UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

ON COMPRESSING MASSIVE STREAMING GRAPHS WITH QUADTREES

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

MICHAEL ANDREW NELSON
Norman, Oklahoma
2015

ON COMPRESSING MASSIVE STREAMING GRAPHS WITH QUADTREES

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Sridhar Radhakrishnan, Chair

Dr. John K. Antonio

Dr. K. Thulasiraman

DEDICATION

to

My parents

RONALD JOHN NELSON, and

SHERRI ANN CUMMINGS

For

Being the greatest parents ever

Acknowledgements

First, I would like to thank Dr. Sridhar Radhakrishnan for advising me and providing me with a job that gave me a wealth of experience. I plan on working with him for the next few years and I am greatly looking forward to it. Next, I would like to thank my committee members: Dean Antonio, Dr. Thulasiraman, and Dr. Dhall (although he couldn't stay on the committee). I would also like to thank Amlan Chatterjee for the initial quadtree work and for his tutelage.

Next, I would like to thank OU School of Computer Science staff: Virginie Perez-Woods, Jaime Hoots, and Emily Pierce for helping me through my graduation process and getting me involved in my student community. I also give special thanks to my former project leader and department system administrator, Jonathan Mullen, for helping bounce ideas around.

Finally, I'd like to give thanks to all of my family for being there through all the tough times and always giving me any tools I needed to succeed.

Contents

List of Tables

List of Figures

Abstract

Social networks are constantly changing as new members join, existing members leave, and friendships are formed and disappear. The model that captures this constantly changing graph is the streaming graph model. Given a massive graph data stream wherein the number of nodes is in the order of millions and the number of edges is the tens of millions, we propose a simple algorithm to compress this graph without having read in the entire graph into the main memory. Our algorithm uses the quadtree data structure that is implicitly constructed to produce the compressed graph output. As a result of this implicit construction, our algorithm allows for node and edge additions/deletions that directly modifies the output compressed graph. We further develop algorithms to solve edge queries (is there any between two nodes?) and node queries (for a given node, list all its neighbors) that directly operates on the compressed graph. We have performed extensive empirical evaluations of our algorithms using publicly available, large social networks such as LiveJournal, Pokec, Twitter, and others. Our empirical evaluation is based on several parameters including time to compress, memory required by the compression algorithm, size of compressed graph, and time and memory size required to execute queries. We also compare our results against the Backlinks compression scheme. We have also presented extensions to the compression algorithm that we have developed.

Chapter 1

Introduction

Sociology is the scientific study of social patterns and behavior. This study extends into computer science under many forms, one of which is the study of social networks. Social networks, like all networks, can be represented as a graph of the form $G = (V, E)$, where V is the set of vertices and $E \subset V \times V$ is the set of edges. Under this representation, social networks are susceptible to various methods of analyzing. However, such analysis requires that the representation provides certain query functions over the graph. Ideally, such a structure has a suitable space-time balance such that the graph is efficiently stored while maintaining a reasonable query time.

1.1 Overview

Usually the input to a graph compression algorithm is the graph represented as an edge list. The graphs are large, sparse, and may be either directed or undirected. Before the actual compression takes place, some preprocessing may be done, such as an ordering algorithm like Bread First Search (BFS). The purpose

of such an ordering would be to reorganize the graph such that edges are more closely grouped together. After the edge list is loaded in memory and any pre-processing is done, the compression may run. When the compression is finished, the compressed graph is loaded in memory as a bit-string and may be queried upon or sent to disk.

Two separate techniques are used in this thesis. The first is called Backlinks Compression (BLC) [8] which is a modified version of the Boldi-Vigna Compression [5] such that it is more specifically targeted towards social networks. It is worth noting that the Backlinks Compression was not constructed using one of Boldi-Vigna's extensions that allow for faster querying times. We have provided a version of Backlinks that includes such an extension. The second technique is a new approach called Quadtree Compression (QTC). This novel social network compression involves loading the graph into a quadtree with a coloring scheme placed on the nodes to add compression. It is important to note that QTC results in a streaming compressed graph, while BLC does not. Both of these compressions will be explained and compared in depth in the following chapters.

In the final results of the thesis, we will see how BLC, QTC, and the QTC extensions compare in various aspects. The results collected involve compression time, compression size, query times, and memory needed during compression and queries. The results also include time and memory needed to add edges to the quadtree compressed graph.

1.2 Motivation

Online social networks (OSNs) are graphs representing individuals or entities as nodes and the associations between them as edges. In the popular example of

Facebook, nodes are people and edges are friendships [12]. With the growth of OSNs, various businesses and individuals have found it advantageous to gather knowledge mined from target social networks. Such data is anonymized to describe social patterns rather than personal information.

An online social network is often large and tends to only become larger. For example, from December 2014 to March 2015, the number of daily active Facebook users grew from 890 million to 963 million [9]. Clearly, such large graphs present a challenge to social network analysis.

Consider a social network with a million nodes. The size of an adjacency matrix (with a single bit for edge present/absent) is about 116 GB; hence, most computers will not have such large main memory to load the graph and perform social network analysis. Given this, our target is to not only compress the graph to a size that can fit in the main memory *but also* provide a mechanism to perform node and edge queries on the compressed structure. Running node and edge queries on the compressed structure is time consuming in comparison with the adjacency matrix structure, if only the adjacency matrix fits in main memory. If the adjacency matrix does not fit in main memory, then we need disk accesses and this cost will be significantly higher than executing the queries on the compressed structure.

Many different queries may be executed on OSNs. Arguably, the neighbor query, which finds all neighbors of a given node, is the most important query in a social network. Such a query is the basis of problems such as community finding, network pattern mining, and friend suggestion.

As previously mentioned, OSNs are constantly changing. This can be attributed to either a changing user base or changing relationships. Either way, we can notice how it would be preferable to have a compression scheme which can

modify the compressed graph as nodes and edges are added and removed.

Our novel compression technique is based using on quadtrees [13]. Quadtrees have been a popular data structure for storing image data. Our motivation for using the quadtree springs from the fact that an adjacency matrix can be thought of as an image matrix. After the quadtree is constructed for the adjacency matrix of a given graph, we code the tree in such a way that edge and neighbor queries can be answered very efficiently. The output of the novel coding of the quadtree for an adjacency matrix represents the compressed graph. It is important to emphasize that *our proposed approach does not construct the adjacency matrix or list and constructs the quadtree only implicitly.*

The contributions in this thesis can be summed up as:

- The quadtree compression scheme is introduced, which incrementally constructs the compressed file as nodes and edges are added and removed (*streaming graph model*). To the best of our knowledge, this is the first work describing such a technique. Existing algorithms require the entire adjacency list or matrix information before the construction of the compressed output.

- Algorithms have been developed to execute edge and neighbor queries that directly operate on the compressed file.

- Several extensions to the introduced compression algorithm has been provided to improve both compression size and query times.

- A modification has been made to the Backlinks compression scheme to include an improvement made by Boldi-Vigna [5] to improve query times.

- A detailed empirical study is also presented that uses the SNAP database

[20] to obtain data for several social networks. Note that the data sets from SNAP are all in edge list format. We are able to read in each edge and directly create the compressed file.

1.3 Related Work

Social network compression is preceded by the more general case of web graph compression. In such a case, the web pages are vertices and hyperlinks are directed edges. While we believe that we are the the first ones to provide a compression technique for streaming graph, there are several existing work that rely on the entire adjacency matrix for their compression algorithm that are work noting.

Adler and Mitzenmacher [4] introduced a web graph compression scheme by finding nodes with similar sets of neighbors.

Randall et al. [18] were the first to use the lexicographical ordering of a web pages URL to compress a graph. Their method exploits the fact that many pages on a common host have similar sets of neighbors.

Boldi and Vigna [5] continued taking advantage of properties of web graphs in lexicographical ordering. They found that proximal pages in URL lexicographical ordering often have similar neighborhoods. This lexicographical locality property allowed them to use gap encodings when compressing edges. In order to further improve compression, Boldi and Vigna. [5] developed new orderings that combine host information and Gray/lexicographic orderings.

In 2009, Chierichetti et al. [8] modified the Boldi and Vigna [5] compression method to better target social networks. Their method exploits the similarity and locality properties of web pages along with the idea that social networks

have a high number of reciprocal edges. This method is called the Backlinks Compression scheme and serves as our benchmark for empirical study. We also include a variation of the Backlinks Compression to speed up query times.

In 2010, Maserrat and Pei [17] introduced a compression scheme specifically designed to compress social networks while maintaining sublinear neighbor queries. They achieve this by implementing a novel Eulerian data structure using multi-position linearization. Their results are the first to answer out-neighbor and in-neighbor queries in sublinear time.

In 2014, Lim et al [16] proposed "Slashburn", a new ordering method to run on the graph before compression. The idea is to stray away from the definition of 'caveman' communities and instead use the idea of real world graphs being more like hubs and spoke connected only by the hubs. After running their ordering function, they use a block-wise encoding method (such as gzip) for the actual compression. The technique described in "Slashburn" [16] is novel as that reduces the total number of blocks (where a block is a sub-matrix with non-zero entries). Their query times focus more on the problem of matrix-vector multiplication, which is used in problems such as PageRank, diameter estimation, and connected components [15].

