UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

COMPRESSING AND PERFORMING ALGORITHMS ON MASSIVELY LARGE NETWORKS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

MICHAEL ANDREW NELSON
Norman, Oklahoma
2019

COMPRESSING AND PERFORMING ALGORITHMS
ON MASSIVELY LARGE NETWORKS

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Sridhar Radhakrishnan, Chair

Dr. Charles Nicholson

Dr. Changwook Kim

Dr. Qi Cheng

Dr. Christan Grant

Acknowledgements

I would like to thank Dr. Sridhar Radhakrishnan for being my committee chair, advisor, co-author, and mentor. He has also given me invaluable real-world work experience with an amazing program that helps provide accessible transportation all over Oklahoma. None of my achievements would have been possible without his guidance.

I would also like to thank Dr. Sekharan Chandra for his help in co-authoring all of my papers. He provided many insights that drove my writing through difficult sections.

Dr. Amlan Chatterjee also deserves acknowledgement not only as my co-author, but also since he was my first research mentor back when I was a Master's student and he was working on his PhD.

Next, I would like to thank Virginie Perez-Woods for being an amazingly helpful academic coordinator. She started her position at the School of Computer Science around the same time I started my undergraduate degree and has helped me every step of the way. No one cares more the students than she does.

Finally, I would like to thank the professors (Dr. Nicholson, Dr. Kim, Dr. Cheng, and Dr. Grant) on my committee for their time and advice.

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

Networks are represented as a set of nodes (vertices) and the arcs (links) connecting them. Such networks can model various real-world structures such as social networks (e.g., Facebook), information networks (e.g., citation networks), technological networks (e.g., the Internet), and biological networks (e.g., gene-phenotype network). Analysis of such structures is a heavily studied area with many applications. However, in this era of big data, we find ourselves with networks so massive that the space requirements inhibit network analysis.

Since many of these networks have nodes and arcs on the order of billions to trillions, even basic data structures such as adjacency lists could cost petabytes to zettabytes of storage. Storing these networks in secondary memory would require I/O access (i.e., disk access) during analysis, thus drastically slowing analysis time. To perform analysis efficiently on such extensive data, we either need enough main memory for the data structures and algorithms, or we need to develop compressions that require much less space while still being able to answer queries efficiently.

In this dissertation, we develop several compression techniques that succinctly represent these real-world networks while still being able to efficiently query the network (e.g., check if an arc exists between two nodes). Furthermore, since many of these networks continue to grow over time, our compression techniques also support the ability to add and remove nodes and edges directly on the compressed structure. We also provide a way to compress the data quickly without any intermediate structure, thus giving minimal memory overhead. We provide detailed analysis and prove that our compression is indeed succinct (i.e., achieves the information-theoretic lower bound). Also, we empirically show that our compression

rates outperform or are equal to existing compression algorithms on many benchmark datasets.

We also extend our technique to time-evolving networks. That is, we store the entire state of the network at each time frame. Studying time-evolving networks allows us to find patterns throughout the time that would not be available in regular, static network analysis. A succinct representation for time-evolving networks is arguably more important than static graphs, due to the extra dimension inflating the space requirements of basic data structures even more. Again, we manage to achieve succinctness while also providing fast encoding, minimal memory overhead during encoding, fast queries, and fast, direct modification. We also compare against several benchmarks and empirically show that we achieve compression rates better than or equal to the best performing benchmark for each dataset.

Finally, we also develop both static and time-evolving algorithms that run directly on our compressed structures. Using our static graph compression combined with our differential technique, we find that we can speed up matrix-vector multiplication by reusing previously computed products. We compare our results against a similar technique using the Webgraph Framework [50] [12], and we see that not only are our base query speeds faster, but we also gain a more significant speed-up from reusing products. Then, we use our time-evolving compression to solve the earliest arrival paths problem and time-evolving transitive closure. We found that not only were we the first to run such algorithms directly on compressed data, but that our technique was particularly efficient at doing so.

Chapter 1

Introduction

Graphs are structures consisting of a set of objects (vertices) and the relationships (arcs) among them. These structures can represent various real-world data such as social networks, information networks, technological networks, biological networks, and many more. While all this data has given us the opportunity to analyze and better understand these fields, the sheer size of the data has presented many challenges. For example, in this age of big data, real-world graphs have grown to contain vertices and arcs on the order of billions to trillions. Even basic structures such as adjacency lists could require petabytes to zettabytes of main memory. Moreover, the space requirements will only increase, as these graphs are ever-growing. Storing these graphs in secondary memory would require I/O access (i.e., disk access) during analysis, thus drastically slowing analysis time. In order to perform analysis efficiently on such large data, we either need enough main memory for the data structures and algorithms, or we need to develop compressions which require much less space while still being able to efficiently answer queries.

In this dissertation, we address this massive space requirement problem by

developing implicit and succinct representations of arbitrary graphs while also allowing fast access for certain queries. We then develop and run algorithms directly on the implicit and succinct representations. Our analysis will show that our compressions are indeed implicit and succinct and that algorithm run-times have been reduced to be *proportional to the size of the compressed graph.*

First, we must introduce some preliminary concepts needed to build a basis for understanding the rest of the dissertation. This includes descriptions of different types of graphs, graph representations, graph queries and algorithms, and graph compression concepts. Throughout this dissertation, we use the terms "network" and "graph" interchangeably, although a graph is technically a representation of a network.

1.1 Types of real-world graphs

In this section, we describe several types of real-world graphs.

1.1.1 Social networks

A social network is a set of people or groups of people and the relationships or interactions among them. Popular services such as Facebook, LinkedIn, and Instagram are all examples of social networks. Of all the real-world networks, social networks have the longest history of substantial quantitative study.

In the early days, social network analysis usually directly involved the participants by having them fill out questionnaires or by conducting interviews. A clever experiment performed by Milgram [93] in 1967 involved asking participants to pass a letter to one of their acquaintances in order to deliver the letter to a target individual. This experiment was the origin of the popular concept of the

"six degrees of separation," even though the phrase itself wasn't coined until later by Guare [57] in 1990. However, with the exception of a few ingenious indirect studies such as Milgram's [93] [125], most social network analysis suffered from problems of inaccuracy, subjectivity, and small sample sizes. A review of these issues has been given by Marsden [89].

Recently, due to the wide-spread use of the internet, many researchers have turned to other methods of probing social networks. Besides the above examples of social media (Facebook, etc.), we also have access to collaboration networks, which are plentiful and have relatively reliable data. A popular example of these networks is IMDB, which thoroughly records affiliations among film actors. Other examples of collaboration networks are coauthorship networks [55] [56] [58] [94] [95] (individuals are linked if they have coauthored one or more papers) and coappearance networks [1] [59] [74] [132] [136] (individuals are linked by mention in the same context). Communication records also contain reliable data from which we can construct social networks. Many studies have been performed on networks that were constructed from telephone calls [13] [91], emails [31] [64] [117], and instant messaging systems [33] [45] [111] [116].

1.1.2 Information networks

Information networks are similar to social networks, but focus more on the dissemination of information, rather than sharing experiences. They tend to be "follower" (i.e., directed) graphs rather than the "friend" model (i.e., undirected). This is not a strict requirement though, as Twitter can be considered a social network and an information network [97].

A well-studied example of such a network is the citation network, which is

a network of citations between academic papers. The structure of the citation network reflects the structure of the information stored at its vertices, hence the term "information network." Citation networks are also acyclic, meaning that papers cannot reference papers that have yet to be written. Research on citation networks goes back at least as far as 1926, with Alfred Lotka's discovery of the Law of Scientific Productivity [30], which states that the distribution of the numbers of papers written by individual scientists follows a power law. In the 1960s, Price [110] began forming networks from citation databases that had recently become available through the work of pioneers in the field of bibliometrics. Studies have continued since then with ever better resources.

The World Wide Web, which is a network of Web pages containing information linked by hyperlinks, is another important example of an information network. Note that The Web is different from the Internet, which is a physical network of computers. The Web is also cyclic, as opposed to a citation network. Our data about The Web comes from "crawls," meaning that we start at a page and we follow the hyperlinks. Thus, in a crawl that only covers a part of The Web (as all crawls currently do), pages are more likely to be discovered if they have many hyperlinks pointing to them. Since its appearance in the 1990s, it has been well-studied with notable mentions of Albert *et al.* [4] [7], Kleinbegk *et al.* [77], and Broder *et al.* [17].

1.1.3 Technological networks

Next, we have technological networks, which are man-made and typically designed to transport some commodity or resource. In this category, we refer to the networks formed by physical connections, not the networks formed by the traffic

of the commodity or resource being transported.

The most popular example of a technological network is the Internet, i.e., the physical (and wireless) connections among computers. The partial structure of the internet is usually reconstructed from large samples of point-to-point data routes. This can be done via the "traceroute" program, which reports the path its data packets took to get from the source computer A to some destination computer B. Thus, a large enough sample of paths should reveal a meaningful picture of the entire network. A few studies of Internet structure are Faloutsos *et al.* [42], Broid and Claffy [19], Chen *et al.* [25], Wu *et al.* [134], Townsend [124], and Ni *et al.* [102].

Telephone networks [85] and delivery networks [43] are other examples of technological networks. Again, we are referring to the actual networks of telephone wires and cables and networks of post-offices and parcel delivery companies, rather than the networks of who called whom.

Some other technological networks include electric power grids [5] [104] [113] [129] [130], airline routes [5] [36] [86], roads [32] [70] [67], and railways [52] [81] [118].

1.1.4 Biological networks

Finally, we see that many biological systems can be usefully represented as networks.

Classic examples of biological networks are neural networks, in which neurons are connected via synapses. Measuring the topology of real neural networks is extremely difficult, but has been done successfully in a few cases. The best known example is the reconstruction of the 282-neuron neural network of the

nematode C. Elegans by White *et al.* [131]. The network structure of the brain at larger scales than individual neurons (functional areas and pathways) has been investigated by Sporns *et al.* [119] [120] [121].

We also have networks formed by food webs, where nodes represent a species and a directed arc from A to B means that species A preys on species B. Constructing these networks is labor-intensive, but many extensive datasets and studies have become available in recent years [37] [38] [79] [108].

Another important class of biological networks is metabolic pathway networks [8] [46] [60] [69] [87] [109] [123] [128], which are representations of metabolic substrates and products with directed arcs joining them if a known metabolic reaction exists that acts on a given substrate and produces a given product.

In the next section, we formally describe how to represent networks as graphs, how to represent graphs with actual data structures, various graph properties, and common graph queries.

1.2 Graphs

A graph can be represented as a set of nodes and a set of arcs connecting the nodes, as in Figure 1.1. More formally, a graph G is a system that $G = (V, E)$, where V is the set of vertices and E is the set of arcs. An arc $(u, v) \in E$ connects node u to node v. Thus, in a social network graph, the nodes are individuals and the arcs represent the relationships among the individuals [138]. For example, in graphs such as Facebook, an arc (u, v) indicates that person u and person v are friends.

A graph can be either undirected (as in Facebook), or it can be directed (as in Twitter). In the former case, a single arc (u, v) implies that the arc (v, u) also

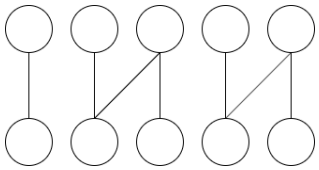

Figure 1.1: A graph example

exists. We say that this relationship is *reciprocated*. However, in directed graphs, $(u, v) \neq (v, u)$. We also see that given the number of nodes n, the numebr of arcs m, and ignoring self-loop arcs (u, u), a directed graph's *sparseness* can be calculated as $m/(n^2 - n)$, where n^2 is the total number of arcs possible and n is the number of self-loops. For undirected graphs, we ignore half the total arcs, giving $m/((n^2 - n)/2) = 2m/(n^2 - n)$.

In Table 1.1, we list some example static social, information, technological, and biological network graphs and their details. We include whether the graph is directed or not, the number of nodes ($|V|$), the number of arcs ($|E|$), and how sparse the graph is based on the directed-ness and using $(m/(n^2 - n))$ or $(2m/(n^2 - n))$ accordingly.

