Graph Compression

Toward a Generalized Algorithm

Daniel Szelogowski

UW - Whitewater
Computer Science Master's Program
Whitewater, WI
szelogowdj19@uww.edu

Abstract. *Currently, most graph compression algorithms focus on in-memory compression (such as for web graphs) – few are feasible for external compression, and there is no generalized approach to either task. These compressed representations are versatile and can be applied to a great number of different applications, with the most common being social network and search systems. We present a new set of compression approaches, both lossless and lossy, for external memory graph compression. These new algorithms may also be applicable for runtime usage (i.e., running graph algorithms on the compressed representation).*

Keywords- Graph Compression; LZW; Lossy Compression; Lossless Compression; Sparse Adjacency Matrix; Run-Length Encoding

1. INTRODUCTION

Graph compression typically refers to the compression of graph structures for massive computations. These algorithms are frequently used by companies such as Netflix, social networks, and web maps, as well as search engines (i.e., the PageRank algorithm). They are used to compress graphs primarily in-memory, but often in external memory as well. In this case, our goal is not necessarily lossless compression, but reducing the amount of space needed to store the most important features of the graph for faster computations (e.g., feature selection and compressed representations).

2. RELATED WORK

The most common approaches include the following (primarily in-memory) algorithms:

- Reduction using trees (Binary tree (usually computed using the Minimum Spanning Trees of the Breadth-First Search of the graph), LZ(77)/LZW tree) [3]
- Approximate representation and greedy algorithms [6, 5]
- Integer optimization for approximation [4]
- Gzip adjacency matrix [2]

As well, there is a framework known as "Partition and Code" that uses a novel graph encoding scheme and converts the graph into a dictionary [1].

3. APPROACH

Our approach focuses on external memory compression (of unweighted graphs), rather than compressed in-memory representation – though our methodology uses the related work as the basis for the algorithm designs. All graphs were converted to the "GraphML" format (see **Figure 1**) to unify the datasets – any IDs that were not already integers were converted to their hash code to reduce complexity. We focused on three approaches, two based on the adjacency matrix representation of the graph, and the third being the compression of the **graphml** file directly – all of which can easily be decompressed to reconstruct the original graph. The final system was implemented in Java [11].