Our proposed work is for streaming graphs and does not have the option of storing the graph in an adjacency matrix or list and renumbering vertices to achieve a good compression (measured as number of bits used per edge on the average). We believe that the algorithms proposed in Maserrat et. al. [17] and Lim et. al. [16] could use our quadtree technique to further reduce the bits per edge quantity.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows. In Chapter 2, we review basic concepts needed to understand compression. Chapter 3 discusses the Backlinks Compression scheme (BLC) and its extension which improves querying times while sacrificing minimal space. Chapter 4 introduces the Quadtree Compression scheme (QTC) and its many extensions that have various space-time tradeoffs. Chapter 5 is a detailed study and comparison of the time-space complexities of BLC and QTC. Conclusion and future work is discussed in Chapter 6.

Chapter 2

Review of basic concepts

In this chapter, we review basic concepts used in the thesis. We cover compression, graph compression, graph representations, and social networks.

2.1 Compression

Compression is performed as follows. Let A be a file of bytes to compress. Apply some compression algorithm $C()$ to A, that is $C(A)$. Now, apply some decompression algorithm $D()$ to $(C(A))$. If $D(C(A)) \rightarrow A$, then $C()$ is a lossless compression. If $D(C(A)) \rightarrow A'$, then $C()$ is a lossy compression. In other words, a lossless compression completely preserves A, while lossy compression loses some information about A.

Usually, a lossless compression results in a larger compressed file than a lossy one. This makes sense because in a lossy compression, information is lost. Also, notice that the size of the original file, A, may be smaller than the compressed file, $C(A)$. In other words, the compression may fail to actually compress the data to a smaller size. Such a behavior is usually attributed to worst-case versions of A

for $C()$. Another possible cause could be the fact that A was too small, as some compression algorithms have a minimum size for A in order for the compression to be successful.

Compression is useful because it reduces the amount of resources used. Hopefully the compression scheme used supports operations on the compressed structure, rather than having to completely decompress before having access to the original data.

2.1.1 Lossless Compression Algorithms

Lossless compression algorithms are used when the data needs to be exactly preserved. This is typically required for text and data files, such as bank records and text articles.

Below are some examples of lossless compression:

- Huffman encoding
- LZ
- DEFLATE
- PNG (Portable Network Graphics)

Huffman encoding [14] is a type of optimal prefix code. It uses a number of bits to represent symbols based on their frequency, with the most frequent only using one bit. LZ [21] is an algorithm designed to reduce the amount of redundant data when compressing. Many lossless algorithms incorporate other lossless techniques. For example, DEFLATE [10] is a compression algorithm using LZ77 and Huffman encoding. PNG is a image format that uses DEFLATE as an optimization.

2.1.2 Lossy Compression Algorithms

Lossy compression algorithms are used when its acceptable to lose information, such as in streaming media or internet telephony. Such losses are sacrifices in the quality of service.

Below are some examples of lossy compression:

- MP3 (MPEG Audio layer 3)
- MPEG-4 (Motion Picture Experts Group)
- JPEG (Joint Photographic Experts Group)

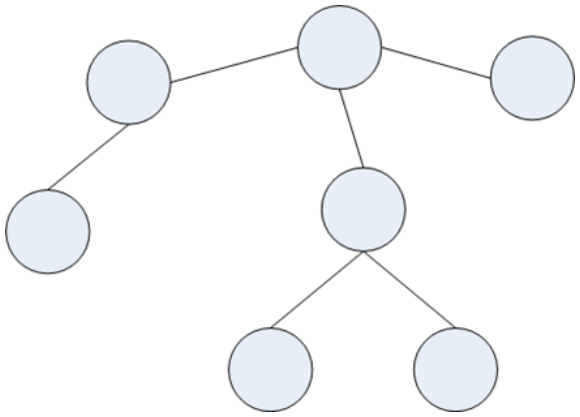
MP3 [1] is a commonly used audio format. When an audio file is compressed, this algorithm removes pieces of the audio that are harder to perceive by humans. MPEG-4 [3] is a method for compressing audio and visual data. MPEG-4 is an amalgamation of several different parts (31 as of 2015) and is still evolving. JPEG [2] is a still image compression based on the discrete cosine transform (DCT). It essentially gets rid of sharp transitions in intensity or color hue from the image.

2.2 Graph Compression

We define a graph as $G = (V, E)$ where V is the set of vertices and E is the set of edges. An illustrative example of a graph is given in Figure 2.2.

Compression on a graph is performed as follows. Let R be some representation of G. An adjacency list is the most commonly used representation. Apply some compression method, $C()$, to R. That is, $C(R) \rightarrow S$, where S is the compressed graph.

The final output, S is a string of bits. Once the user has obtained S, they may perform any operations on S that is supported by the compression. Preferably,



S can be at least be queried without being completely decompressed. Once the user is done with S, they may send it to secondary storage or decompress it back to R.

2.2.1 Lossy Graph Compression

Although most graph compressions are lossless, it is important to see that lossy graph compression is also useful.

Usually, lossy compression involves cutting out parts of the graph. That is, it compresses some graph, $G' \subset G$. This type of compression is useful for algorithms such as determining if a graph is k-connected. It has been shown that if G' is k-connected, then G is also k-connected [7]. Since the graph G' is strictly smaller than G, the k-connected algorithm should run faster.

2.3 Graph Representations

In this section, we review basic graph representations. When choosing an appropriate representation, the user must consider the various space-time trade-offs. If the representation has slow access times, the compression will take longer to

perform. If the representation is too large, it may not fit in memory. The representations reviewed are the adjacency matrix, adjacency list, and edge list. We also cover the space-time trade-offs of these structures.

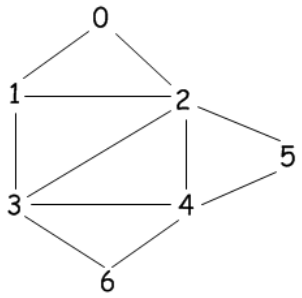


Figure 2.1: A sample graph to convert

2.3.1 Adjacency matrix

When representing a graph G as an adjacency matrix, we create a 2D $n \times n$ matrix, where $n = |V|$ is the number of vertices. Then for each edge $(u, v) \epsilon E$, we mark the $[u][v]$ entry in the matrix with a one. Figure 2.2 is the adjacency matrix representation of Figure 2.1.

2.3.2 Adjacency list

An adjacency list consists of two parts: an array of size n and a collection of pointers. Each index in the array represents the corresponding node. From any given index we have a reference to a chain of pointers to each node that index's node is connected to. This representation is most common in compression methods, due to its agreeable space-time trade-off. An example of this structure is given in Figure 2.3.

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	1	1	1	0
3	0	1	1	0	1	0	1
4	0	0	1	1	0	1	1
5	0	0	1	0	1	0	0
6	0	0	0	1	1	0	0

Figure 2.2: Figure 2.1 as an adjacency matrix

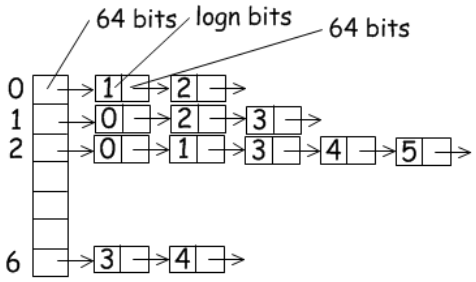


Figure 2.3: Figure 2.1 as an adjacency list

2.3.3 Edge list

An edge list is a simple list of edges. For each entry in the list, we store two integers, (u, v). This is the representation that is usually given to the compression algorithm as input. The compression will usually convert this edge list to an adjacency list for better access times. The edge list is chosen because it is the smallest representation, thus it is best suited in data transfers. Figure 2.4 is an example of an edge list.

0	1
0	2
1	0
1	2
1	3
2	0
2	1
2	3
2	4
2	5
3	1
3	2
3	4
3	6
4	2
4	3
4	5
4	6
5	2
5	4
6	3
6	4

Figure 2.4: Figure 2.1 as an edge list

2.3.4 Space-time trade-offs

In Table 2.1, we review the sizes of the above representations. Clearly, we see that the adjacency matrix consumes the most space, while the edge list requires the least. The math for the adjacency list is more complicated, but as long as the graph is sparse, the adjacency list should be much less space than the adjacency matrix.

Table 2.1: Graph representation sizes

Data Structure	Total bits
Adjacency Matrix	n^2
Array of Linked Lists	$n \times 64 + 2m(logn + 64)$
Edge List	$2m(2logn)$

In Table 2.2, we review the time complexities of the previously mentioned structures. In the table, $d(G)$ is the maximum degree of the graph. That is, the associated method can take no longer than the node with the highest degree.

The following are common operations that can performed on a graph.