1.2.1 Time-evolving graphs

In Chapter 4, we will be considering time-evolving graphs. A time-evolving graph can be considered as a series of graphs (static snapshots) $G_1, G_2, ..., G_\tau$, for some lifetime τ. That is, you can see exactly how the graph has evolved over time by storing its state at each time frame.

Definition 1.1 (*static arcs*). The set of static arcs are those resulting from $\bigcup_{i=1}^{\tau} E_i$, for each $E_i \in G_i$. That is, the set of distinct arcs throughout all time frames.

Definition 1.2 (*contact*). If an arc (u, v) exists from t_i to t_j, then we provide

Social	Directed?	$\|V\|$	$\|E\|$	Sparsity
Facebook	FALSE	4039	88234	0.01
Friendster	FALSE	65608366	1806067135	8.4×10^{-7}
LiveJournal	TRUE	4847571	68993773	2.9×10^{-6}
LiveJournal(com)	FALSE	3997962	34681189	4.3×10^{-6}
NotreDame	TRUE	325729	1497134	1.4×10^{-5}
Pokec	TRUE	1632803	30622564	1.2×10^{-5}
Information				
Twitter	TRUE	81306	1768149	3.5×10^{-4}
cit-Patents	TRUE	3774768	16518948	2.3×10^{-6}
ca-AstroPh	FALSE	18772	198110	0.001
web-Google	TRUE	875713	5105039	6.7×10^{-5}
Technological				
roadNet-CA	FALSE	1965206	2766607	1.4×10^{-6}
roadNet-PA	FALSE	1088092	1541898	2.6×10^{-6}
roadNet-TX	FALSE	1379917	1921660	2.0×10^{-6}
Biological				
CC-Neuron	FALSE	1018524	24735503	4.8×10^{-5}
ChCh-Miner (drug)	FALSE	1514	48514	0.04
PP-Pathways (protein)	FALSE	21557	342353	0.001

Table 1.1: Some static graphs

two *contacts* (u, v, i) and (u, v, j) stating that the arc began at i and ended at j. This technique is used when representing a time-evolving graph as an contact list (Section 1.2.2) to avoid listing out the arc $j - i$ times.

In Table 1.2, we list some examples of time-evolving social network graphs. We describe them by their number of nodes ($|V|$), number of static arcs ($|E|$), lifetime, and number of contacts. We not only develop a compression for these graphs in Chapter 4, but we also run time-evolving algorithms directly on the compression in Chapter 5.

1.2.2 Representations

Now we review some common representations for graphs.

	|V|	|E|	Lifetime	Contacts
I-Comm.Net	10000	15940743	10001	19061571
I-Powerlaw	1000000	31979927	1001	32280816
I-Wiki-Links	22608064	564224135	414347809	731468598
I-Yahoo-Netflow	103661224	321011861	114193	955033901
G-Flickr-Days	2585570	33140018	135	33140018
soc-RedditHyperlinks	55863	858490	104976000	858490
sx-stackoverflow	2601977	36233450	239673600	63497050
email-Eu-core-temporal	986	24929	69459254	332334

Table 1.2: Time-evolving network graphs

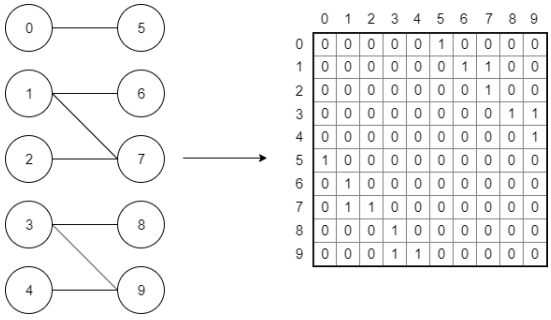

Figure 1.2: A graph as a Boolean adjacency matrix

In Figure 1.2, we show how a dots-and-lines graph can be be represented as a *Boolean adjacency matrix*. Each node is assigned a number from 0 to $n-1$, where $n = |V|$. Then for each arc $(u, v) \in E$, we set the row u, column v entry in the Boolean adjacency matrix to 1, and the rest are set to 0. This representation offers fast (i.e., constant) access time, but requires n^2 bits of space. On a billion node graph, this is 125 petabytes.

In Figure 1.3, we show the *adjacency list* representation of the same graph. Again, each node is assigned a number from 0 to $n-1$, where $n = |V|$. Then we allocate an array of size n, and for each node $u \in V$, we list each neighbor v. This list is created with pointers which each cost 64 bits. Thus, the space an

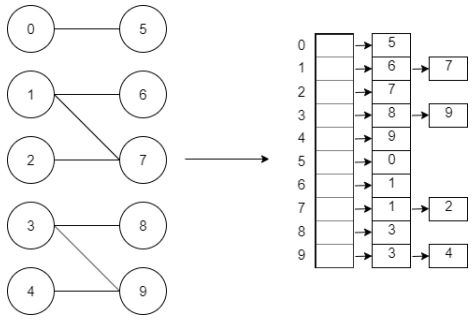

Figure 1.3: A graph as an adjacency list

	Size (bits)	arc(u,v)	N(v)
Adjacency Matrix	n^2	1	n
Adjacency List	$2m \times 64 + 2m \log_2(n) + n \log_2(n)$	$O(\delta(G))$	$O(\delta(G))$
arc List	$2m \log_2(n)$	$O(m)$	$O(m)$

Table 1.3: Common graph representations (assuming 64-bit)

adjacency list representation occupies is $2m \times 64 + 2m \log_2(n) + n \log_2(n)$, where $m = |E|$. When $n = 10^{10}$ and $m = 10^{15}$ ($2 \times 10^{-5}\%$ sparsity) this is 24.3 petabytes. This structure offers better space requirement than the Boolean adjacency matrix when n is large and m is small, i.e., the graph is *sparse*. However, it is slightly slower to answer queries with a worst case time complexity of $O(\delta)$, where δ is the size of the largest set of neighbors (i.e., maximum degree) of the graph.

A graph can also be represented as an *arc list*. This representation simply lists out each arc (u, v). Thus, it requires $m \log_2(n)$ space with a worst case query time of $O(m)$. When downloading static graphs from the internet, they will usually be in this form.

Table 1.3 three types of static graph representations with their complexities.

Time-evolving graphs

If a static graph can be thought of as a 2D Boolean adjacency matrix, then a time-evolving graph can be thought of as a 3D Boolean adjacency matrix, also called a *presence matrix* [47]. For each arc $(u, v, t) \in E$, we set the corresponding presence matrix entry to 1, and the rest we set to 0. This representation requires n^3 bits of space. Because of this, we see that basic data structures are not easy to adapt to time-evolving graphs. For another example, one could consider providing a set of adjacency lists for the graph at each time frame, but if an arc exists in many time frames, then we will have to create many redundant arcs. Because of the similarity between adjacency time frames, many works have been done using differential compression.

Additionally, we may represent a time-evolving graph as a *contact list*. This representation consists of a set of triplets (u, v, t), in which a triplet represents a change in an arc (u, v) at a time t, where u is the source and v is the destination. If an arc already exists at a time t_i and then appears again at t_j, where $i < j$, then the arc is deactivated. Thus, if an arc has an odd number of occurrences at a given time, it is activated. It is deactivated otherwise.

1.2.3 Properties

Static graphs

Many types of graphs tend to exhibit a certain set of properties. The graphs we examine are usually:

- Simple - The arcs are single, unweighted, undirected, and have no self-loops.
- Labeled - The users are anonymized and represented as integers.

- Large - There are nodes in tens of millions and arcs in the billions.
- Sparse - The sparsity is almost always below 0.01 ($m/(n^2-n)$) or ($2m/(n^2-n)$).
- Streaming - Nodes and arcs constantly being added.
- Queryable - Many algorithms are run on these graphs in order to better understand their structure.

Time-evolving graphs

Additionally, time-evolving graphs can be:

- Interval - Arcs exist in time intervals $[t_i, t_j)$, where $i < j$.
- Incremental - Once an arc has been activated, it remains so until the end of the lifetime. An arc may exist in several intervals and intervals may not overlap.
- Point - Arcs only appear at a single time frame, i.e., $[t_i, t_{i+1})$. Arcs may appear multiple times.

Time-evolving graphs also have a time granularity (e.g., second, hour, day, etc.).

1.2.4 Queries

Static graphs

Common queries on graphs are:

- Arc existence $(u, v) \in E$? - is there an arc from u to v?

- List neighbors $N(v)$ - list all the out-neighbors of node v.
- Reachability $R(u, v)$? - is there a path from u to v?
- Community queries (complete bipartite graphs) - a suite of operations such as finding all communities of a fixed size, listing all maximal communities, and enumerating all communities.
- Clique queries (complete subgraphs) - supports operations similar to community queries.
- Graph pattern matching - find all subgraphs that conform to some graph pattern.

Time-evolving graphs

Common queries on time-evolving graphs are:

- DirectNeighbors(u,t) - returns active adjacent neighbors of u at time t.
- ReverseNeighbors(v,t) - returns active reverse neighbors of v at time t.
- arc((u,v),t) - returns true if arc (u,v) is active at time t, false otherwise.
- arcNext((u,v),t) - returns the instant of the next activation of (u, v) after t, or t if it is active; otherwise returns ∞.
- Snapshot(t) - returns all active arcs at time t.
- Activatedarcs(t) - returns all arcs that were activated at time point t.
- Deactivatedarcs(t) - returns all arcs that were deactivated at time point t.
- Changedarcs(t) - returns all arcs that were activated or deactivated at time point t.

1.3 Compression

Given some data X of length $|X| = n$, a compression is a function $C : X \to X'$ such that $|C(X)| = n' \leq n$. In other words, compressing some data *reduces* the space that data occupies. Compression can be lossy or lossless. For the purposes of this dissertation, we only focus on lossless compression.

1.3.1 Lossless compression

A lossless compression is one such that no information is lost, and thus the data can be perfectly recovered. More formally, given a compression function C, a decompression function D, and some data X, a lossless compression is such that $D(C(X)) = X$. Such compressions are usually seen in applications such as: images (PNG), audio files (FLAC), and general compressions (gzip, zip, 7zip).

A natural question is: "how much can we losslessly compress data?" While there is a *theoretical limit* on every compression, it is generally uncomputable. Obviously, there is some limit to lossless compressibility. All we are doing is representing a binary string of size n with a binary string of size $m < n$. Since there are less strings of size m than those of size n, clearly we can't *uniquely* compress every possible string of size n.

Kolmogorov complexity - the length of the shortest program to compress a string.

Finding this measure is undecidable in general because it requires knowing whether a program produces a given output. This means that having an oracle that told you the Kolmogorov complexity of an arbitrary string would be enough to solve the halting problem. We demonstrate this informally using Algorithm 1.1.

Algorithm 1.1: GenerateComplexString()

```
1 for i = 1 to ∞ do
2     foreach string s of length exactly i do
3         if KolmogorovComplexity(s) ≥ 6 × 10^9 then
4             return s;
```

Let's assume that Algorithm 1.1 requires 5001203792 bits of space; 5×10^9 bits for the language, 1.2×10^6 bits for the KolmogorovComplexity program, and 3792 bits for the GenerateComplexString program. GenerateComplexString is designed to generate all strings (starting from the shortest) until it returns one s that has a Kolmogorov complexity of at least 6×10^9 bits. However, this is not possible. The Kolmogorov complexity is equal to the length of the minimal description of s. Yet our program is only 5001203792 bits. Since it is not possible to calculate the minimum description of an arbitrary input, it is also not possible to calculate its optimal compression ratio.

Information-theoretic lower bound - we can still theoretically calculate the storage lower bounds when considering a class of graphs. In particular, when considering an arbitrary (undirected) graph with n vertices and m arcs, the number of such graphs is $Z = \binom{\binom{n}{2}}{m}$, the storage lower bound is $\lceil \log_2 Z \rceil$.

1.3.2 Queryable compression

Typically, general compressions, such as gzip, cannot read the data while it is compressed. The data must be completely decompressed before being able to be read. Compressions that are designed to support compressed reads are usually called *compressed structures*, as opposed to *compression schemes*.