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <graph id="G" edgedefault="undirected">
    <node id="n0"/>
    <node id="n1"/>
    <node id="n2"/>
    <node id="n3"/>
    <node id="n4"/>
    <node id="n5"/>
    <node id="n6"/>
    <node id="n7"/>
    <node id="n8"/>
    <node id="n9"/>
    <node id="n10"/>
    <edge source="n0" target="n2"/>
    <edge source="n1" target="n2"/>
    <edge source="n2" target="n3"/>
    <edge source="n3" target="n5"/>
    <edge source="n3" target="n4"/>
    <edge source="n4" target="n6"/>
    <edge source="n6" target="n5"/>
    <edge source="n5" target="n7"/>
    <edge source="n6" target="n8"/>
    <edge source="n8" target="n7"/>
    <edge source="n8" target="n9"/>
    <edge source="n8" target="n10"/>
  </graph>
</graphml>
```

Figure 1: An example GraphML file [9]

4. IMPLEMENTATION

We present three different algorithms for comparison:

Sparse Matrix – convert the graph into an adjacency matrix, then return the sparse format (comprised of two arrays, one holding the running sum of each row's number of 1's + the previous sum, and the other holding the index of each 1 per row); a hash table was used as the dictionary for the vertex IDs to reduce the space necessary for storing the adjacency matrix. Finally, Huffman encode the sparse matrix and dictionary string and write out the new files.

RLE Adjacency Matrix – convert the graph into an adjacency matrix, then Run-Length Encode (RLE) the multi-line binary string representation of the adjacency matrix (using "f" for 0 and "t" for 1), then finally Huffman encode the RLE matrix and dictionary string and write out the new files; it is worth noting that we also attempted Move-to-Front Encoding before RLE, but this greatly increased runtime and often overwhelmed the heap space.

LZW – read in the graphml file as a string (or convert an adjacency list/matrix to graphml string), compress using the LZW algorithm (an extension of LZ78 that uses a pre-initialized dictionary with all possible characters/symbols), encode using Huffman, then write out the new files.

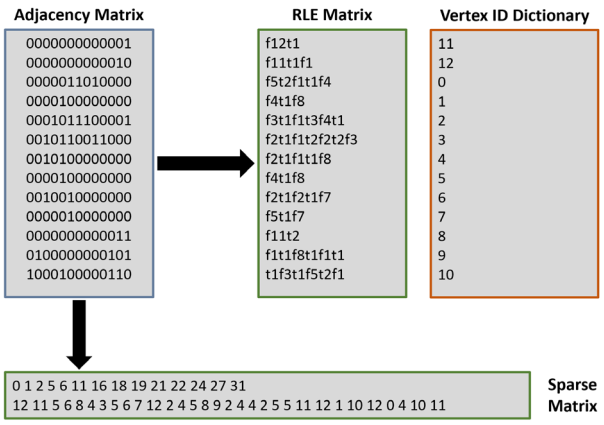

Figure 2: Adjacency Matrix Representations

5. EVALUATION

The algorithms were evaluated on four different graph datasets:

- Tiny (a small, 13 vertex graph with 17 edges)
- Nashville Meetup Network [8]
- Manhattan Street Network [7]
- New York City Street Network [7]

An additional dataset (Spotify Playlists [10]) was also used for testing, but due to the large size of the dataset, it was infeasible for comparison because most algorithms would run out of heap space before completion. Each algorithm's **average compression and decompression times** were reported, along with the **final compression size for each file**.

6. RESULTS

Overall, the Sparse Matrix algorithm provided the best compression and decompression time.

	Sparse Matrix	LZW	RLE Adjacency Matrix
Compression Time	**3179.587 ms**	8025.988 ms	18792.012 ms
Decompression Time	**910.305 ms**	13814.059 ms	27478.092 ms

Table 1: Average Performance

For datasets that were already small, the RLE Adjacency Matrix tended to produce the smallest file (or with a negligible difference to the Sparse Matrix), but massive graphs always performed better with the Sparse Matrix algorithm.

	Tiny	Manhattan	Nashville_Meetup	New York City
Uncompressed	1.12 KB	4.40 MB	55.5 MB	54.8 MB
Sparse Matrix	242 B	**44.7 KB**	2.52 MB	**698 KB**
LZW	1.18 KB	2.62 MB	31.8 MB	32.2 MB
RLE Adjacency Matrix	**205 B**	53.3 KB	**2.19 MB**	807 KB

Table 2: Compression Rates

7. CONCLUSION

Overall, the Sparse Matrix algorithm appears to be the most effective "generalized" approach, though the RLE Adjacency Matrix is comparable – a hybrid approach based on the size of the dataset may potentially perform even better. However, should one prefer lossless compression, LZW is likely sufficient. Currently, the adjacency matrices are unweighted, though weighted edges could easily be added, as well as potentially storing another dictionary (or multiple) containing additional vertex data.

REFERENCES

[1] Giorgos Bouritsas, Andreas Loukas, Nikolaos Karalias, and Michael M. Bronstein. 2021. Partition and Code: learning how to compress graphs. In Advances in Neural Information Processing Systems. Retrieved from http://openreview.net/forum?id=qL_juuU4PY

[2] Amit Chavan. 2015. An Introduction to Graph Compression Techniques for In-memory Graph Computation. University of Maryland, College Park.

[3] Shenfeng Chen and J.H. Reif. 1996. Efficient lossless compression of trees and graphs. In Proceedings of Data Compression Conference - DCC '96, IEEE Comput. Soc. Press. Retrieved April 18, 2022 from http://dx.doi.org/10.1109/dcc.1996.488356

[4] Arindam Pal. Algorithms for graph compression. IIT Delhi / Yahoo! Research. Retrieved from https://www.cse.iitd.ac.in/~arindamp/talks/GraphCompression.pdf

[5] Hannu Toivonen, Aleksi Hartikainen, Fang Zhou, and Atte Hinkka. 2011. Compression of Weighted Graphs. In KDD '11: Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, 973. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.221.1324&rep=rep1&type=pdf

[6] Fang Zhou. 2019. Graph Compression. In Encyclopedia of Big Data Technologies. Springer International Publishing, Cham, 814–814. Retrieved April 18, 2022 from http://dx.doi.org/10.1007/978-3-319-77525-8_100146

[7] Chris Cross. Street Network of New York in GraphML. Kaggle. Retrieved May 21, 2022 from https://www.kaggle.com/datasets/crailtap/street-network-of-new-york-in-graphml?select=manhatten.graphml

[8] Stephen Bailey. Nashville Meetup Network. Kaggle. Retrieved May 21, 2022 from https://www.kaggle.com/datasets/stkbailey/nashville-meetup?select=member-to-group-edges.csv

[9] Ulrik Brandes. GraphML Primer. GraphML. Retrieved May 21, 2022 from http://graphml.graphdrawing.org/primer/graphml-primer.html

[10] Larxel. Spotify Playlists. Kaggle. Retrieved May 21, 2022 from https://www.kaggle.com/datasets/andrewmvd/spotify-playlists?select=spotify_dataset.csv

[11] Daniel Szelogowski. 2022. danielathome19/GraphCompression. GitHub. Retrieved May 21, 2022 from https://github.com/danielathome19/GraphCompression