- N(v) - give all the neighbors of node v.
- D(v) - give the number of neighbors of node v.
- Edge(u,v) - determine if an edge exists between u and v.
- Add(u,v) - add an edge between u and v.
- Remove(u,v) - remove the edge between u and v.

Table 2.2: Graph representation sizes

Data Structure	N(v)	D(v)	Edge(u,v)	Add(u,v)	Remove(u,v)
Adjacency Matrix	n	n	1	1	1
Adjacency List	$d(G)$	1	$d(G)$	$d(G)$	$d(G)$
Edge List	m	m	m	1	m

After examining the above space-time complexities, we can see that the adjacency list is ideal for graph compression as long as the graph is sparse. We will see in the next section that social networks are indeed sparse.

2.4 Social Networks

We define a social network as a graph $G = (V, E)$ where V is a set of people (vertices) and $E \subset V \times V$ is the relationships between them(edges). In this section we mention the general properties of social network graphs.

Social network graphs are inherently sparse. This means that the ratio of actual edges to all possible edges is incredibly low. For example, in a graph of four million people, one person may only have about 200 friends. In the LiveJournal graph we use, it has about 4.8 million nodes and 69 million edges. Thus, the ratio would be about 2.93×10^{-6}. Furthermore, a sparse adjacency matrix of the graph would be filled mostly with zeros.

These graphs are also highly reciprocal. This means that if u is friends with v, then v is also friends with u. More formally, if an edge $(u, v)\epsilon E$, then $(v, u)\epsilon E$ as well. Notice that in an undirected graph, the number of reciprocal edges is equal to the total number of edges.

Social graphs may also be disconnected. This means that there may be a node that is unreachable from other nodes. An example of this would be a person without any relationships, or a group of people that only know each other.

DEFINITION 1 (*similarity*). *Similarity* is the tendency of nodes that are proximal in the lexicographical ordering to have similar sets of neighbors.

DEFINITION 2 (*locality*). *Locality* is the tendency of nodes to point to nodes nearby in the lexicographical ordering.

DEFINITION 3 (*reciprocal*). In a graph G, if both $(u, v) \in E$ and $(v, u) \in E$ and $u < v$, then we call the edge (v, u) to be *reciprocal*.

2.5 Node Re-ordering

The previous section introduced an interesting concept about social networks; they all have some inherit structure. However, since the ordering of nodes is usually random, this structure is hidden. Thus, we should be able to re-order the nodes such that the true structure of the graph is represented. Such a re-ordering should allow us to exploit patterns and redundancies to achieve better compression.

There is a concept known as the Caveman community [16] in social networks. This essentially means that people only know others inside their "cave" and are oblivious to people outside the "cave." Ordering methods based on this idea, usually focus re-ordering the nodes such that the edges in the adjacency matrix

form blocks. While their are many algorithms to do this type of re-ordering, BFS performs effectively enough. Nodes visited in the BFS order are reassigned the number in which they were visited. This process is called lexicographical BFS re-ordering and is given in Algorithm 1.

Algorithm 1: Lexicographical BFS re-ordering

```
   Input: the adjaceny list adj
   Output: an array with the new ordering, order
 1 begin
 2     num_v ← adj.size();
 3     queue < int > q;
 4     vector < int > order(n);
 5     k ← 0;
 6     for i = 0; i < num_v; ++i do
 7         if order.at(i)! = −1 then
 8             continue;
 9         order.at(i) = k++; q.push(i); while !q.empty() do
10             v ← q.front();
11             q.pop();
12             for j = 0; j < adj[v].size(); ++j do
13                 u = adj[v][j];
14                 if order.at(u)! = −1 then
15                     continue;
16                 order.at(u) = k++;
17                 q.push(u);
```

2.6 Query Definitions

The first and most basic query to be executed is the edge existence query. Given a graph $G = (V, E)$, return *true* if an edge $(u, v) \in E$ and *false* otherwise.

Next, the neighbor query must be divided into two parts. For a vertex $u \in V(G)$, $v_1 \in V(G)$ is an out-neighbor of u if $(u, v_1) \in E(G)$.

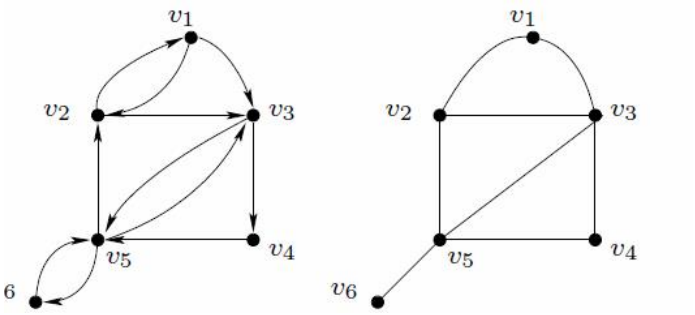


Figure 2.5: A directed graph (a) and an undirected graph (b)

Similarly, $v_2 \in V(G)$ is an in-neighbor of u if $(v_2, u) \in E(G)$. Our two subqueries can be used to find all out-neighbors and all in-neighbors for a given node.

Example 1 (neighbor query): In Figure 2.5 (a) an out-neighbor query on v_1 returns $\{v_2, v_3\}$, and an in-neighbor query on v_3 returns $\{v_1, v_2, v_5\}$

Chapter 3

Backlinks Compression (BLC)

The Backlinks (BL) compression scheme is a modified version of the Boldi-Vigna (BV) compression scheme. Therefore, it is necessary to first describe the BV method and then introduce the modifications made by BL.

The BV compression scheme is based off three main ideas. First, it incorporates the notion of similarity. In other words, if a graph has nodes whose neighborhoods are similar, then those nodes' neighborhoods can be described by nodes with similar neighborhoods. This functionality is provided by the so-called copying step. Second, it exploits the locality of the graph by encoding the destination of edges with small integers relative to the sources. Third, gap encodings are used to store a sequence of edge destinations. The idea behind this is that even if the integers of the edge destinations are large, the gaps between them should be small.

BL compression incorporates an additional idea on top of BV, namely link reciprocity. Since social networks are highly reciprocal, this property is valuable. For each edge encoded, an additional bit is also added representing whether that edge is reciprocal. Thus, when its time for the reciprocal edge to be encoded, its

simply skipped.

3.1 Integer Encoding

When we compress via Backlinks, we will essentially end up with a sequence of integers. Those integers will needed to be represented in order in the final bitstring. However, we will need to come up with some method for unambiguously appending bit together. For example, if we want to encode the integer sequence $1, 2$ into a bitstring, it would look like: 110. However when reading this bitstring, we don't know where the 1 and 2 begin. Thus, we could possibly read 110 as 6.

There are many solutions for such a problem. One approach is to use special bitstrings as delimiters to mark the beginning and ending of each entry. These delimiters are of some constant size x. Thus, if we are encoding n integers, we will have nx bits dedicated to delimiting. For large values of n and small integer size, this approach can be inefficient.

The method we use is an integer encoding algorithm, which is specifically designed to unambiguously encode integers without using delimiters. For this thesis, we use the Elias' delta code [11]. When the Backlinks compression appends integers to the bitstring, it will first pass the integer to the encoder.

The algorithm for Elias' code is given in Algorithm 2. Looking at the algorithm, we can see that it consists of the following steps:

1. Let $length\_val = \lfloor \log_2(val) \rfloor$ be the highest power of 2 in val, so $2(length\_val)val < 2(length\_val) + 1$.

2. Let $length\_length\_val = \lfloor \log_2(length\_val + 1) \rfloor$ be the highest power of 2 in $length\_val + 1$, so $2(length\_length\_val)length\_val + 1 < 2L + 1$.

3. Write *length_length_val* zeros, followed by

4. a one, followed by

5. the *length_length_val* + 1-bit binary representation of *length_val* + 1, followed by

6. all but the leading bit (i.e. the last *length_val* bits) of *val*.

Algorithm 2: Elias' delta code

Input: an integer, *val*
Output: the encoded integer as a bitstring

```
1 begin
2     length_val ← log2(val + 1) − 1;
      length_length_val ← log2(length_val + 1) − 1; for
      i = 0; i < length_length_val; i++ do
3         bitstring.append (0);
4     bitstring.append (1);
5     for i = length_length_val − 1; i ≥ 0; i−− do
6         bitstring.append ((length_val + 1) >> i&1);
7     for i = length_val − 1; i ≥ 0; i−− do
8         bitstring.append ((val + 1) >> i&1);
```

The integer decoding algorithm is given in Algorithm 3. It should be obvious how it uses the information from Algorithm 2 to decode the integer.