More formally, a *query preserving compression* is designed to compress data

relative to a class $\mathcal{Q}$ of queries. Thus, given any query $Q \in \mathcal{Q}$, some data X, and the compressed data X', we have $Q(X) = Q(X')$. This typically does not reduce the complexity of the query, but it does reduce the space required to run the query.

1.3.3 Incremental compression

In many real-life cases, the data we are dealing with changes frequently. When compressing such data, it is beneficial to have an *incremental compressed structure*. In other words, we would like to have a compression that can *directly edit* the compressed data *without having to decompress the data*.

More formally, given a compression function $C : X \to X'$, some data X, the compressed data X', and a list of changes ΔX, an *incremental compression* directly computes changes $\Delta X'$ to X' such that $X' \bigoplus \Delta X' = C(X \bigoplus \Delta X)$.

1.4 Graph compression

First, we ask: "Why not just use general compressions (gzip)?" The answer is two-fold:

- Graph compressions can be designed to take advantage of *structural properties of the graph* that general compressions are ignorant of.
- As mentioned before, general compressions are not typically queryable.

In Figure 1.4, a general compression would end up compressing the full 42 arc list. A graph compression could recognize this is a complete graph, and therefore represent it with a single integer, 7.

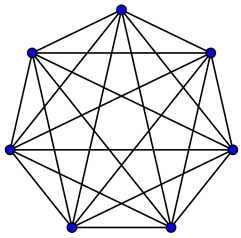

Figure 1.4: A complete graph of size 7

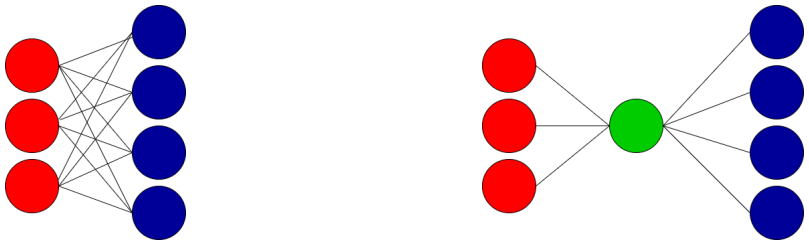

Figure 1.5: Reducing a complete bipartite graph

In Figure 1.5, we have changed the representation from needing $m + n$ nodes and mn arcs, to $m + n + 1$ nodes and $m + n$ arcs. Obviously, we could instead represent this graph with two integers, m and n, but we would lose links entering and leaving this subgraph.

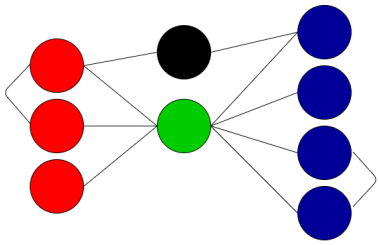

Figure 1.6: A reduced complete bipartite subgraph

In Figure 1.6, notice that this reduction also preserves links between nodes in the same partition and also link from outside the subgraph (black). Clearly, no links from outside the subgraph will attach to the virtual node (green).

1.4.1 Exploitable structural properties

The following structures and properties are some of those found throughout real-life network graphs and can be exploited to improve compression:

- Cliques (complete subgraphs) - Clearly, networks like social networks will have several groups of people where everyone is friends with one another. Such structures can be represented with a single integer.

- Communities (complete bipartite graphs) - Similar to cliques, these structures can be found in social networks. They can be reduced either by using only two integers, or by introducing a virtual node and redirecting all the arcs to this new node.

- Similarity - This is a property that many nodes will end up having similar sets of neighbors. This presents an opportunity to encode nodes by having them reference other nodes. This is prevalent throughout many real-world networks.

- Locality - This property is exposed when the graph has undergone an appropriate node reordering. The resulting labels of neighbors of a node will tend to be close to each other in the ordering. This property is the basis of gap encoding.

1.4.2 Node reordering

Node reordering algorithms are meant to better expose patterns and redundancies in the graph. This is illustrated in Figure 1.7. Node reordering is often combined with existing compression algorithms. Many graphs found online will be in the common lexicographical BFS ordering.

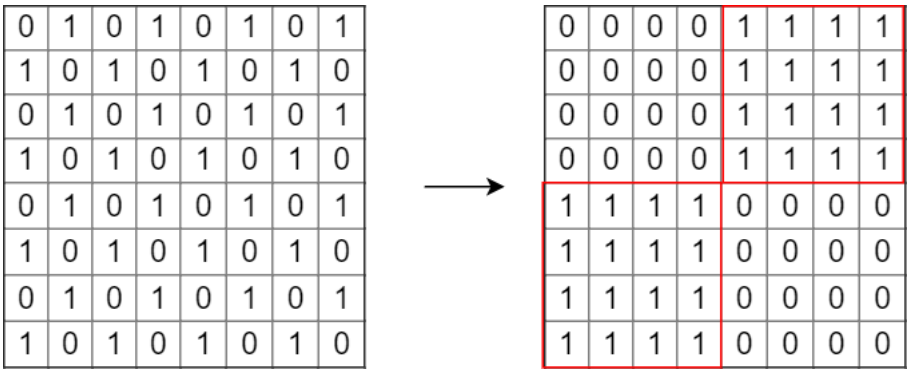

Figure 1.7: An unordered adjacency matrix (left); the same, reordered adjacency matrix (right)

1.4.3 Compressed Sparse Row

Although used since the mid-1960s, the first complete description of the compressed sparse row technique (CSR) was given by Tinney and Walker in 1967 [21]. Denoting the number of nodes with n and the number of arcs with m, this format represents an $m \times n$ matrix M by three one-dimensional arrays that respectively contain nonzero values, the extents of rows, and column indices. This format allows fast row access and matrix-vector multiplications (Mx). The details of the three arrays (A, IA, and JA) are as follows:

Let NNZ denote the number of nonzero entries in M.

- A is of length NNZ and holds all the nonzero entries of M in left-to-right, top-to-bottom ("row-major") order.
- IA is of length m + 1 and is defined by the following recursive definition:
  - $IA[0] = 0$
  - $IA[i] = IA[i1]+$ (number of nonzero elements on the $(i-1)$-th row in the original matrix)

- JA is of length NNZ and contains the column index in M of each element of A.

For example, given the matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 9 \\ 0 & 4 & 0 & 0 \\ 8 & 0 & 0 & 0 \end{bmatrix}$$

we have A = [2 9 4 8], IA = [0 0 2 3 4], and JA = [2 3 1 0].

1.5 Prior work

In this section, we discuss various prior works which succinctly represent graphs with no or little assumption about the structure of the graph itself. There are many other techniques which require some assumptions (such as planarity), but they are outside the scope of this dissertation. First, we must define different terms that relate how "close" a data structure is to the information-theoretic lower bound, in terms of space.

Suppose that Z is the information-theoretical optimal number of bits needed to store some data. A representation of this data is called:

- *implicit* if it requires $Z + O(1)$ bits of space
- *succinct* if it requires $Z + o(Z)$ bits of space
- *compact* if it requires $O(Z)$ bits of space

Reference*	Size (bits)	Labels	arcs	Fast access	Fast encoding	Graph restrictions	Scheme type
Turan [126]	$\binom{n}{2} - \frac{1}{8} n \log_2(n) + O(n)$	no	unsp.	no	no	-	-succinct
Naor [98]	$\binom{n}{2} - n \log_2(n) + O(n)$	no	unsp.	yes	yes	arbitrary adj. matrix	-succinct
Raman [112]	$\lceil \log_2 \binom{n^2}{m} \rceil + o(m)$	no	dir.	yes	no	-	succinct
Farzan [44]	$\log_2 \binom{n^2}{m} + o(\log_2 \binom{n^2}{m})$	yes	dir.	yes	no	$m > \frac{n^2}{\log_2^{1/3} n}$	succinct
Farzan [44]	$(1+\epsilon) \log_2 \binom{n^2}{m}$	yes	dir.	yes	no	$\frac{n^2}{\log_2^{1/3} n} \geq m > \frac{n}{2}$	succinct
Farzan [44]	$\log_2 \binom{n^2}{m} + \epsilon m \log_2 m$	yes	dir.	yes	no	$\epsilon > 0, m \leq \frac{n}{2}$	succinct
Farzan [44]	$\log_2 \binom{n^2/2}{m}$	yes	undir.	yes	no	$\frac{n^2}{4} > m > \frac{n^2}{\log_2^{1/3} n}$	succinct
Farzan [44]	$(1+\epsilon) \log_2 \binom{n^2/2}{m}$	yes	undir.	yes	no	$\frac{n^2}{\log_2^{1/3} n} \geq m > \frac{n}{2}$	succinct
Farzan [44]	$\log_2 \binom{n^2/2}{m} + \epsilon m log_2 m$	yes	undir.	yes	no	$\epsilon > 0, m \leq \frac{n}{2}$ $k = O(n), k = m - n + 1$	succinct
Fischer [48]	$(2n+m) \log_2 3 + h \log_2 n + k \log_2 h$ $+o(m + k \log_2 h) + O(\log_2 \log_2 n)$	unsp.	dir.	yes	no	$h = \lvert K \rvert \leq k$ $K = \{v \epsilon V : \lvert N_{in,v} > 1\}$	succinct

Table 1.4: Succinct graph representations for arbitrary graphs

In Table 1.4, we list several succinct data structures for graphs with little or no restrictions (arbitrary). We include the name of the first author (and a reference), the size complexity in bits, whether the representation is vertex or arc labeled, if the representation provides fast access or encoding, the graph's restrictions, and the scheme type (i.e., implicit, succinct, compact).

1.6 Conclusion

We have covered what a network is and how to represent one. We have also defined some compression concepts and how to apply them to graphs. In this dissertation we shall develop novel, implicit and succinct, queryable, incremental, arbitrary graph compressions based on binary trees. Each subsequent chapter tackles different challenges that emerge from massive space requirements and gives discussion on related work, detailed algorithms, analysis, and empirical study.

The rest of the dissertation is outlined as follows:

- We begin by developing a queryable, incremental, arbitrary, static graph compression that reduces the graph's size to the theoretical minimum (Chapter 2).
- We continue this work by adapting the compression to speed up matrix-vector multiplication by reusing products via differential compression (Chapter 3).
- Next, we move our compression to the field of time-evolving network graphs, again reaching the theoretical minimum (Chapter 4).

- We adapt our time-evolving graph compression to run algorithms such as the earliest arrival paths problem, and our own novel problem definition of time-evolving transitive closure (Chapter 5).

- Finally, we conclude the dissertation (Chapter 6).

Chapter 2

Arrays of compressed binary trees

In this chapter, we propose to build a compressed data structure that has a compressed binary tree corresponding to each row of the adjacency matrix. We do not explicitly construct the adjacency matrix, and our algorithms take the arc list (even gzipped) representation as input for its construction. This allows for minimal memory overhead. Our compressed structure allows for faster arc and neighborhood queries, and also it allows arcs to be added and removed directly from our compressed structure (streaming operations). We have evaluated our compression technique and compared the resulting size with existing compression algorithms along with many other parameters. For the purposes of our experiments for this chapter, we focus on social networks and the Webgraph framework, which is the current state-of-the-art social network compression. We have used the publicly available network data sets such as Friendster, LiveJournal, Pokec, Twitter, and others. We show that our improvements allow the structure to perform better than the current state-of-the-art technique in terms of compression

size and other key metrics.

2.1 Introduction

A network can be represented as a graph. Most social networks like Facebook are undirected, meaning that the relationship is mutual. We use the terminology reciprocity or reciprocal to describe the relationship that flows both ways (viz., undirected). In contrast, a network like Twitter is directional, as meant by their concept of 'following'. Clearly, we can see that knowledge learned from these graphs has been used in the industry to better coordinate events, suggest friends, advertise, and recommend games.

Most real-world networks are ever growing. For example, from Q2 2017 to Q2 2018, the number of daily active Facebook users grew from 2.0 billion to 2.23 billion [29]. Many networks comprise not only of individuals but also of business entities and hence the size of such graphs can pose a high level of difficulty for analysis.

