;
    while (1) {
        uint8_t b = *start++;
        gap += ((b & 0x7f) << shift);
        if (LAST_BIT_SET(b))
            shift += 7;
        else
            break;
    }
    return gap;
}
```

- Any issues with byte-codes? What if gaps are really small?

Source: Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+
Variable length codes

4-bit codes (nibbles)

- Same ideas work, encode data in blocks of k-1 bits
- Decoding cost grows in practice, more branches

What is a 1-bit code?

- Recall gamma codes: to encode a number x
  - store T, the largest power of two < x in unary
  - store a “0” (delimiter)
  - store x % T

1-bit code is effectively a gamma code

Source: Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+
Variable length codes

Run-length encoded byte-codes

- Branches are costly in practice

#bytes/gap 1 1 1 1 1 1 2 1 1 1 1 1

run length encode

1-byte header 0 1 0 0 0 0 1 1 data blocks

#bytes/gap #blocks (max value is 2^6) data blocks

- Increases space, but decoding is cheaper (less branches)

Source: MIT-6.172 Lecture 21, Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+
Reordering graphs for locality

What is locality?

Adjacency matrix plots:

- initial order: looks bad
- slightly better order
- another order; is this better?
- lots of empty cells, is this even better?

Source: Compressing Graphs and Indexes with Recursive Graph Bisection
Reordering graphs for locality

What is locality?

I don’t really know how to define it. Maybe you know.

Here’s one idea:
• Measure the number of bits needed to difference encode all adjacency lists, and just call this locality
• Measure is known in literature as “log-gap cost”
Reordering graphs for locality

Log-gap cost

Fix some order, $\pi$

Cost of an adjlist, $f_\pi(v, out(v)) = \sum_{i=1}^{\deg(v)-1} \log |\pi(v_{i+1}) - \pi(v_i)|$

Problem: find $\pi$ minimizing $\sum_{v \in V} f_\pi(v, out(v))$

This problem is NP-hard

Finding the best ordering for difference-coding is hard

Related to Minimum Linear Arrangement (MLA), which comes up in VLSI design

$$\min_{\pi} \sum_{(u,v) \in E} |\pi(u) - \pi(v)|$$

Source: Compressing Graphs and Indexes with Recursive Graph Bisection
Reordering graphs for locality

Shingling

Originally purpose: detecting duplicate documents
- Compute a “fingerprint” of a vertex
- Order vertices that have similar fingerprints together

Jaccard coefficient: \( J(A, B) = \frac{|A \cap B|}{|A \cup B|} \)

Pick a hash function \( f \) (see [2] for necessary properties)

\[
M_f(A) = \arg \min_{a \in A} (f(a))
\]

\[
P[M_f(A) = M_f(B)] = \frac{|A \cap B|}{|A \cup B|} = J(A, B)
\]

Vertices with the same shingle likely to have high Jaccard similarity

[1] On Compressing Social Networks
[2] Identifying and Filtering Near-Duplicate Documents
Reordering graphs for locality

Recursive bisection

Source: Compressing Graphs and Indexes with Recursive Graph Bisection
Reordering graphs for locality

Recursive bisection

Source: Compressing Graphs and Indexes with Recursive Graph Bisection
Reordering graphs for locality

Recursive bisection

Source: Compressing Graphs and Indexes with Recursive Graph Bisection
Reordering graphs for locality

**Algorithm: Bisection**

- Initialize bisection randomly
- While not converged
  - swap two vertices that improve the optimization goal

*Kernighan-Lin Heuristic*

Source: *Compressing Graphs and Indexes with Recursive Graph Bisection*
Reordering graphs for locality

Experimental results

<table>
<thead>
<tr>
<th>Graph</th>
<th>Algorithm</th>
<th>LogGap</th>
<th>Log</th>
<th>BV</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td>Natural</td>
<td>10.43</td>
<td>17.44</td>
<td>14.61</td>
</tr>
<tr>
<td></td>
<td>BFS</td>
<td>10.52</td>
<td>17.59</td>
<td>14.69</td>
</tr>
<tr>
<td></td>
<td>Minhash</td>
<td>10.79</td>
<td>17.76</td>
<td>15.07</td>
</tr>
<tr>
<td></td>
<td>LLP</td>
<td>7.46</td>
<td><strong>12.25</strong></td>
<td>11.12</td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td><strong>7.03</strong></td>
<td>12.79</td>
<td><strong>10.73</strong></td>
</tr>
<tr>
<td>Twitter</td>
<td>Natural</td>
<td>15.23</td>
<td>23.65</td>
<td>21.56</td>
</tr>
<tr>
<td></td>
<td>BFS</td>
<td>12.87</td>
<td>22.69</td>
<td>17.99</td>
</tr>
<tr>
<td></td>
<td>Minhash</td>
<td>10.43</td>
<td>21.98</td>
<td>14.76</td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td><strong>7.91</strong></td>
<td><strong>20.50</strong></td>
<td><strong>11.62</strong></td>
</tr>
<tr>
<td>FB-NewOrlean</td>
<td>Natural</td>
<td>9.74</td>
<td>14.29</td>
<td>14.64</td>
</tr>
<tr>
<td></td>
<td>BFS</td>
<td>7.16</td>
<td>12.63</td>
<td>10.79</td>
</tr>
<tr>
<td></td>
<td>Minhash</td>
<td>7.06</td>
<td>12.57</td>
<td>10.62</td>
</tr>
<tr>
<td></td>
<td>TSP</td>
<td>5.62</td>
<td>11.61</td>
<td>8.96</td>
</tr>
<tr>
<td></td>
<td>LLP</td>
<td>5.37</td>
<td><strong>9.41</strong></td>
<td>8.54</td>
</tr>
<tr>
<td></td>
<td>Spectral</td>
<td>7.64</td>
<td>11.49</td>
<td>11.79</td>
</tr>
<tr>
<td></td>
<td>Multiscale</td>
<td>5.90</td>
<td>9.58</td>
<td>9.25</td>
</tr>
<tr>
<td></td>
<td>SlashBurn</td>
<td>8.37</td>
<td>13.06</td>
<td>12.65</td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td><strong>4.99</strong></td>
<td>9.45</td>
<td><strong>8.16</strong></td>
</tr>
<tr>
<td>FB-1B</td>
<td>Natural</td>
<td>19.63</td>
<td>27.22</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Minhash</td>
<td>14.60</td>
<td>26.89</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td><strong>8.66</strong></td>
<td><strong>18.36</strong></td>
<td></td>
</tr>
</tbody>
</table>

Edges on facebook: ~8bits/edge

Source: Compressing Graphs and Indexes with Recursive Graph Bisection
Highly compressible graph families

- Succinct data-structure: uses space that is “close” to information theoretic lower bound
  
  $$\text{lower bound: } \Omega(Z) \quad \text{succinct: } Z + o(Z)$$

- Classic results: planar graphs can be represented in $O(n)$ bits
  - Similar results for constant genus graphs

- Graphs that admit an $O(n^c)$-separator theorem in $O(n)$ bits

Source: Compact Representations of Separable Graphs
Conclusion: challenges in graph algorithms

- Compressed representations important (memory isn’t free)
- Efficient representations important
  - Tradeoffs between space-efficiency and fast decoding
  - Formats should be amenable to parallelization

Real world graphs are highly compressible!

- Web graphs in a few bits/edge
- Social networks: no simple (lex) order with high locality
- Special graph families are highly compressible
- RW-graphs have much smaller separators than expected*

*Source: Compact Representations of Separable Graphs