3.2 The Compression Algorithm

Assume that some ordering method has already been applied to the graph. That is, the edge list has already been loaded into memory, inserted into an adjacency list, and had an algorithm such as BFS run on it. This provides an additional array of size n which is a mapping for the new lexicographical ordering. The

Algorithm 3: Elias' delta code - decode

```
Input: a bitstring in, the position to decode from i
Output: the decoded integer
1  begin
2      length_length_val ← 0;
3      while !in.GetBit(i) do
4          ++length_length_val;
5          ++(i);
6      ++(i);
7      length_val ← 1;
8      for j = 0; j < length_length_val; ++j do
9          length_val ← (length_val << 1)|in.GetBit(i);
10         ++(i);
11     −−length_val;
12     val ← 1;
13     for j = 0; j < length_val; ++j do
14         val ← (val << 1)|in.GetBit(i);
15         ++(i);
16     returnval − 1;
```

integer encoding may be swapped out, but for our purposes we use Elias delta code [11]. The BL algorithm is applied as follows:

To further describe the algorithm, we can see that it consists of four parts:

1. *Base Information.* The current node we are compressing, the potential previous node we are copying neighbors from, and a bit representing if there is a self-loop on the current node.

2. *Copying.* The node v is being encoded based on the node u. Note that $v \geq u$. If $v = u$, then no copying is performed. Otherwise, add a bit for each out-neighbor of u, indicating if it is also an out-neighbor of v.

3. *Residual edges.* Let the list of out-neighbors of v that have yet to be encoded be, $v_1 \dots v_k$. Sort this list in increasing order. Gap encode the resulting

Algorithm 4: Backlinks compression

Input: Graph as an adjacency list
Output: Compressed graph as a bitstring

```
1  begin
2      for i = 0, i < |V|, i++ do
3          for j = i; j > i WINDOW_LENGTH &&j >= 0; j-- do
4              bitstring.append (i − j) // copy vertex
5              bitstring.append ((i, i) ∈ E) // self-loop?
6              if i ≠ j // copy neighbors then
7                  for each neighbor of j do
8                      if also neighbor of i then
9                          bitstring.append (1)
10                     else
11                         bitstring.append (0)
12             bitstring.append (# of remaining edges)
13             for each remaining edge //in sorted order do
14                 // gap encode
15             for  each out-neighbor k of i //in sorted order do
16                 bitstring.append (hasReciprocal (k, i))
```

list, storing a sign bit as appropriate.

4. *Reciprocal Edges.* This is the part unique to BL. For each out neighbor u of v, such that $u > v$, encode one bit representing whether $u \in rec(v)$ or not.

3.3 Backlinks Compression with Indexes

In its current form, the Backlinks compressed graph must be decoded sequentially when being queried. The resulting function is also broken into two versions. The first is a forgetting algorithm which does not keep any information about the nodes its already decoded. The second version keeps a structure in memory during the query such as an adjacency list. Nodes decoded along the way are stored in the adjacency list. The difference between the two is that, the former version, the recursive call for decoding the copying step will take longer as the copy chain increases.

A solution to the sequential access problem is to include the position of each node in the bit-string. This information is included at the very beginning of the bit-string. Thus, when a node in the graph is being queried, we can jump straight to it in the bit-string and decode it immediately. This also helps with the recursive copy chain decoding. Just as with the original node, we can jump straight to the node we are copying from to decode it.

3.4 Querying the Compressed Graph

Querying this structure is a simple as decoding the information provided for each node. Since we modified the original Backlinks compression, we have direct access

to the nodes' positions in the bit-string. Algorithm 5 details the query function.

Algorithm 5: Backlinks Query

```
   Input: Graph as an adjacency list, x, y
   Output: True or False
 1 begin
 2     cur ← 0;
 3     for i = 0, i < |V|, i++ do
 4         index[i] = bitstring.GetNextIndex(cur++);
 5     cur ← index[x];
 6     j ←bitstring.GetBit (cur++) // copy vertex
 7     self ←bitstring.GetBit (cur++) // self-loop?
 8     if selfandx == y then
 9         return true;
10     if x ≠ j then
11         // recursive decoding
12     numResidual ←bitstring.GetNextIndex (cur);
13     if numResidual ≠ 0 then
14         sign ← bitstring.GetBit(cur++)?1 : −1;
15         now ← x + bitstring.GetNextIndex(cur++) ∗ sign;
16         if now == to then
17             return true;
18         for j = 1; j < numResidual; j++ do
19             now ← bitstring.GetNextIndex(cur++);
20             if now == y then
21                 return true;
22     for k = 0; k < numNeighbors; k++ do
23         if bitstring.GetBit(cur++)andneighbors[k] == y then
24             return true;
25     return false;
```

We can see that this algorithm proceeds through the following steps:

1. *Indexes.* We read in the list of indexes at the beginning of the bit-string. Then we set our current index to its position stored in that list.

2. *Base Information.* The node we copied from is decoded into j. If it is different from the node x we are currently on, we need to perform the

recursive copy decoding. Next the self loop bit is decoded. If $x = y$ we return based on the self loop bit.

3. *Copying.* Remember, if $x = j$, then no copying is performed. Otherwise, we must recursively decode each node along the copy chain. The base case is thus a node v where the decoded copy node $u = v$. At the end of the recursion, we should have a partial list of neighbors of x. If y is in this list, we return true.

4. *Residual edges.* In this step we simply reverse the gap encoding to get the list of residual neighbors. If y is in this list, we return true.

5. *Reciprocal Edges.* We loop through each node in the sorted list of neighbors collected so far. If the next bit decoded is 1, we return true.

This same algorithm can be performed when querying for out-neighbors. Instead of returning when we find a neighbor, we keep a collection of all neighbors decoded and return it. Note that main source of complexity is the recursive copy step, which depends on the length of the copy chain. Finally, notice that this compression does not contain any support for in-neighbor queries. The entire list of nodes must be decoded to return the list of in-neighbors.

Chapter 4

Quadtree Compression (QTC)

The quadtree data structure was first conceived in 1974 by Kinkel and Bentley [13]. Provided along with the definition of the data structure were several algorithms related to construction, insertion, deletion, etc. However, to the best of our knowledge, no one has yet adapted the data structure and its algorithms to compress social networks as we have.

4.1 Quadtrees Reintroduced

A quadtree is a data structure which is used to normally represent images using partitioning of the two dimensional space by recursively subdividing into four quadrants or regions; each internal node of the quadtree has exactly four children [19]. The most common type of quadtree is the Point-Region Quadtree, also referred to as the PR quadtree.

A PR quadtree represents data points in a two dimensional region. The region is subdivided into four quadrants if it contains more than 1 point in it. If a region contains a single point or no points, it is designated by a leaf node

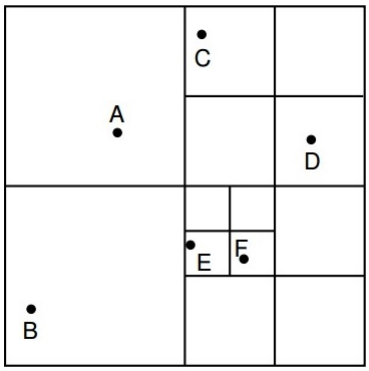


Figure 4.1: A map of data points for a two dimensional region

in the representation. A sample region data is shown in Figure 4.1 and the corresponding PR quadtree is shown in Figure 4.2.

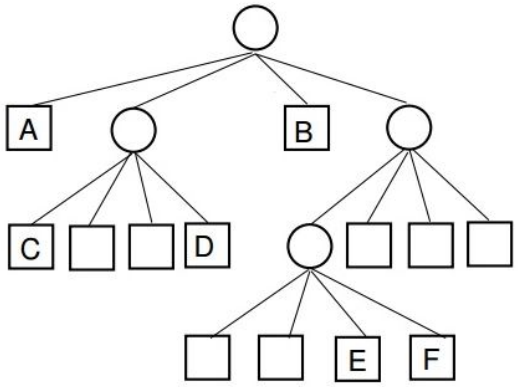


Figure 4.2: The PR Quadtree for the region shown in Figure 4.1

4.1.1 Graphs as Quadtree

Quadtree representation can also be used to store graphs efficiently. The quadrants of an adjacency matrix of a graph can be converted and stored according to the row major order. The decision to further expand a quadrant into four sub-quadrants is taken based on the data present in the same quadrant. If the

data matches with one of the first three patterns mentioned below , then the quadrant is represented as a leaf node in the tree. Quadrants matching a pattern can be stored internally using integer values that map to the specific pattern. Therefore, using the quadtree representation, the entire graph information can be stored in the form of an array using bits. The contents of the bit array can be stored as follows:

- **0**: all 0's in quadrant
- **1**: all 1's in quadrant
- **2**: 0's in diagonal, and rest 1's
- **3**: the quadrant needs to be expanded further

Since there are only 4 types of values, using 2 bits to represent each quadrant is enough when storing in the bit array. Consider the following graph shown in Figure 4.3.

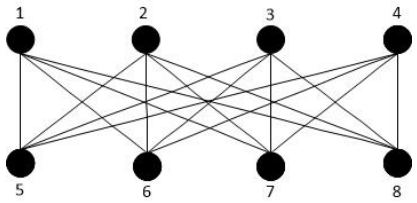


Figure 4.3: A sample graph

To convert to a quadtree, the adjacency matrix of the graph is considered. The adjacency matrix for the graph in Figure 4.3 is given in Figure 4.4. The quadtree representation of the graph given in Figure 4.3 is shown in Figure 4.5. It should be noted that, the generation of quadtree based on adjacency matrix information in this Section is provided as a background to explaining the data

structure; the algorithms introduced for performing the empirical analysis *does not* construct the adjacency matrix.