Many different queries may be run on networks. When developing a queryable compression technique, the compressed structure is usually designed to be efficient with a specific set of queries [73]. The most popular of these are typically community operations and the reachability query. When building the algorithms to answer these queries, neighborhood enumeration and arc existence operations are put to heavy use, especially in problems such as network pattern mining and friend suggestion.

Consider a snapshot of Friendster database with $n = 65608366$ nodes and $m = 1806067135$ arcs. Using a Boolean adjacency matrix representation, we get a size of 65608366^2 bits = 538TB. Assuming 64-bit pointers and using the

equation $2m \times 64 + 2m \log_2(n) + n \log_2(n)$, even an adjacency list representation would consume about 41GB. Clearly, most computers will not have enough main memory to process such large graphs. As such, it must make access calls to disk, thereby incurring a high time penalty. Given this, our desire is to compress the graph to a size that can not only fit in the main memory but also provide mechanisms to perform neighbor and arc queries on the compressed structure itself.

Most raw, uncompressed graphs from various resources are represented as plain text files. These files are merely the graph in arc list form. That is, each line consists of two numbers, u and v, separated by a space. A common requirement for most compression algorithms is an intermediate structure, such as an adjacency list, that is built from this arc list and used to efficiently build a final compressed structure [88]. Since we can incrementally build our compression, we do not require such an intermediate structure.

For obvious reasons, the original arc list text files are stored with common compression programs such as gzip. Preprocessing large graphs like our Friendster graph requires at least 31GB of memory, and 7GB if the arc list was compressed with gzip [Table 5.1]. Here, we also present a method of compressing the graph directly from its gzipped arc list format. Since our compression scheme uses no intermediate structure when compressing, this ensures minimal memory overhead. Our compression technique is based on indexed arrays of compressed binary trees [100], with enhancements provided by preorder traversal. The binary trees will be responsible for compressing a node's arcs, and the indexes will provide quick access to those nodes in the compressed graph. Our motivation for using binary trees springs from our earlier research with the quadtree data structure [99]. The quadtree structure could be thought of as compressing the graph's entire 2D adja-

cency matrix representation, whereas binary tree representation compresses each individual row of the matrix. Our contributions can be summed up as:

- We use a novel indexed array of compressed binary trees structure that can incrementally construct the compressed file as nodes and arcs are added and removed (*streaming graph model*). This representation is used in conjunction with preorder traversal for efficient query execution. Existing algorithms require the entire adjacency list or adjacency matrix before the construction of the compressed output can begin.

- We improve our compression ratio by better encoding groups of ones, as well as branches that only contain one arc.

- We provide improved encoding for directed graphs by only compressing the upper triangular matrix and including an extra bit per arc to represent reciprocity.

- We improve our compression ratio by more efficiently encoding our tree structure once we reach the bottom levels.

- We provide a method for compressing the graph directly from its gzipped arc list form.

- We present algorithms to execute arc and neighbor queries that directly operate on the compressed file.

- We present a special in-memory data structure to further improve run-times for streaming data.

- We present detailed experimental results using the SNAP database [122] to obtain performance benchmarks on several networks, including a compar-

ison of BFS vs preorder traversal. It may be noted that SNAP datasets in the form of arc lists are directly read in to memory for compression, avoiding intermediate memory usage.

- We believe we are the first to provide a compression technique for streaming graphs that efficiently support widely-used operations.

The rest of the chapter is organized as follows. In Section 2.2, we review related works which are based on the availability of the entire adjacency matrix or list for compression to complete. In Section 2.3, we formally define our queries and review common network graph terminology. We then present the details of binary tree compression in Section 2.4. We report empirical results in Section 2.5. Finally, we conclude the chapter in Section 2.6.

2.2 Related work

There are several existing works that present compression algorithms that are similar in idea to the ones we present here. In general, our compression concept is most similar to k^2-trees [16]. However, k^2-trees only compress the Boolean adjacency matrix as a whole, whereas we use compressed binary trees to represent each row of the adjacency matrix. We also use different encoding schemes to better compress runs of ones and branches that only contain one arc. We do not compare against k^2-trees since Backlinks compression (BLC) outperforms them in terms of directed static graphs.

Adler and Mitzenmacher [2] introduced a web graph compression scheme by finding nodes with similar sets of neighbors. Randall et al. [114] were the first to use the lexicographical ordering of the URL's on a web page to compress a graph.

Their method exploits the fact that many pages on a common host have similar sets of neighbors. Boldi and Vigna [12] also exploited inherent properties of web graphs for compression. 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 arcs. In order to further improve compression, Boldi et al. [12] developed new orderings that combine host information and Gray/lexicographic orderings.

In 2009, Chierichetti et al. [27] modified the Boldi and Vigna compression method [12] 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 arcs. This method is called the Backlinks Compression scheme and serves as the key benchmark for our empirical study.

In 2014, Liakos et al. [83] also improved Boldi-Vigna's compression ratio and access times by separately compressing the dense main diagonal stripe of the graph's adjacency matrix.

In 2010, Maserrat and Pei [92] 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. [84] invented Slashburn, a new ordering method to execute on the graph before a general 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 spokes connected only by the hubs. After executing their ordering function, they use a block-wise encoding method (such as gzip) for the actual compression. The novel technique described in Slashburn [84] 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 [72].

Our work here targets graphs that are streamed in and therefore does not lend itself to storing the graph in an adjacency matrix or list and renumbering vertices to achieve a good compression (measured as number of bits used per arc on the average).

The work presented here is an enhanced version of our work described in [100] which was inspired by our previous work involving quadtree compression [99] where we treated the Boolean adjacency matrix as a 2D space to be compressed by a quadtree. The resulting tree was outputted in BFS order as a string of bits. Our transition from quadtrees to an indexed array of binary trees is based on the principle that if a quadtree can compress the entire $n \times n$ Boolean adjacency matrix as a 2D image, then n binary trees can each compress a single row of the matrix. This transformation in conjunction with (i) using preorder traversal, (ii) providing an in-memory structure, and (iii) incorporating a key modification to better encode reciprocal arcs, and (iv) a massive improvement to the general encoding scheme, we show that we are able to improve the query and streaming times.

2.3 Preliminaries

In this section, we describe some network characteristics, common operations, and standard compression techniques.

2.3.1 Real-world networks

Here we shall describe the general characteristics that are typical in these types of graphs.

Sparseness

Real-world networks are very sparse. For example, our LiveJournal graph contains about 4.8 million nodes and about 69 million arcs. The total number of possible arcs is $(4.8 \times 10^6)^2 - 4.8 \times 10^6 = 2.3 \times 10^{13}$. Therefore we only have $(100 \times (6.9 \times 10^6)/(2.3 \times 10^{13}) = 3 \times 10^{-5}\%$ of the total arcs possible. If we were to draw the Boolean adjacency matrix for this graph, it would be mostly zeros.

Similarity

This property states that nodes close to each other in a lexicographical ordering have similar sets of neighbors. For example, in a high school with 300 students, the nodes might be numbered $0 - 299$ in a lexicographical ordering. Since the students are likely to be friends with each other, many of the nodes will have similar sets of neighbors.

Locality

With locality, a node tends to be connected to other nodes closer to its position in the lexicographical ordering. In other words, node 0 is more likely to be connected to node 100 rather than node 1000000. Locality is tied closely with similarity, but the distinction is important.

2.3.2 Real-world network compression

Next, we outline common compression exploits for the characteristics described in Section 2.3.1. Our new compression technique doesn't make explicit use of any of these exploits, but our benchmark compression and many other algorithms do. Regardless, these techniques are standard background knowledge when understanding any SN compression.

Gap encoding

This technique takes advantage of the locality property. Even though the label numbers for a node's neighbors may be large, the numbers should be close to each other. Therefore instead of actually encoding the large label numbers, we encode the differences (gaps) between them. For example, if we have nodes labeled 1000000 and 1000001, instead of encoding these two large numbers, we can encode the difference between them, 1.

Removing neighbor redundancies

Next, we describe how to exploit the similarity property. Since this property states that nodes close to each other in the ordering share common neighbors, we can see that those similar neighbors present redundant information that can be optimized out. When encoding a node n, we check k of the previously encoded nodes for common neighbors. If a sufficiently similar node x is found, node n is compressed based on x.

It is important to realize the negative effect this technique has on access operators. Since we are linking nodes to previous nodes, we can form a chain of references. Therefore, when querying a node, we may end up having to query all

the nodes in the reference chain. This is also why streaming is usually impractical on such compression schemes. One arc update may cause a large chain of updates.

Node reordering

If the ordering of the nodes in the graph is random, then the previous two exploits will fail. Therefore, a common bit of preprocessing is to reorder the nodes in the input graph based on some ordering scheme. The standard is a BFS lexicographical reordering. Recently, there is also Slashburn - a social network specific, node reordering technique [84].

2.3.3 Real-world network operations

Lastly, we review the common operations performed on social network graphs.

Arc existence queries

The most basic operation is checking whether or not an arc exists between two nodes. Formally, given a graph $G = (V, E)$ and an arc (u, v), determine if $(u, v) \in E$.

Node neighbor queries

As previously stated, neighbor queries are arguably the most important for real-world networks. Given a graph $G = (V, E)$ and a node u, list all the out-neighbors of u.

Arc addition/removal

This operation is available only in streaming graph compressions, otherwise known as incremental graph compressions. These compressions allow arcs and nodes to be efficiently added and removed without having to completely re-compress the graph.

2.4 Indexed array of compressed binary trees

Next, we move on to the description of our compression technique. For every input graph $G = (V, E)$, we assume that E is sorted and nodes in V are labeled from 0 to $|V| - 1$. We also assume that $|V|$ is rounded up to the next power of two.

This assumption is only necessary for a direct construction, rather than an incremental one.

2.4.1 Node-centric, indexed structures

As previously mentioned, most real-world networks and their query operations are node-centric. This indicates that our compression method must also be node-centric. That is, our compression technique will compress one node at a time, in order. Compressing a node consists of compactly representing all the node's neighbors in a lossless way. A node is compressed into a string of bits that is then appended to the final bit-string.

A consequence of being node-centric is that when querying for an arc (u, v), we must start at the beginning of the compressed graph and read sequentially up to node u. A workaround for this is to include an array of indexes that point to

the positions of every node in the compressed graph. This is a common practice and sacrifices minimal space for a great speed increase on queries [83]. In our case, the space requirement is $O(nd\log(n))$, yet we receive a neighbor query time complexity of $O(d\log(n))$, where d is the degree of the graph.

2.4.2 Compressed binary trees

A binary tree is a tree in which every non-leaf node has two children. A full binary tree of depth k has 2^k leaf nodes. Let $n = |V|$. When a graph $G = (V, E)$, as in Figure 2.1 (a), is represented as an $n \times n$ Boolean adjacency matrix, as in Figure 2.1 (b), each row i of the matrix represents all the neighbors of node i. Knowing this, we can represent a matrix row of width n with a binary tree of depth $\log_2(n)$. We do this for each node's adjacency row as each time frame. This process is summarized in Figure 2.1 (c).

For a compressed binary tree, each node in the tree is encoded with one bit each. A node is set to true if it is a non-leaf node, or if it is a leaf node corresponding to an arc. All nodes marked false are leaf nodes. This encoding scheme prunes off sections of the matrix row that are empty, while the path to an arc must travel to the bottom of the tree. Examples of compressed binary trees are illustrated in Figure 2.1 (d). Nodes have brackets indicating which indexes of the adjacency matrix row the nodes range over.

2.4.3 Improved encoding

Next, we show that this encoding scheme can be even further improved. Currently, our tree efficiently compresses runs of zeros, but must expand to the maximum depth whenever an arc is active (i.e., there is a one). Now, notice that

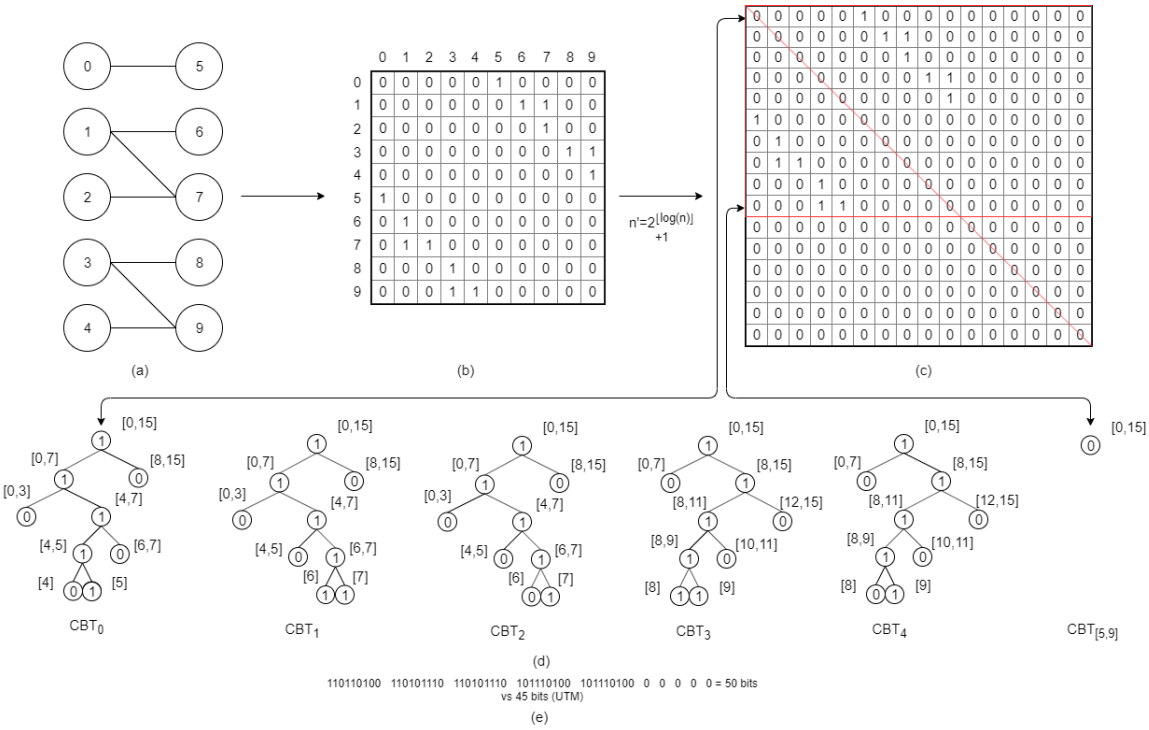

Figure 2.1: A graph represented as a series of compressed binary trees

when a non-leaf node is labeled with a one, it should normally never be followed with two zeros [23]. We can exploit this fact to better compress our ones. We can use this to not only compress consecutive arcs, but also branches that only contain one arc.

We show how to encode these two cases in Figure 2.2. If a non-leaf node is followed by two zeros, then an additional bit must follow. If the bit is zero, then the current branch contains only ones, as in indexes [0,15] of Figure 2.2. If the bit is one, then the branch contains a single arc, as in index [28] of Figure 2.2. After the bit, we then provide a relative binary path to the arc, given in Algorithm 2.1.

In Algorithm 2.1, we are given "begin" and "end" which represent the range of our current node, along with the target index "j." We calculate the relative depth in line 2, which will be the length of our returned bit-string. In lines 5

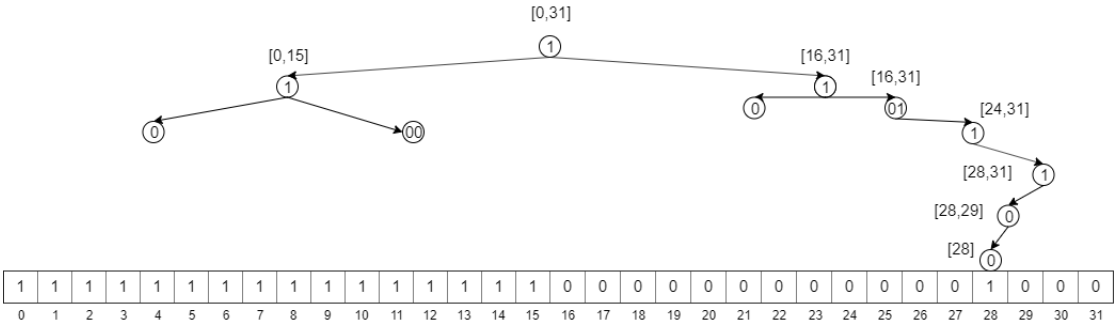

Figure 2.2: Improved compression of an adjacency row (n=32) containing 17 arcs with only 13 bits

through 12, we loop through each level of the remaining depth, and append a one or a zero, depending on which child the path should navigate to. We return the path's bit-string in line 13. Next, we describe Algorithm 2.2, which encodes a node given its neighbor list.

In Algorithm 2.2, we are given as input a node's neighbors a a list of positions indicating the indexes of all the ones in the Boolean adjacency matrix row. If a target is already a neighbor of the node, it is to be removed. We use the visitor pattern and some LINQ notation [9] for ease of reading. We start by traversing through the CBT representing the node's current neighbors. On line 6, we get the node's neighbors as a list of the positions of the ones in the row of the Boolean adjacency matrix. Line 10 makes use of our new encoding scheme by appending '000', indicating that this entire branch is full of ones. Lines 12 through 16 also uses a new encoding scheme by appending '001', followed by the relative binary path to the target position of the only one in this branch. If there were no ones for this branch, then we simply append a zero, as in line 18. We return the differential CBT as a bit-string in preorder traversal in line 21.

We further improve our encoding by noticing that as we reach the bottom of our compressed tree where a node only spans two indexes of the row, it is no longer

Algorithm 2.1: A relative binary path

```
   Input: int begin, int end, int j
   Output: The relative binary path as a bit-string
1  begin
2      BitString s;
3      depth = ⌈log_2 (end − begin)⌉;
4      s.Initialize(depth);
5      for i = 0; i < depth; i++ do
6          mid = ⌈(begin + end)/2⌉;
7          if begin ≤ j < begin + mid then
8              s.AppendBit(0);
9              end − = mid;
10         else
11             s.AppendBit(1);
12             begin + = mid;
13     return s;
```

efficient to maintain our tree structure. We instead directly replace this section of the tree with the two indexes (bits) they are supposed to represent. Both approaches have a best case of four bits when dealing with two nodes that span four indexes together. However, the old approach requires six bits in its worst case, compared to the new approach's requirement of four bits. This change is illustrated in Figure 2.3. However, for simplicity's sake in our algorithms and analyses, we maintain our notion that the last level keeps to the original compressed binary tree structure.

When a compressed binary tree is output to a bit-string, we have many choices of how to traverse the tree, e.g., BFS or preorder traversal. In our previous work [100], we traversed the graph in the BFS order. However, it can be seen that a preorder traversal would provide a better average time complexity when querying, since the leaf nodes are not all at the end of the bit-string. Such a traversal would also improve streaming operations, since all the bits related to an arc are grouped

Algorithm 2.2: Improved Binary Tree Compression

```
   Input: List<int> ones
   Output: CBT'_{i+1} as a bit-string
 1 begin
 2     BitString s;
 3     Node node = cbt.Root;
 4     Visitor vtr = PreOrderTraversal(node);
 5     while !vtr.End() do
 6         nodeTargets = ones.Where(x => node.Spans(x));
 7         if nodeTargets.Count() > 0 then
 8             s.AppendBit(1);
 9             if node.IsFull(nodeTargets) then
10                 s.AppendBitString('000');
11                 vtr.Ignore(node);
12             else if nodeTargets.Count() == 1 then
13                 s.AppendBitString('001');
14                 path = RelativeBinaryPath(node.begin, node.end, target);
15                 s.AppendBitString(path);
16                 vtr.Ignore(node);
17         else
18             s.AppendBit(0);
19             vtr.Ignore(node);
20         node = vtr.Next();
21     return s;
```

next to each other, as opposed to being scattered throughout the bit-string when applying BFS. Therefore, using preorder traversal, the bit-string for Figure 2.3 would be 1100010011100. For comparison, the corresponding BFS ordering would be 1110000011100.

In Figure 2.2, we have an adjacency row of size $n = 32$ containing 17 arcs and compressed to only 13 bits using our improved encoding scheme. This example demonstrates the case where a node's left child contains all ones, and the right child only contains a single one. In the latter case, it means that the current branch leads to a single arc to which we then provide a direct binary path,

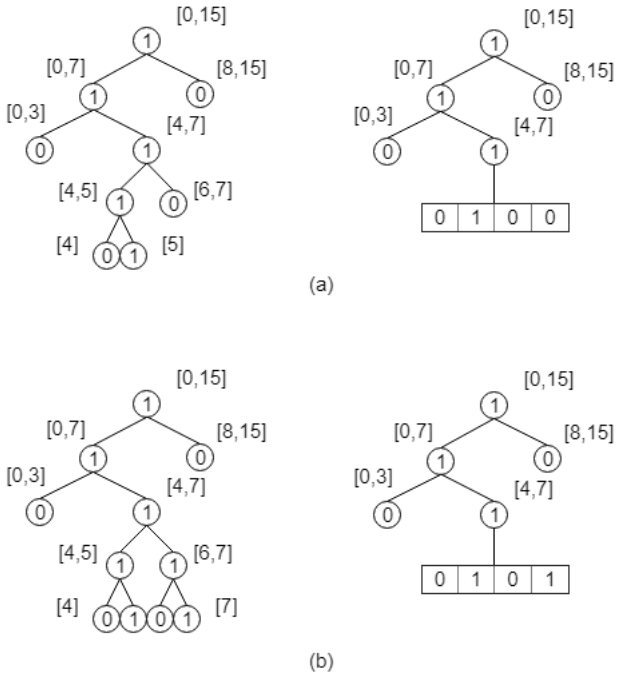

Figure 2.3: The improved compressed tree encoding comparisons in the (a) best case and (b) worst case

relative to the current branch. That is, while the example's change is encoded at the right child of the root of the tree ($d = 1$), it could occur at any depth $d < \log_2(n) - 3$ with a saving of $\log_2(n) - 2d - d - 3 = \log_2(n) - d - 3$ bits.

2.4.4 Analysis

Suppose that a arbitrary graph has n nodes and m arcs. Each arc is stored as a 1 in the Boolean adjacency matrix, with the rest of the entries being 0.

Lemma 2.1: Assuming a graph with $n = 2^k$ nodes where $k > 0$, the binary tree corresponding to a node can have a depth of at most k.

Proof: At each depth, the range that a binary tree's node can span is halved. Thus, the maximum depth is $\log_2(2^k) = k$. ■

Lemma 2.2: Given a directed graph with $n = 2^k$ and $k > 0$, a node with a single arc can be represented with $k + 4$ bits.

Proof: Using Lemma 2.1, our improved encoding, and given a depth of k, we need 4 bits to indicate that we are giving a relative binary path, plus the k bits for the actual path. Thus, we have $k + 4$ bits. ■

Lemma 2.3: Given an arbitrary graph with $n = 2^k$, m arcs, and an average degree $\delta = m/n$, a single node from the graph can be encoded with a total space of $\delta(\log_2(n/\delta)) + O(\delta)$ bits.