0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0

Figure 4.4: Adjacency matrix of graph shown in Figure 4.3

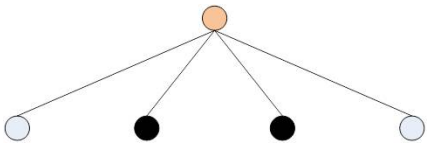


Figure 4.5: Quadtree representation of graph shown in Figure 4.3

The byte representation of the quadtree is given by $Q = \{3, 0, 1, 1, 0\}$. The value 3 corresponds to the quadrant covering the entire adjacency matrix. The following values of 0, 1, 1 and 0 represents the top-left, top-right, bottom-left and bottom-right quadrants respectively.

Algorithm 6 describes the method for generating the quadtree from the adjacency matrix of a graph. The method *CheckMatrixUniformity* checks whether the adjacency matrix provided as the parameter is uniform or not. Depending on the data, the method returns a 0, 1, 2 or 3 for all 0's, all 1's, 0's in diagonal and rest 1's, and non-uniform matrix respectively. For the first 3 cases, the returned value is added to the quadtree representation; for the last case Algorithm 6 is called recursively for all the quadrants of the matrix generated by using the method *DivideMatrixIntoQuadrants*. For a graph $G = (V, E)$, where $|V| = n$,

Algorithm 6: QuadGen: Quadtree generation from adjacency matrix

```
Input: Adjacency matrix A[ ][ ] of graph G
Output: Quadtree Q of G
1  begin
2      Code ← CheckMatrixUniformity(A);
3      if Code = 0 then
4          Concatenate(Q, 0);
5      else if Code = 1 then
6          Concatenate(Q, 1);
7      else if Code = 2 then
8          Concatenate(Q, 2);
9      else
10         Quadrants{} ← DivideMatrixIntoQuadrants(A) ;
11         forall the Quadrants_i ∈ Quadrants{} do
12             QuadGen (Quadrants_i);
13     Output ← Q;
```

the algorithm loops over total of n^2 elements in each level of the quadtree. Since in the worst case there are $\log_2 n$ levels, the time complexity of the algorithm for generating the quadtree from the adjacency matrix is $O(n^2 \log_2 n)$.

4.1.2 Compression using Quadtree

The sample graph shown in Figure 4.3 consists of 8 nodes. Therefore, the space required to store the graph using the adjacency matrix representation is $8 \times 8 = 64$ bits. Now, the corresponding quadtree representation shown in Figure 4.5 when stored in the bit array requires information for 5 elements, each of which requires 2 bits of data. Therefore, the space required to store the quadtree representation is $5 \times 2 = 10$ bits. Hence, in this case, the graph data is compressed using the quadtree representation, and requires just $\frac{1}{6}$ of the original space. So, efficient compression can be achieved by using quadtree representation of graphs.

Let the graph we are compressing be $G = (V, E)$, where V is the set of

vertices and E is the set of edges. Let $n = |V|$ and choose an n' such that $2^x < n \leq n' \leq 2^{x+1}$. In other words, n' should be n rounded up to the next highest power of two. Next, take the $n \times n$ adjacency matrix representing G and extend it to an $n' \times n'$ adjacency matrix, filling in the empty cells with zeros. This adjacency matrix is the visual representation of what the quadtree is compressing. An example of this process is the conversion of the graph in Figure 1(a) to the adjacency matrix in Figure 2.

4.2 Edges as Quadstrings

At the beginning of compression, the algorithm will be given the graph in edge list form. A helpful bit of preprocessing would be to convert each edge into a form describing its location in the quadtree. This form is called a quadstring, and an example of converting an edge to a quadstring is given in Example 2.

Let the graph we are compressing be $G = (V, E)$, where V is the set of vertices and E is the set of edges. Let $n = |V|$ and choose an n' such that $2^x < n \leq n' \leq 2^{x+1}$. In other words, n' should be n rounded up to the next highest power of two. Next, take the $n \times n$ adjacency matrix representing G and extend it to an $n' \times n'$ adjacency matrix, filling in the empty cells with zeros. This adjacency matrix is the visual representation of what the quadtree is compressing. An example of this process is the conversion of the graph in Figure 2.5(a) to the adjacency matrix in Figure 4.6.

The algorithm takes the value n', and the end-point nodes of an edge (x, y). The process continues to determine the location of the edge recursively in a quadrant of an equivalent adjacency matrix representation. The quadrants are ordered as follows: $NW = 0, NE = 1, SW = 2, SE = 3$. A formal algorithm for

0	1	1	0	0	0	0	0
1	0	1	0	0	0	0	0
0	0	0	1	1	0	0	0
0	0	0	0	1	0	0	0
0	1	1	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 4.6: 8x8 adjacency matrix representation of Figure 2.5(a). Note the nodes have been re-ordered with a zero index

the process is given in Algorithm 7.

Algorithm 7: Edge to quadstring

Input: n', x, y
Output: (x, y) as quadstring

```
1  begin
2      begRow ← 0, begCol ← 0;
3      endRow ← n′, endCol ← n′;
4      quadstring ← “”;
5      while !leafQuadrant do
6          midRow ← (begRow + endRow)/2;
7          midCol ← (begCol + endCol)/2;
8          qx ← x < midRow ? 0 : 1;
9          qy ← y < midCol ? 0 : 1;
10         begRow ← (begRow × !qx) + (midRow × qx);
11         begCol ← (begCol × !qy) + (midCol × qy);
12         endRow ← (endRow × qx) + (midRow × !qx);
13         endCol ← (endCol × qy) + (midCol × !qy);
14         quadstring.append ((2 × qx) + qy)
```

Example 2 (Quadstring): The quadtree's adjacency matrix representation of the graph in Figure 2.5(a) is given in Figure 4.6. The quadstring form of the edge $(0, 1)$ is 001. From the root, we move to the NW quadrant (0), then the NW quadrant again (0), and finally the NE quadrant (1).

4.3 The Compression Algorithm

There are two methods to construct the compressed quadtree. The first is by calling the insertion method for each edge. This method takes $O(|E|log_4(n'))$ construction time. A more interesting method, and more suitable for social networks, is to construct the quadtree when all the edges are given at once. It turns out, this type of construction can be done in $O(|E|)$ time. Algorithm 8 formally describes the construction.

Algorithm 8: Quadtree Compression

```
   Input: The graph as an edge list
   Output: The compressed graph as a bitstring
1  begin
2      for each edge (u, v) do
3          quadstrings.append (ToQuadstring (n', u, v));
4      sort (quadstrings);
5      curStr ← ToQuadstring (n', 0, 0);
6      for each quadstring, qs do
7          dcq ← deepestCommonQuad (curStr, qs);
8          for each node from curString up to dcq do
9              color rest of children with 0's
10             color parent
11         for each node from dcq down to qs do
12             color children up to qs
13             color parents if last child
14         curStr ← qs
15     dcq ← deepestCommonQuad (curStr, ToQuadstring (n', n' − 1, n' − 1));
16     for each node from curString up to dcq do
17         color rest of children with 0's
18         color parent
19     for each node from dcq down to qs do
20         color children up to qs
21         color parents if last child
```

Lines 1 through 4 involve converting the edge list to a sorted list of quad-

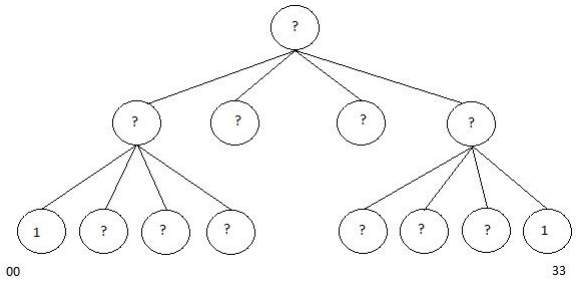


Figure 4.7: Uncolored quadtree of a 4x4 adjacency matrix with an edge at (0,0) and (3,3).

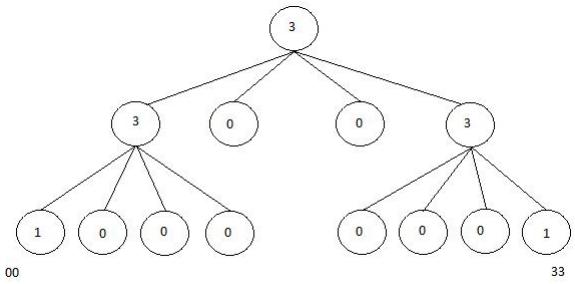


Figure 4.8: Colored quadtree of a 4x4 adjacency matrix with an edge at (0,0) and (3,3).

strings. The quadstring conversion for an edge is given by Algorithm 7.

Lines 6 through 14 process each of these quadstrings. This process is visualized by Figure 4.7 and Figure 4.8. In these examples, the deepest common quadrant, as referenced in lines 7 and 13, is the root node. In other words, the deepest common quadrant is the deepest node in the quadtree that two nodes have in common.