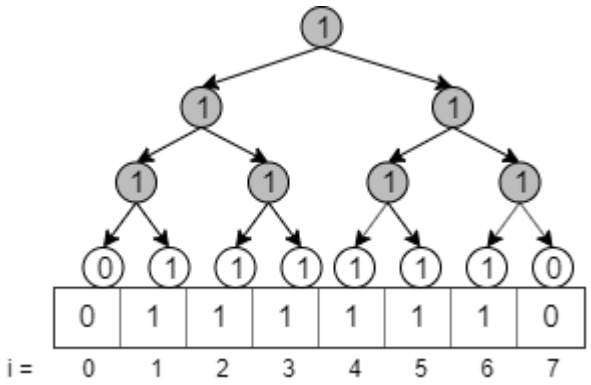

Figure 2.4: A full binary tree of a node with 6 neighbors, and 7 shared (gray) nodes

Proof: Using Lemma 2.2 and the graph conditions explained above, a compressed node would yield a total of $k\delta + 4\delta$ bits. However, not all of the paths (in the binary tree) to each neighbor may be unique. We can see that the worst case is when every node in the binary tree is present up to depth $\lfloor \log_2 \delta \rfloor$. We can see these shared nodes illustrated in Figure 2.4. After that level, the worst case follows with each path to the δ neighbors being unique, giving a total space of $\sum_{j=1}^{\lfloor \log_2 \delta \rfloor}(2^j) + \delta(k - \lfloor \log_2 \delta \rfloor - 1) + 4\delta = \delta(\log_2(n/\delta)) + O(\delta)$ bits. ■

Proposition: Given an arbitrary graph with n nodes and m arcs, the information-theoretic lower bound for storage is $m \log_2(n^2/m) + O(m)$ bits [18].

Theorem 2.1: Given an arbitrary graph with $n = 2^k$ nodes, m arcs, and average degree $\delta = m/n$, our compression's space requirement is $m \log_2(n^2/m) + O(m)$ bits, thereby achieving the information-theoretic lower bound.

Proof: Given the graph conditions explained above, we must use the formula in Lemma 2.3 for all the n nodes, giving $n(\delta(\log_2(n/\delta)) + O(\delta)) = m \log_2(n^2/m) + O(m)$. ■

A 3D-plot of our upper bound space requirements is given in Figure 2.5.

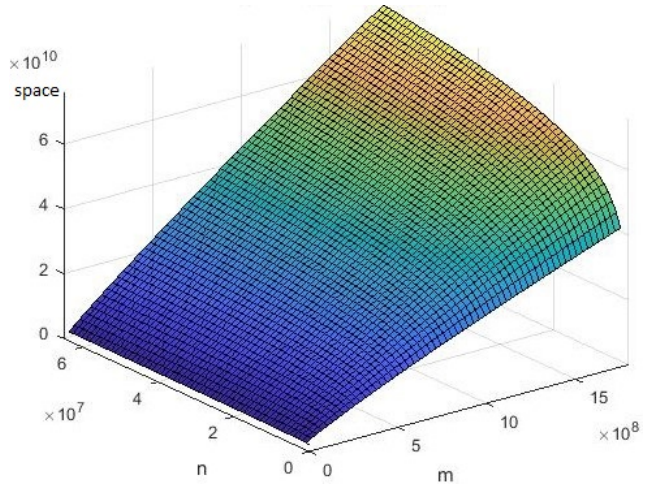

Figure 2.5: 3D-plot of space (bits) requirements of our technique given n (nodes) and m (arcs)

We can see that the worst case results in a perfect binary tree, which is where all nodes have two children and all leaf nodes are at the same depth. This gives us $2^{n+1} - 1$ nodes. These trees can be formed from graphs such that every other arc (in the sorted order) is missing, such as the adjacency matrix in Figure 2.6. Here, we are given a Boolean adjacency representation of a graph. Since it is checkered, there is no possible compression for this graph, and we are left with a perfect binary tree. There are other possible worst case graphs, but there can be

no two arcs missing, (u_n, v_n) and (u_{n+1}, v_{n+1}), such that n is even. The parent node of those arcs would become compressed, and we would no longer have a perfect binary tree. Since most real-world graphs are extremely sparse (e.g., social networks), they are far away from the worst case, as verified in Section 2.5.

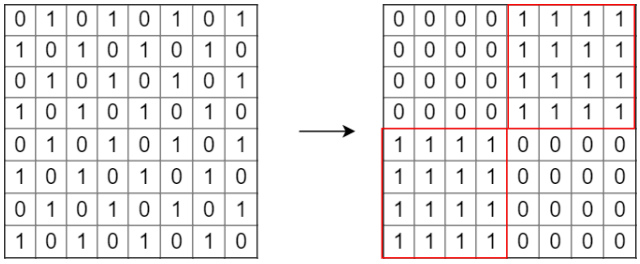

Figure 2.6: A worst case adjacency matrix (left); the same matrix, reordered (right)

2.4.5 Direct construction

Since the input graph is usually a list of arcs, there are two ways to compress. This first is to incrementally add (stream) each arc one at a time. This yields a time complexity of $O(m2^{\log_2(n)})$, which is slow on graphs with a number of arcs in the billions, especially due to the cost of the decompression step. A more interesting approach is to assume we are given all the arcs at once. After sorting the arcs, each node's compressed tree preorder traversal bit-string can be constructed left-to-right directly. This way, we can achieve an initial compression with a time complexity of $O(m \log_2(n))$. We now formally describe this process in Algorithm 2.3.

Algorithm 2.3 uses the visitor pattern for traversing a tree using preorder traversal. The visitor is responsible for keeping track of which node in the tree corresponds to the current bit we are reading. For each arc (line 5), we build the path to its leaf node. The span function on line 7 returns true if that node ranges

Algorithm 2.3: Binary tree compression

```
   Input: A node's sorted arc list
   Output: The compressed node as a bit-string
 1 begin
 2     BitString s;
 3     Node node = root;
 4     Visitor vtr = PreOrderTraversal(node);
 5     for arc=(u,v) in arclist do
 6         while !node.isLeaf() do
 7             if node.spans(v) then
 8                 s.AppendBit(1);
 9             else
10                 s.AppendBit(0);
11                 vtr.Ignore(node);
12             node = vtr.VisitNext(node);
13         s.AppendBit(1);
14     //fill rest of tree with 0s;
15     return s;
```

over our current arc, and we set the corresponding bit appropriately. If the node does not span our arc, we also tell our visitor to ignore the rest of that branch when traversing.

Finally, we use Algorithm 2.3 to compress each node and then we store them in an indexed array. We describe this process in Algorithm 2.4. For our index's integer encoding scheme, we use Delta Encoding [41].

Algorithm 2.4 loops through each node and uses Algorithm 2.3 to compress the node. While doing so, it also keeps track of the index position of the encoded node. Finally, it returns the list of indexes appended with the list of compressed nodes.

Algorithm 2.4: Indexed array of binary trees compression

Input: The graph as a sorted arc list
Output: The compressed graph as a bit-string

```
1 begin
2     length ← 0;
3     index ← int[|V|];
4     for each node, n do
5         index[n] ← length;
6         compressed[n] ← CompressNode (node.arcs);
7         length+ = Delta(compressed[n].length);
8     return index + compressed;
```

Proof of correctness

Theorem 2.2: Given a graph $G = (V, E)$, Algorithm 2.4 provides a lossless compression of G.

Proof: Since Algorithm 2.4 already outputs a compressed version of G, we must prove that this compression can be unambiguously reverted back to the original graph.

Let G' be the compressed version of G. Then it is sufficient to show that given G', we can obtain the original arc list E of the graph G.

Here we must make a note that the list of neighbors of each node is equivalent to E. That is, $\bigcup_{u=0}^{|V|-1}(u, v \in N(u)) = E$, where $N(u)$ lists the neighbors of u.

Therefore, since our technique compresses the neighbors of each node, it is sufficient to show that an arbitrary compressed binary tree correctly stores the neighbors of the node it belongs to. In other words, we must prove that every arc destination v appears as the appropriate leaf node in $u's$ binary tree.

As before, let u be an arbitrary node and $N(u)$ be the neighbors of u. Clearly, row u of the Boolean adjacency matrix will contain $1's$ for each index $v \in N(u)$ and $0's$ for every other index in the row.

If we start at the root of the compressed binary tree, then that node represents indexes 0 through $|V|-1$ of the adjacency matrix row. If the node is set to $TRUE$, then it has children to navigate to; i.e., the node leads to an arc. If it is set to $FALSE$, then it contains no children; i.e., it does not lead to an arc.

As we traverse through the left or right children, the range of indexes that the current node covers strictly decreases based on which child we navigated to. If we reach the maximum depth, the range of the current leaf node targets a single index of the adjacency matrix row. Since Algorithm 2.3 produces trees with leaf nodes only where the target index is $v \in N(u)$, and Algorithm 2.4 uses it to actually compress the nodes, our compression is correct. ■

Compressing directly from a gzip compressed arc list

During our experiments, we encountered such large graphs that even the raw arc list format required over 30GB of RAM. Since most computers these days do not have access to large amounts of main memory, we devised a way to compress directly from a much smaller 8GB gzipped arc list. It is important to note that the gzipped file must also be sorted.

The technique uses the zLib library [34] that gzip is built on. This library allows us to partially inflate (decompress) the file in chunks. Since the file is sorted, we can decompress a single node before we re-compress it with our method. Obviously, this technique only affects compression time and memory required for compression. These differences are shown in Section 2.5.

2.4.6 Querying the compressed structure

We provide three operations that can be performed on our structure: checking arc existence, getting a node's neighbors, and adding/removing arcs. While these are all separate operation, they all involve knowing how to efficiently traverse the compressed tree in bit-string form.

Normally when traversing trees, the user has access to pointers. However, since this is a compressed tree, we must read bit-by-bit from left to right. We stored the tree as labels in preorder traversal, therefore we must start at the root and keep track of where in the tree we are as we traverse. Algorithm 2.5 demonstrates this through the arc query.

Arc query

In this section, we present and describe the algorithm for checking arc existence in our structure. The process is formally given in Algorithm 2.5.

Like Algorithm 2.3, Algorithm 2.5 also uses the visitor pattern for traversing a tree in preorder. The current bit is read on line 5. As before, the span function in lines 7 and 11 returns true if the current node ranges over the arc we are looking for. If the node spans our arc and is a leaf node, then the arc has been found. If we encounter a node that spans our arc but is labeled as 0, then the branch has been compressed away and the arc does not exist. If it is only labeled 0, then we simply notify the visitor to ignore the rest of that part of the tree.

Neighbor query

The neighbor query completely reads through the compressed tree, returning any leaf nodes set to true. This can be done in one read, as we simply need to read

Algorithm 2.5: Array of Binary Trees (ABT) Compression - preorder Arc Query

```
   Input: The compressed adjacency row as a bit-string s, v
   Output: True or False
1  begin
2      Node node = root;
3      Visitor vtr = PreOrderTraversal(node);
4      for i = 0; i < s.Size(); do
5          label = s.GetBit(j);
6          if label == 1 then
7              if node.spans(v) then
8                  if node.isLeaf() then
9                      return True;
10         else
11             if node.spans(v) then
12                 return False;
13             vtr.Ignore(node);
14         node = vtr.VisitNext(node);
```

the entire tree.

Streaming arcs

Adding an arc begins by finding the deepest node along its path that hasn't been pruned off. Once we find the proper location, we build the rest of the path by inserting the proper bits into the bit-string. This process is formally described in Algorithm 2.6.

Removing an arc is the mirror of adding, but it still begins by finding the highest node in the path that it can compress. Then every bit representing nodes below it in the path are removed. Finally, the highest node is set to false. Compression benefits will be better the farther away the arc to be removed is from the other arcs.

Algorithm 2.6: ABT Compression - Streaming an arc

```
   Input: The compressed graph as a bit-string, x, y
 1 begin
 2     Node node = root;
 3     Visitor vtr = PreOrderTraversal(node);
 4     for i = 0; i < s.Size(); do
 5         label = s.GetBit(j);
 6         if label == 0 then
 7             if node.spans(v) then
 8                 if node.isLeaf() then
 9                     return;
10         else
11             if node.spans(v) then
12                 s.SetBit(j, 1);
13                 while !node.isLeaf() do
14                     if node.spans(v) then
15                         s.AppendBit(1);
16                     else
17                         s.AppendBit(0);
18                         vtr.Ignore(node);
19                     node = vtr.VisitNext(node);
20                 s.AppendBit(1);
21             vtr.Ignore(node);
22         node = vtr.VisitNext(node);
```

The streaming operation of Algorithm 2.6 is a combination of the arc query in Algorithm 2.5 and the construction method used in Algorithm 2.4. That is, it must first traverse through the tree to find the compressed node that ranges over our arc. This first step completes at line 11. Immediately, the next thing to do is set this node to 1 as we are about to partially decompress it. A partial decompression means we only add children set to 1 on the path to our leaf node. All other children are left compressed and set to 0. Since we are operating on a bit-string, this consists of inserting the nodes' bits into their proper position.