In Figure 4.7, the algorithm needs to color the quadrants between the edges (0,0) and (3,3). Since the edges are in the quadtree ordering, we know that all leaf quadrants between them are 0s. Figure 4.8 shows the colorings after the leaf quadrants are marked 0 and their parents are colored appropriately.

Finally, lines 15 through 21 deal with all the quadrants after the last quad-

string. This section fills the rest of the quadrants with zeros.

To summarize, we know from our sorted list of quadstrings that each edge is in the order that would appear in the tree. Because of this, we can construct each depth of the tree at the same time. To do this, we maintain a list of pointers to beginning of each depth in the tree. Any node added to a depth is simply appended on.

4.4 Reading the Compressed Quadtree

Next, we provide a method of querying this structure with minimal overhead in respect to memory. This method is described by Algorithm 9

As an explanation of Algorithm 9, normally with a non-compressed quadtree, we could calculate exactly where a node is and jump straight to it. However, since some nodes are pruned out due to compression, we need to traverse through the bitstring and keep track of the node we are currently interested in. We do this by going through each node in each depth of the tree.

We start by grabbing the color of the current quadrant we're in. Line 7 does this by getting the next two bits at position j. If this is a node we are interested in, we update our next index or return based on what color it is. If not, we simply update our calculations for how many nodes are in the next depth.

Notice that the queries perform differently based on where they are in the quadrant. It varies slightly due to compression, but queries in the NW quadrants are resolved faster than ones in the SE quadrants.

Obviously the brute force approach for the neighbor query would be to run the edge query for each possible neighbor. Instead, we slightly modified the edge query to maintain certain information for each node in the quadtree that we

Algorithm 9: Quadtree Compression - Edge Query

Input: The compressed graph as a bitstring, x, y
Output: True or False

```
1  begin
2      cur ← 0;
3      begRow ← 0, begCol ← 0;
4      endRow ← n′, endCol ← n′;
5      nextIndex ← 0, nodesAtDepth ← 1, nodesAtNextDepth ← 0;
6      for i = 0; i < bitstring.Size(); do
7          for j = i; j < nodesAtDepth; j++ do
8              curQuad ← bitstring.GetQuad (j);
9              if j = nextIndex then
10                 if curQuad = 3 then
11                     //udpate nextIndex
12                     nodesAtNextDepth ← nodesAtNextDepth + 4;
13                 else if curQuad = 1 then
14                     return True;
15                 else if curQuad = 0 then
16                     return False;
17                 else if curQuad = 2 then
18                     if begCol − begRow + x = y then
19                         return False;
20                     else
21                         return True;
22             else if curQuad = 3 then
23                 nodesAtNextDepth ← nodesAtNextDepth + 4;
24         i ← i + nodesAtDepth;
25         nodesAtDepth ← nodesAtNextDepth;
26         nodesAtNextDepth ← 0;
```

are interested in. In Algorithm 9, the method must keep track of the bounding box of the current quadrant we are in, as well as the index of the next node to consider. In the neighbor query, this information must be tracked for each node in the tree that we are interested in. The number of nodes doubles at each depth. This method takes slightly more memory and requires us to traverse more of the compressed bitstring.

4.5 Edge Streaming

The technique to add edges is also similar to the edge query method. The algorithm searches for the location where the edge should be in the quadtree. When it finds a leaf node, it checks if the node needs to be expanded. Nodes colored as 2's and nodes that are not at the maximum depth and are colored 0 are the only cases in which the node must be expanded. After expansion, the algorithm traverses to one of the four newly created nodes. This process repeats until the node cannot be expanded further. By this point the algorithm marks the current node as 1 and then climbs back up the tree, updating the compression along the way. This process is formally described in Algorithm 10.

Algorithm 10 can be easily modified to include removing edges. Instead of expanding nodes colored with a 2 or 0, we'll need to expand nodes with 2's or 1's. The leaf node is colored with a 0 and compression is updated as normal.

A significant time factor in this method is how the bitstring is manipulated while the compression is being updated. Currently, the bits in the bitstring are simply slid around at each update. However, a possible modification could be to maintain a list of updates to the bitstring and then batch them all into one write.

Along with edge addition and removal, our quadtree compression also sup-

ports adding nodes. Since our compression already expands n to n', modifications only need to be made if adding a node exceeds the limit of n'. In such a case, we simply increase n' to n'' (the next highest power of two). The root of the old quadtree is then set to the NW quadrant of the new quadtree, and the rest of the new quadrants are compressed appropriately.

4.6 Quadtree Extensions

There are three additional techniques added to the quadtree compression. Each of these are used individually or combined in some way to make different variants of the quadtree compression.

1. *QTC-H* Applies Huffman compression to the nodes.
2. *QTC-DP* Includes the pointers to each depth at the beginning of the quadtree.
3. *QTC-HDP* Combines the Huffman compression and depth pointer approaches.
4. *QTC-BFS-HDP* Performs the Breadth First Search ordering on the nodes before QTC-HDP begins.

4.6.1 Huffman Compression

The Huffman code [14] is a type of prefix code used for lossless data compression. During compression, we keep track of the total number of each color in the quadtree. Then we assign one bit to represent the most frequently used on, two bits to encode the second most frequently used, and so on. This method assumes that the color counts differ by large number, which they do in social networks.

Algorithm 10: Quadtree Compression - Add Edge

Input: Compressed graph $G = (V, E)$ as a bitstring, x, y
Output: Compressed graph $G' = (V, E \cup (x, y))$

```
1  begin
2      cur ← 0;
3      begRow ← 0, begCol ← 0;
4      endRow ← n′, endCol ← n′;
5      nextIndex ← 0, nodesAtDepth ← 1, nodesAtNextDepth ← 0;
6      for i = 0; i < bitstring.Size(); do
7          for j = i; j < nodesAtDepth; j + + do
8              curQuad ← bitstring.GetQuad (j);
9              if j = nextIndex then
10                 if curQuad = 3 then
11                     //udpate nextIndex
12                     nodesAtNextDepth ← nodesAtNextDepth + 4;
13                 else if curQuad = 1 then
14                     return;
15                 else if curQuad = 2 then
16                     //expand
17                     //update nextIndex
18                 else if curQuad = 0 then
19                     if !isLeaf then
20                         //expand
21                         //update nextIndex
22                     else
23                         //mark current quadrant as 1
24                         //update compression
25                         return;
26             else if curQuad = 3 then
27                 nodesAtNextDepth ← nodesAtNextDepth + 4;
28         i ← i + nodesAtDepth;
29         nodesAtDepth ← nodesAtNextDepth;
30         nodesAtNextDepth ← 0;
```

In our study, we have found the color 0 to be the most frequent. This is because zeros are the majority of the adjacency matrix, as the graph is sparse. The next highest color was 3. The color 1 was after that, with 2 being the lowest used. This also makes sense is the color 2 can be thought of as a special case.

Applying Huffman compression on the quadtree results in a noticeable space-time trade-off. While it does reduce the size of the compressed graph, the extra overhead in decoding slows down access times. It also means we can't use certain optimizations that take advantage of the fact that all nodes in the bit-string are two bits.

4.6.2 Depth Pointers

In Algorithm 9 we kept track of how many nodes were in the current and next depths. Notice when we reach one of our nodes of interest we no longer need any more information about the current depth. Thus, a possible optimization is to include the position of the beginning of each depth in the quadtree. Now when we are done with our current depth, we can jump straight to the next one.

As one would expect, this additional information is negligible as far as compression size and time is concerned. Also, this method mainly optimizes queries on nodes that are earlier in the quadtree ordering. In other words, nodes from the NW quadrant are optimized more than nodes in the SE quadrant.

4.6.3 Lexicographical BFS Re-ordering

Like any compression scheme, it is possible to perform some preprocessing to help compress. The process or reordering of the nodes in this case is done by the BFS algorithm. This is a typical approach as exploits the idea of 'caveman' com-

munities. The idea is to group nodes with similar neighbors in the lexicographic ordering.

In order to run the BFS algorithm, an adjacency list must be constructed. However, since the adjacency list is not required for the quadtree compression, the list may be freed by memory as soon as the BFS algorithm is finished.

Chapter 5

BLC vs QTC - Results

Our experiments involve performing Backlinks compression, quadtree compression and the various quadtree compression extensions on the datasets listed in Table 5.1. The empirical evaluation is based on several parameters, and those are described in Table 5.2. The datasets are stored as an edge list and the final output of the compressions is a bitstring. All space and time units reported in this Section are in mega-bytes and seconds, respectively. The results for the analysis on anonymized Facebook dataset is reported in Table 5.3. Similarly, the results for the LiveJournal, LiveJournal.com, Pokec and Twitter datasets are reported in Table 5.4, Table 5.5, Table 5.6 and Table 5.7 respectively.

The algorithms performed are as follows:

1. QTC - The standard quadtree compression.
2. QTC-H - QTC with Huffman compression.
3. QTC-DP - QTC with depth pointers.
4. QTC-HDP - QTC with huffman compression and depth pointers.