Now that we are using preorder traversal, all of these bits are guaranteed to be right next to each other.

Alternate In-Memory Structure

Now that we are streaming using preorder traversal, our main speed inefficiency comes from the time it takes to shift bits around. A typical vector from the C++ std library results in a copy of the underlying array after the maximum size is reached. Therefore, we introduce a structure for when the graph has been loaded into memory and is ready to have arcs streamed to it. The data structure is essentially an unrolled linked list adapted to using bit operations. We set k to be $\log(n)$ since this is the maximum length of any path along the tree.

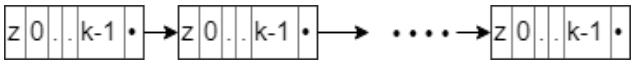

Figure 2.7: An unrolled linked list

In Figure 2.7, we illustrate our unrolled linked list. We can see that each node in the list contains three elements. First is a number z, indicating how much of the node has actually been used. Second, we have the actual vector of size k. Finally, we have the pointer to the next node in the list. We do not mind that the list is limited to sequential access, as we have to start from the beginning of the tree with each read anyway.

2.5 Experiments and results

Our experiments involve running ABT compression and our benchmark Backlinks compression [27] on the datasets given in Table 2.1. These datasets are available from snap.stanford.edu. Backlinks compression (BLC) was chosen as

	Directed?	$\|V\|$	$\|E\|$	#Reciprocal arcs
Facebook	FALSE	4039	88234	88234
Friendster	FALSE	65608366	1806067135	1806067135
LiveJournal	TRUE	4847571	68993773	26142536
LiveJournal(com)	FALSE	3997962	34681189	34681189
NotreDame	TRUE	325729	1497134	407026
Pokec	TRUE	1632803	30622564	8320600
Twitter	TRUE	81306	1768149	425853

Table 2.1: The dataset stats

the benchmark compression since it is the state-of-the-art technique for social networks [83]. We have also compared the bits-per-arc of these two techniques against Slashburn compression [84] and reported the results in Table 2.8. We chose to compare against Slashburn since they are technically a reordering algorithm that is subsequently compressed with gzip, which we also apply to our compressions as an additional step.

In this chapter, we have set BLC to use BFS for the reordering algorithm and we have set the window size to $k = 10$. The datasets are stored as an arc list and the final output of the compressions is a bit-string.

We also provide side-by-side comparisons of ABT when using BFS versus preorder traversal.

We run all of our algorithms on a machine with an Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (4 cores) with 64GB of RAM.

2.5.1 Compression

Table 2.2 shows a comparison among the sizes of the raw graphs, our array of compressed binary trees (ABT) compression, the BLC benchmark, and all of their

	.txt	.txt.gz	ABT	ABT.gz	BLC	BLC.gz
Facebook	835KB	214KB	93.02KB	92KB	110.32KB	93KB
Friendster	31GB	8.2GB	5.5GB	5.2GB	5.7GB	5.2GB
LiveJournal	1.1GB	248MB	120.85MB	119MB	139.98MB	122MB
LiveJournal(com)	479MB	119MB	108.33MB	94MB	110.21MB	95MB
Pokec	405MB	127MB	66.40MB	64MB	70.59MB	64MB
Twitter	20MB	6.1MB	2.92MB	2.9MB	3.3MB	3.0MB

Table 2.2: Compressed graph sizes

	ABT	BLC
Facebook	0.32s	1.1s
Friendster	4.1h	10h
LiveJournal	8.6m	18.1m
LiveJournal(com)	4.0m	11.4m
Pokec	4.3m	8.9m
Twitter	10.9s	26.3s

Table 2.3: Compression times

	txt → ABT	gz → ABT	txt → BLC	gz → BLC
Facebook	931KB	310KB	3027KB	2406KB
Friendster	39GB	15GB	58GB	36GB
LiveJournal	1324MB	366MB	1818MB	966MB
LiveJournal(com)	590MB	231MB	920MB	560MB
Pokec	516MB	183MB	711MB	433MB
Twitter	25MB	9MB	38MB	24MB

Table 2.4: Compression memory usage

Confidence Level=95%	ABT-BFS (μs)	ABT-PRE (μs)	BLC (μs)
Facebook	110.677 ± 2.869	76.453 ± 4.410	120.181 ± 11.800
Friendster	457.349 ± 10.707	293.589 ± 19.777	8353.309 ± 1770.793
LiveJournal	343.058 ± 7.511	156.454 ± 9.367	4335.078 ± 873.965
LiveJournal(com)	179.324 ± 4.100	84.836 ± 6.793	2892.767 ± 163.617
Pokec	120.422 ± 2.162	42.571 ± 3.490	1618.460 ± 153.371
Twitter	440.113 ± 4.915	196.446 ± 12.881	864.995 ± 87.509

Table 2.5: Arc existence execution times

Confidence Level=95%	ABT (μs)	BLC (μs)
Facebook	82.270 ± 1.252	148.547 ± 8.076
Friendster	301.553 ± 8.102	8802.230 ± 1688.793
LiveJournal	159.892 ± 3.684	4568.701 ± 809.209
LiveJournal(com)	85.897 ± 2.512	3059.430 ± 137.012
Pokec	46.760 ± 1.133	1765.696 ± 122.169
Twitter	200.461 ± 3.385	920.333 ± 75.571209

Table 2.6: Neighbor query execution times

respective gzipped files. The *.txt* files are the original arc list representations of the graphs. The *.txt.gz* are those files after being gzipped. This data is important because we have implemented a method of compressing the graph directly from the smaller gzip files.

The data in Table 2.3 shows the total run time of both the ABT and BLC algorithms on each graph. Clearly, the run-times depend on the size of the input graph.

In Table 2.4, we list the different memory requirements for running our algorithms. Not only does it show that ABT requires less memory than BLC, but it also shows the benefits of loading directly from a gzipped graph file.

Additionally, we provide Table 2.8 as a quick comparison with Slashburn [84]. The size metrics are in the traditional $bits - per - arc$.

When examining the compression sizes of ABT and BLC, we see that ABT outperforms BLC on every graph. Our new improvements involving better encoding of runs of ones as well as branches only containing one arc allows us to outperform BLC's use of differential encoding. This raises the question of whether ABT can be even further compressed with differential encoding.

BLC's process of building copy lists is also one of the reasons why their com-

Confidence Level=95%	ABT-BFS (μs)	ABT-PRE (μs)	ABT-ULL (μs)	BLC (μs)
Facebook	676.265 ± 112.029	388.765 ± 212.474	77.472 ± 4.632	-
Friendster	4109.340 ± 532.659	2074.590 ± 1389.989	291.719 ± 20.785	-
LiveJournal	2256.770 ± 309.552	1294.901 ± 719.367	166.538 ± 9.496	-
LiveJournal(com)	480.729 ± 45.348	232.235 ± 140.790	84.495 ± 7.577	-
Pokec	309.195 ± 66.481	186.863 ± 104.001	42.830 ± 3.856	-
Twitter	1003.675 ± 115.175	489.237 ± 269.207	192.524 ± 13.748	-

Table 2.7: Add arc execution times

	ABT	ABT.gz	BLC	BLC.gz	SB
LiveJournal	15.7	15.4	16.2	15.4	16.5
Barabasi	14.2	8.3	20.8	10.3	8.5

Table 2.8: Comparison with Slashburn graphs (bits-per-arc)

pression times are so long. As previously mentioned, we have set the window size to a common $k = 10$. This means that we actually examine each node 10 times. Additionally, since BLC requires an adjacency list intermediate structure, the query times for that list also affect compression run-times. ABT builds the structure directly from the arc list, therefore it compresses much faster.

Our reasoning for applying the additional gzip compression is a matter of in-memory querying versus storing on secondary memory. A non-gzipped compressed graph is easily queryable but takes up more memory. Once the user is done querying the graph, they may gzip the structure and store it in secondary memory.

Next, we report that ABT uses much less memory during compression than BLC. This is obviously because ABT does not require an intermediate structure in order to efficiently compress. This benefit is massive, as many techniques are restricted to smaller graphs due to the larger graphs requiring too much memory. This fact, coupled with our new ability to also compress directly from the gzipped arc list file, guarantees minimal memory overhead.

Finally, since we use gzip to improve our results, we also include a compression comparison with Slashburn [84]. This is because Slashburn is technically a reordering algorithm that uses gzip to compress blocks in the adjacency matrix that formed as a result of the reordering. For the sake of consistency with Slashburn's paper, we present the results using the traditional bits per arc metric.

2.5.2 Query operations

Table 2.5 shows the arc query execution confidence intervals, assuming a confidence level of 95%. All times are in microseconds. Table 2.6 is the same as Table 2.5, but for the neighbor query instead.

Our ABT structure facilitates two query operations, the arc query and the neighbor query. These are also the two queries supported by BLC. Since we have updated our query operations after switching to preorder traversal we include a comparison of query times between BFS and preorder.

Both BFS and preorder traversal have a worst case of reading the entire bit-string. This is because while BFS always has to go towards the end of the bit-string, preorder's first arc may be only $\log_2(n)$ bits deep into the tree. This is reflected in the experimental query times.

Since both compression techniques are node-centric, we can use indexes for fast access to each node. For ABT, once the node of interest has been navigated to, we immediately begin reading the compressed tree. If it is an arc query, we stop when we find the arc, or when we find a compressed node that indicates that the arc does not exist. If it is instead a neighbor query, we must read the entire tree.

Similar to ABT, BLC may also use indexes to jump to the node being queried. However, the decoding process is more complicated than reading our simple tree. It not only involves back-tracing the copy lists, but also many integer decodings. Thus, on average, ABT outperforms BLC on both the arc and neighbor queries.

The neighbor query is essentially the same as the arc query for both ABT and BLC. Just as ABT requires the entire tree to be read, BLC requires that the entire adjacency list be read. Though again, BLC's neighbor query suffers from

the same problem that its arc query has with the copy lists. Therefore, since the queries are so similar, we can see that their access times are nearly identical but with higher variance.

2.5.3 Streaming operations

Table 2.7 is the same as Table 2.5 but for streaming access times. Note that since BLC is not a streaming compression, it does not have any entries.

As described earlier, our compression method supports the streaming operation. That means that we are able to directly add/delete arcs into the compressed structure without having to completely re-compress the graph. Conversely, BLC does not support this operation, mainly due to its incorporation of copy lists.

In Algorithm 2.6, we state that the streaming process is identical to the arc query operation. The only difference is that once we have navigated to the correct node, we may decompress or compress it by adding or removing the appropriate bits respectively.

By comparing Table 2.5 and Table 2.7, we see that although the arc query times are clearly faster than the streaming times, there is still a direct relationship between them. Since both operations navigate the tree in the same manner, the gaps between times are due to the actual cost of moving bits in the bit-string.

Streaming operations are where benefits of the switch to preorder really shine. Before, the BFS ordering would cause all related bits to an arc to be scattered across the bit-string. This would cause us to navigate to and edit multiple places throughout the bit-string. Now, all bits relating to an arc are all grouped together, resulting in one localized edit.

We also observe the time savings introduced by our adapted unrolled linked

list. This speed-up is attributed to the linked list not needing to recopy the entire bit-string, as the previous vector implementation did.

2.6 Conclusions

In this chapter, we have improved upon our previous queryable, incremental network compression using an indexed array of compressed binary trees [100]. We build this structure directly from the graph's gzipped arc list text file without using any intermediate structure such as an adjacency list. These two techniques guarantee minimal memory overhead. Our compression also supports compressed queries, namely the arc and neighbor queries in addition to arc additions and removals (streaming operations) directly from the compressed structure.

Our improvements involve query and streaming speedups by massively improving the encoding scheme to better encode branches full of arcs and branches that only contain one arc, changing the traversal ordering from BFS to preorder, and by providing an in-memory structure to more quickly shift/insert bits. We also better encode reciprocal arcs for directed graphs, allowing us to only encode the upper triangular matrix.