5. QTC-BFS-HDP - QTC-HDP with breadth-first-search re-ordering.

6. BLC - Backlinks compression with the indexes extension.

The environment these experiments were performed in consisted of the following:

1. Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz

2. Cpu cores: 8

3. Cpu cache size: 10240 KB

4. MemTotal: 8100016 KB

Table 5.1: The dataset stats

	Directed?	|V|	|E|	#Reciprocal Edges
Facebook	FALSE	4,039	88,234	88,234
LiveJournal	TRUE	4,847,571	68,993,773	26,142,536
LiveJournal(com)	FALSE	3,997,962	34,681,189	34,681,189
Pokec	TRUE	1,632,803	30,622,564	8,320,600
Twitter	TRUE	81,306	1,768,149	425,853

In Table 5.1 we display the properties of the various graphs we test against. Its important to look at both directed and un-directed graphs, since in un-directed graphs, every edge is also a reciprocal edge. Note that these graphs are extremely sparse. For example, the edge density of the LiveJournal graph is $2.94 * 10^{-6}$.

We outline the attributes examined in Table 5.2. C-TIME is the time the compression algorithm took to compress the graph. This depends largely on the complexity of the compression algorithm, including whether it needed an adjacency list. C-SIZE is the final size of the compressed graph. C-MEM is the memory usage taken up during compression. It includes the edge list, compressed

Table 5.2: Result attribute descriptions

Attribute	Description
C-TIME	Time to compress the graph (seconds)
C-SIZE	Size of the compressed graph (mega-bytes)
C-MEM	Memory usage during compression
E-TIME	Time to run 1000 edge queries (seconds)
E-MEM	Memory usage during edge query (mega-bytes)
N-TIME	Time to run neighbor queries (seconds)
N-MEM	Memory usage during neighbor queries
ADD-TIME	Time to add edges (seconds)
ADD-MEM	Memory usage during edge adding (mega-bytes)

graph, and any structures used by the compression algorithm (such as an adjacency list). E-TIME, N-TIME, and ADD-TIME are the times taken to perform an edge query, neighbor query, and add edge, respectively. E-MEM, N-MEM, and ADD-MEM are the memory usages during their respective operations. Since these methods are designed to use minimum space, the size is essentially equivalent to C-SIZE.

5.1 Graph Compression

In this section, we examine the attributes related to compression: C-TIME, C-SIZE, and C-MEM.

Looking at the compression times, we see that QTC takes less time to compress than BLC. This is mainly attributed to the fact that BLC needs to construct an adjacency matrix to perform. Another possible explanation is because BLC has a window size which decides how far back a node may look for a copy candidate. Also note that, as expected, adding Huffman compression increases the QTC time to compress, and the depth pointers are negligible as far as compression time is concerned.

Table 5.3: Facebook dataset results

	QTC	QTC-H	QTC-DP
C-TIME	0.47	0.76	0.26
C-SIZE	0.154	0.118	0.154
C-MEM	2.528	2.384	2.536
E-TIME	0.22	0.54	0.43
E-MEM	0.154	0.118	0.154
N-TIME	13.49(x1000)	13.79(x1000)	9.23(x1000)
N-MEM	0.155	0.119	0.43
ADD-TIME	0.94	1.41	0.74
ADD-MEM	0.16	0.128	0.158

	QTC-HDP	QTC-BFS-HDP	BLC
C-TIME	0.76	0.7	1.33
C-SIZE	0.118	0.123	0.11
C-MEM	2.388	4.032	3.864
E-TIME	0.46	0.48	0.02
E-MEM	0.118	0.123	0.11
N-TIME	9.85(x1000)	0.48(x1000)	0.02(x1000)
N-MEM	0.118	0.124	0.11
ADD-TIME	1.02	1.08	-
ADD-MEM	0.122	0.127	-

Table 5.4: LiveJournal dataset results

	QTC	QTC-H	QTC-DP
C-TIME	392.42	409.96	394.66
C-SIZE	336.045	229.207	336.046
C-MEM	1448.7	1186.492	1448.708
E-TIME	256.22	574.84	157.32
E-MEM	336.045	229.207	336.046
N-TIME	230.14(x10)	220.75(x10)	200.75(x10)
N-MEM	336.045	229.207	336.046
ADD-TIME	343.21	722.13	211.23
ADD-MEM	336.046	229.208	336.048

	QTC-HDP	QTC-BFS-HDP	BLC
C-TIME	492.19	568.71	700.78
C-SIZE	229.208	243.974	139.979
C-MEM	1186.78	2200.168	2080.732
E-TIME	420.59	404.71	3.19
E-MEM	229.208	243.974	139.979
N-TIME	250.03(x10)	260.04(x10)	3.19(x1000)
N-MEM	229.208	243.974	139.979
ADD-TIME	578.23	493.54	-
ADD-MEM	229.21	243.974	-

Table 5.5: LiveJournal(com) dataset results

	QTC	QTC-H	QTC-DP
C-TIME	189.51	197.88	187.19
C-SIZE	161.875	110.85	161.875
C-MEM	762.34	631.372	762.34
E-TIME	67.61	185.14	38.75
E-MEM	161.875	110.85	161.875
N-TIME	113.00(x10)	100.94(x10)	20.72(x10)
N-MEM	161.875	110.85	161.875
ADD-TIME	151.44	421.82	101.99
ADD-MEM	161.875	110.85	161.877

	QTC-HDP	QTC-BFS-HDP	BLC
C-TIME	243.4	263.56	439.44
C-SIZE	110.85	133.084	110.026
C-MEM	631.372	1278.928	1229.644
E-TIME	134.85	220.73	2.55
E-MEM	110.85	133.084	110.026
N-TIME	100.31(x10)	140.15(x10)	2.60(x1000)
N-MEM	110.85	133.084	110.026
ADD-TIME	344.66	462.73	-
ADD-MEM	110.85	133.084	-

Table 5.6: Pokec dataset results

	QTC	QTC-H	QTC-DP
C-TIME	178.54	193.28	177.34
C-SIZE	204.666	136.922	204.666
C-MEM	770.276	720.288	770.296
E-TIME	298.02	811	205.7
E-MEM	204.666	136.922	204.666
N-TIME	145.57(x10)	143.69(x10)	141.70(x10)
N-MEM	204.666	136.922	204.666
ADD-TIME	309.12	1042.76	285.55
ADD-MEM	204.667	136.923	204.667

	QTC-HDP	QTC-BFS-HDP	BLC
C-TIME	231.54	208.13	345.69
C-SIZE	136.922	127.129	70.468
C-MEM	720.272	1142.952	990.96
E-TIME	557.67	461.71	1.05
E-MEM	136.922	127.129	990.96
N-TIME	149.32(x10)	128.42(x10)	1.05(x1000)
N-MEM	136.922	127.129	990.96
ADD-TIME	766.43	727.39	-
ADD-MEM	136.922	127.129	-

Table 5.7: Twitter dataset results

	QTC	QTC-H	QTC-DP
C-TIME	8.1	9.09	8.05
C-SIZE	9.582	6.43	9.582
C-MEM	52.432	41.092	52.452
E-TIME	34.6	87.63	25.89
E-MEM	9.582	6.43	9.582
N-TIME	670.44(x1000)	692.34(x1000)	595.53(x1000)
N-MEM	9.582	6.43	9.582
ADD-TIME	39.77	106.3	45.98
ADD-MEM	9.582	6.431	9.582

	QTC-HDP	QTC-BFS-HDP	BLC
C-TIME	10.62	7.66	16.58
C-SIZE	6.43	3.411	2.663
C-MEM	41.264	57.044	55.08
E-TIME	62.92	5.19	0.05
E-MEM	6.43	3.411	55.08
N-TIME	590.99(x1000)	284.59(x1000)	0.06(x1000)
N-MEM	6.43	3.411	55.08
ADD-TIME	83.48	10.02	-
ADD-MEM	6.43	3.413	-

Although BLC outperforms QTC in terms on final compression size, the difference is comparably small in some QTC variants. While QTC-H, on average, gives the smallest compressed graph, QTC-HDP is negligibly larger (while still receiving the benifit of depth pointers). The only case where QTC-H is outperformed is when the QTC-BFS-HDP variant re-orders the nodes significantly enough. An example of this is in Table 5.7. This means that the other graphs were already in some ordering similar to BFS. A possible further improvement to QTC would be to try better ordering methods other than breadth-first-search, such as Slashburn [16].

Compression memory usages involve the loaded edge list, compressed graph, and structures used by the compression algorithm. Since all algorithms require the edge list to be loaded, we can ignore the space it uses in our comparison. Better compression algorithms obviously achieve smaller compressed graphs, which affect the C-MEM comparison. However, since we know C-SIZE, we can subtract this from C-MEM to get the true difference in memory usage. Once this is done, we can see that QTC-BFS-HDP and BLC use substantially more memory than the other algorithms. This is, of course, because they use an adjacency list. However, it is important to note that QTC-BFS-HDP only uses the adjacency list during the pre-processing stage, and does not need it for the actual compression. Thus, the maximum memory overhead for QTC-BFS-HDP is less than BLC.

5.2 Edge Query

Here we examine the results obtained from E-TIME and E-MEM. All edge queries are generated randomly and executed $1,000$ times. Once again, we mention that Huffman compression adds overhead to querying while depth pointers speed up

queries.

The QTC E-TIME performances make sense based on our understanding of the different variants we implemented. Huffman compression adds enough overhead that the only variant faster than the original QTC algorithm is QTC-DP. The only exception to this is the QTC-BFS-HDP in Table 5.7. As previously stated, this graph must have not been already ordered efficiently. Consequently, not only does the compression size improve, but the query times do as well. This makes sense, as our tree is pruned better, thus we have less to navigate through when querying.

In general, QTC is outperformed by BLC in E-TIME. This is mainly because the indexes provided at the beginning of BLC provide constant access time to that node. However, one would expect that in a graph with a high copying rate, the speed would decrease as the copy chain grows. Thus, a fully connect graph would yield the best compressed size, but would give the worst access times.

The memory used in E-MEM for all algorithms is negligible. Since QTC and its extension use forgetting query methods, E-MEM is essentially equal to C-SIZE. BLC has direct access to a node because of its indexes, and thus does not need to know anything about the preceding nodes. The only special case with BLC is when the node being queried has performed copying from another node. Since we implemented the de-copying to be iterative instead of recursive, we avoid the overhead of actual recursion. Thus E-MEM in this case is still kept minimal.

5.3 Neighbor Query

N-TIME and N-MEM are examined in this section. All neighbor queries are generated randomly, but the expensiveness of the query on QTC has forced some of the larger graphs to run a lower number of queries. The number of queries run is in parenthesis next to the query run time.

The neighbor query for BLC gives the same pattern of access times as the edge query. Again, note that in a graph with a high copy rate, the access time increases as a nodes copy chain grows.

To reiterate, these query methods were designed to use minimal overhead with respect to memory. Thus N-MEM is approximately equal to C-SIZE. However, N-MEM will obviously still be slightly larger than E-MEM since we must keep track of all the neighbors collected. It is worth noting that N-MEM for the QTC algorithms will also have even slightly more space needed since it must maintain a list of nodes in the quadtree that it is interested in.

The neighbor queries given are all out-neighbor queries. Out-neighbor queries correspond to reading an entire row on the matrix, whereas in-neighbor queries read an entire column. It can be reasonably assumed that in-neighbor queries would have similar performances.

5.4 Edge Streaming

Here we look at ADD-TIME and ADD-MEM. Like the edge queries, all add-edge operations are generated randomly and are executed $1,000$ times.

ADD-TIME and ADD-MEM are not available to BLC, since it is not a streaming compression algorithm. If there needs to be a change to the graph, the entire

compression must run again.

Since the add-edge method is essentially the same as the edge query, the ADD-TIME pattern among the QTC variants should be the same as the E-TIME. The actual difference between E-TIME and ADD-TIME are caused by the sub-method of the add-edge query that updates the nodes' colors.

Again, since the process for adding an edge is essentially the same as querying an edge, ADD-MEM is approximately E-MEM which is also approximately C-SIZE. The only additional information that ADD-MEM requires is the list of nodes is must revisit and recolor once the leaf node has been found and updated.

Chapter 6

Conclusion

In this thesis, we gave a different solution to the problem of compressing social networks in a query friendly way. Somewhat unique to our method is the fact that it does not require an adjacency matrix to compress. Additionally, our method also supports the ability to add and remove edges after compression. Finally, we demonstrated the customizability of our method by introducing several variants and comparing them all against Backlinks compression.

6.1 Future Work

Seeing as how this is a new social network compression method, there is still much room for improvement.

First, better ordering methods may yield better compressed sizes and thus better access times. We saw in 5.7 that QTC-BFS-HDP achieved better compression based on the new ordering of the nodes. This consequently improved query times since we had a smaller tree to navigate through. When looking at another ordering algorithm such as Slashburn [16], we see that is pushes all the

edges towards the upper left of the adjacency matrix. This is perfect for quadtree compression since queries perform better in the NW quadrant. This behavior is illustrated in Figure 6.1(g).

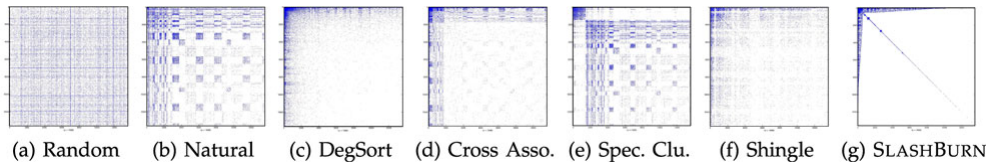


Figure 6.1: Various re-ordering methods

Also, a different scheme for labeling nodes may provide better compression. Currently, the color 2 represents a flexible color case. We may be able to arbitrarily change what the color means to something better reflecting the quadrant we are compressing. For example, it may turn out that making the color 2 represent $NW = 1, NE = 1, SW = 0, SE = 0$ may provide better compression. Such a re-definition of a color may rely heavily on the re-ordering algorithm used.

Next, we may be able to perform a partial decompression to improve query times. Instead of reading only the compressed bitstring, we could load the quadtree into a pointer tree structure. Now when querying, instead of keeping track of the next index and other information, we can simply follow the path of pointers. Note that this gives an edge access time of $log_4(n)$ and $nlog_4(n)$ for neighbor queries.

Last, custom underlying structures may help the add-edge method when it is sliding bits. We saw earlier that ADD-TIME suffers overhead from re-coloring nodes. If a quadrant must be added or removed, bits must be slid around. Thus we could use some structure more suited to contain our bitstring while supporting sliding operations. An alternate option may be to queue up a list of changes that need to be made to the bitstring after adding and edge. Then, the bitstring may

be re-written with these changes in one batch operation.

Bibliography

[1] Information technology generic coding of moving pictures and associated audio information part 3: Audio, 1995.

[2] So/iec jtc 1/sc 29/wg 1 coding of still pictures (sc 29/wg 1 structure), 2009.

[3] Information technology – coding of audio-visual objects – part 1: Systems, 2010.

[4] Micah Adler and Michael Mitzenmacher. Towards Compressing Web Graphs. In *Proceedings of the Data Compression Conference*, DCC '01, pages 203–212, Washington, DC, USA, 2001.

[5] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 595–602, New York, NY, USA, 2004. ACM.

[6] Amlan Chatterjee. Counting problems on graphs: Gpu storage and parallel computing techniques. In *Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW), 2012 IEEE 26th International*, IEEE '12, pages 804–812, Shanghai, 2012. IEEE.

[7] Joseph Cheriyan. Algorithms for parallel k-vertex connectivity and sparse certificates. In *STOC '91 Proceedings of the twenty-third annual ACM symposium on Theory of computing*, STOC '91, pages 391–401, New York, NY, USA, 1991. ACM.

[8] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On Compressing Social Networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 219–228, New York, NY, USA, 2009. ACM.

[9] CNBC. http://www.cnbc.com/id/102610670, 2015.

[10] Peter L. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, RFC Editor, May 1996.

[11] P. Elias. Universal codeword sets and representations of the integers. In *IEEE TRANSACTIONS ON INFORMATION THEORY*, pages 194–203. IEEE, 1975.

[12] Facebook Statistics. https://newsroom.fb.com/company-info/, 2014.

[13] Raphael Finkel and Jon Louis Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. In *Acta Informatica - ACTA*, ACTA '74, pages 1–9, Secaucus, NJ, USA, 1974. Springiner-Verlag New York.

[14] D.A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Proceedings of the IRE*, IRE '52, pages 1098–1101. IEEE, 1952.

[15] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[16] Yongsub Lim, U. Kang, and C. Faloutsos. SlashBurn: Graph Compression and Mining beyond Caveman Communities. *Knowledge and Data Engineering, IEEE Transactions on*, 26(12):3077–3089, Dec 2014.

[17] Hossein Maserrat and Jian Pei. Neighbor Query Friendly Compression of Social Networks. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 303–325, New York, NY, USA, 2010. Springiner-Verlag New York.

[18] Keith H. Randall, Raymie Stata, Janet L. Wiener, and Rajiv G. Wickremesinghe. The Link Database: Fast Access to Graphs of the Web. In *Proceedings of the Data Compression Conference*, DCC '02, pages 122–131, Washington, DC, USA, 2002. IEEE Computer Society.

[19] Hanan Samet. Using quadtrees to represent spatial data. In Herbert Freeman and GoffredoG. Pieroni, editors, *Computer Architectures for Spatially Distributed Data*, volume 18 of *NATO ASI Series*, pages 229–247. Springer Berlin Heidelberg, 1985.

[20] Stanford Network Analysis Project. Stanford Large Network Data Collection. https://snap.stanford.edu/data/index.html, 2011.

[21] J Zav and A Lempel. SlashBurn: Graph Compression and Mining beyond Caveman Communities. *Information Theory, IEEE Transactions on*, 23(3):337–343, May 2003.