We also provide various comparisons among our compression, Backlinks [27] (as a benchmark) and Slashburn [84] (as an addition). These comparisons use metrics such as compression size, time to compress, memory usage, and query times. Our experiments show that our improvements achieve better compression than BLC. Furthermore, our improved basic query operations run, on average, 20 times faster than our BLC benchmark. This combined with our improved capability for streaming makes our compression highly desirable. Lastly, we believe our compression technique can be used in the algorithms described in [92] and [84] to further reduce the number of bits per arc.

Chapter 3

Boolean matrix-vector multiplication on compressed static graphs

Billion-scale Boolean matrices in the era of big data occupy storage that is measured in 100s of petabytes to zetabytes. The fundamental operation on these matrices for data mining involves multiplication which suffers a significant slowdown as the required data cannot fit in most main memories. In this chapter, we propose new algorithms to perform Matrix-Vector and Matrix-Matrix operations directly on compressed Boolean matrices using innovative techniques extended from our previous work on compression. Our extension involves the development of a row-by-row differential compression technique which reduces the overall space requirement and the number of matrix operations. We have provided extensive empirical results on billion-scale Boolean matrices that are Boolean adjacency matrices of webgraphs. Our work has significant implications on key problems such as page-ranking and itemset mining that use matrix multiplication.

3.1 Introduction

Frequent itemset mining and page ranking are classical data mining problems that involve the use of Boolean matrices. A Boolean matrix is a matrix where each entry is either true or false, represented as 1 or 0, respectively. For frequent itemset mining, with n items we have an $n \times n$ matrix M and $M(i, j) = 1$ if items i and j occur in the same transaction, otherwise it is 0. A variation of this involves a matrix whose column size is equal to the number of items and row size is equal to the number of transactions. If a transaction i contains an item j, then the corresponding position of the matrix is set to 1, otherwise 0.

A web graph is then a graph $G = (V, E)$ where V is the set of nodes (web pages) and $(u, v) \in E$ means that page u has a link to page v. These graphs can be represented as a $|V| \times |V|$ Boolean adjacency matrix. That is, a cell (u, v) in the matrix is a 1 if $(u, v) \in E$ and is 0 otherwise.

Let $n = |V|$ and A be our $n \times n$ matrix representing G. Given a real-valued vector $x \in \mathbb{R}^n$, a matrix-vector multiplication is then $A \cdot x^T$. This operation is key in many algorithms such as association rule mining [82] and PageRank [107] [28], which are our main motivations and discussed in our experiments.

Most real-world matrices such as the transaction-item matrix and the adjacency matrix corresponding to the web graph are very sparse, i.e., most of the cells are 0. Rather than running a brute force Matrix-Vector multiplication algorithm in $O(n^2)$ time, it would be preferable to reduce the number of operations to $O(m)$, where m is the number of non-zero cells in the matrix which could be much less than n^2.

There have been a number of works that compress a graph's Boolean matrix [39] [40] [16] [12] [84] [101]. Out of these compressions, k^2-trees [16] and We-

bgraph [12] are generally considered to be state-of-the-art, with results in [16] showing that k^2-trees outperform Webgraph in all datasets. Although similar to k^2-trees, our work uses better compression, faster to query, supports streaming, and contains many other improvements. Additionally, only Webgraph [12] and our work involves differentially compressing the Boolean adjacency matrix row-by-row. Results show that these compression schemes are the most suited for matrix operations thus far.

Franciso et. al [50] used the compression algorithm of Boldi and Vigna [12] to perform matrix-vector multiplication directly on the compressed matrix. They found that because of the row-by-row differential compression technique, they could reuse a product $A_i \cdot v^T$ in the calculation of $A_j \cdot v^T$ if j was compressed differentially from i. This reduced the total number of matrix-vector operations to being proportional to the *size of the compressed graph* [50].

Our contributions can be summed up as follows:

- We improve our previous compressed binary trees algorithm [100] [101] by adapting it for differential compression with an enhanced encoding scheme. The improved encoding allows us to better compress branches that *only contain one maximum depth leaf node* and branches that are *full*.

- We provide algorithms for fast matrix-vector multiplication that directly work on the compressed structure by reusing previously computed products.

- We extend this technique to perform fast Boolean matrix-matrix multiplication. Here there are no intermediate matrix presentations and the final resultant matrix is directly stored as the differential compressed binary trees.

- Our empirical results for matrix-vector multiplication demonstrate that this new technique outperforms the technique of Franciso et. al [50] on all datasets in terms of execution time.

The rest of the chapter is organized as follows. In Section 3.2, we discuss related work in data mining applications, compression algorithms, and other matrix-vector multiplication speedup techniques. In Section 3.3, we describe differential compressed binary trees with improved encoding. The matrix-vector multiplication is described in Section 3.4 and the matrix-matrix multiplication is presented in Section 3.5. In Section 3.6, we present our experimental findings with discussion, and we conclude in Section 3.7.

3.2 Related work

We know that many data mining algorithms require repeated matrix-vector multiplication. Most famously, we have the PageRank algorithm [107], which calculates a score that measures the importance of web pages. This can be done by repeatedly multiplying a Boolean adjacency matrix of the web graph with some real-valued vector. We initialize this vector to all ones, and we run our experiments on PageRank using differentially compressed binary trees.

In other data mining applications, we also have diameter calculation [71], which involves solving the all-pairs shortest paths problem via repeated matrix-matrix multiplication and returning the largest diameter [68]. Our technique would differentially compress the Boolean matrix that we use to repeatedly multiply. This not only saves space, but also improves calculation time by allowing the reuse of products.

We also provide compression algorithms for Boolean Matrix-Matrix multi-

plication, which is helpful for other data mining algorithms such as connected components [6] [62] [53] [127]. While the technique no longer benefits from the reuse of products, it still saves space and allows us to short-circuit the Boolean Matrix-Vector multiplication. This means we can return true as soon as a single AND term is satisfied.

This chapter is based on our previous work involving arrays of compressed binary trees [100] [101]. The compression concept is similar to k^2-trees [16], except we are the first to have each row of the matrix mapped to a compressed binary tree with our improvements (ours is a row-by-row compression technique). We also use different encoding schemes, and supply a technique to find the differential CBT between two CBTs with these same improvements.

In 2018, Francisco et al. [50] studied the Matrix-Vector multiplication benefits of the Webgraph Framework by Boldi and Vigna [12]. Their results show that speedup is indeed possible through *computation-friendly* compressions.

In 2014, Nishino et al. [105] used adjacency forests to speedup matrix multiplication. Their technique allows adjacency lists to share common suffixes. However, the authors consider matrices with real values instead of Boolean matrices. Although they are not a compression, they observed that computational results could indeed be reused. They also ran their experiments using PageRank and their results show that similar adjacency rows allows for better compression and Matrix-Vector multiplication speedups.

We note that this work is also relevant to the field of Online Matrix-Vector (OMV) multiplication. Given a stream of binary vectors, $x_1, x_2, ...$ if adjacent vectors are similar enough, the results of previous vectors can be reused to speedup computing later rows [61] [80]. However, none of the existing approaches preprocesses the graph to exploit its redundancies. Thus by using techniques such as

node reordering, we can achieve a better compression, and thus better computation times.

3.3 Differential compressed binary trees

In this section, we show how an $n \times n$ Boolean matrix can be represented as a series of differentially compressed binary trees (CBT′). Note that our technique is very general and can be applied to matrices whose number of columns and rows can be different. We then improve CBT′ by introducing a new encoding scheme which allows us to better compress *branches that are full* and *branches that only contain one maximum depth leaf node* (Section 2.4.3).

In Figure 3.1, we have a Boolean matrix (a) and its series of CBT′s (b). We see that each CBT′ in (b) represents a row of the adjacency matrix in (a). Figure 3.1 (c) is the complete bitstring of the trees output in preorder traversal.

These CBT′s use the standard encoding where a one represents that the branch contains a change, and a zero indicates otherwise. A consequence of this standard encoding is that branches containing a change must travel to the maximum depth of $\log_2(n)$. What makes these CBT′s *differential*, is that when a CBT′ encodes a node u, it only encodes the changes from node $u-1$, i.e., $|A_{u+1} - A_u|$, where $0 < u < n$.

When we are encoding a node u_j, we may also maintain a window W allowing us to choose any node u_i to differentially encode from, where $0 < j - i \leq W \leq n$ where $0 \leq i < j < n$ and n is the number of nodes. Figure 3.1 has W set to 1.

In Figures 3.2 and 3.3, the rows that the CBTs represent are obtained from $|A_{i+1} - A_i|$. This means that an entry v in the resulting row $A'_i = |A_{i+1} - A_i|$ indicates that $A_{i+1,v} \neq A_{i,v}$. In a differential CBT, a zero indicates that the

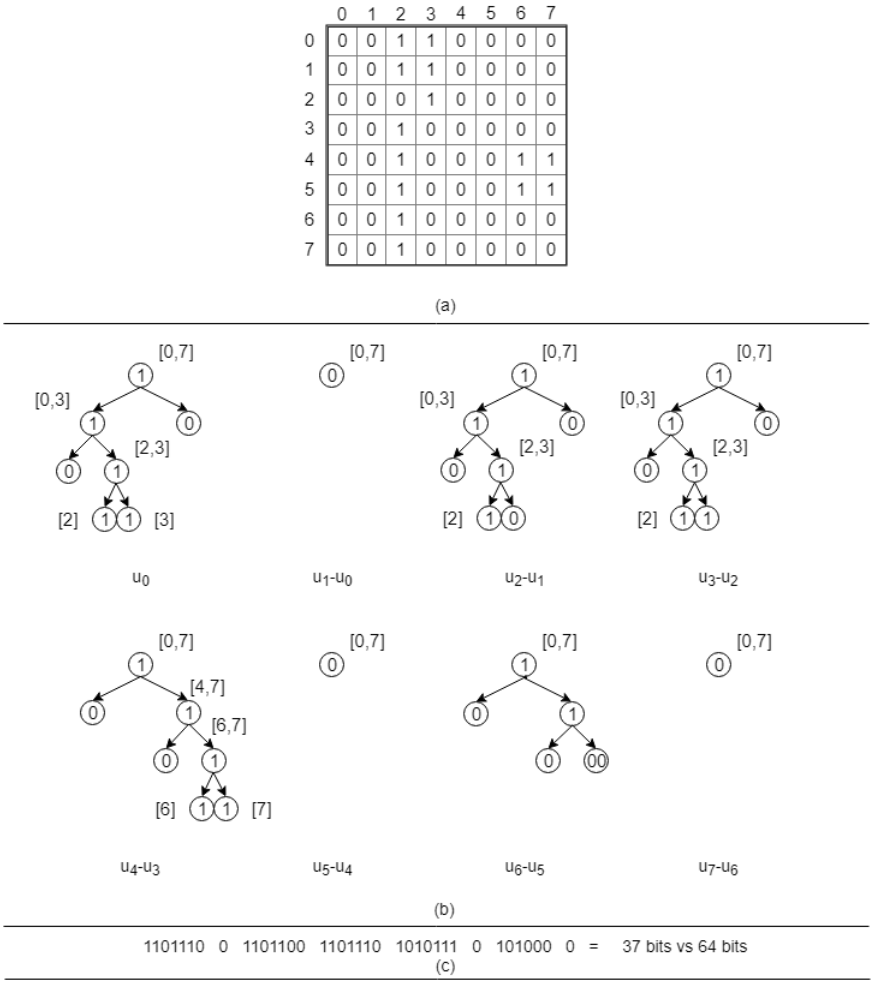

Figure 3.1: A Boolean matrix (a) and a series of differential compressed binary trees (b); bit-strings in preorder traversal (c)

branch does not contain any ones, and a one indicates that the branch does contain a one.

3.4 Matrix-vector multiplication

Assume we are given a CBT′ (a differential CBT), j, the CBT$_i$ it was differentially encoded from, where $i < j$, and $A_i \cdot x^T$. Now, when we calculate $A_j \cdot x^T$, we can simply calculate the product of the differential, $(A_j - A_i) \cdot x^T$, and add

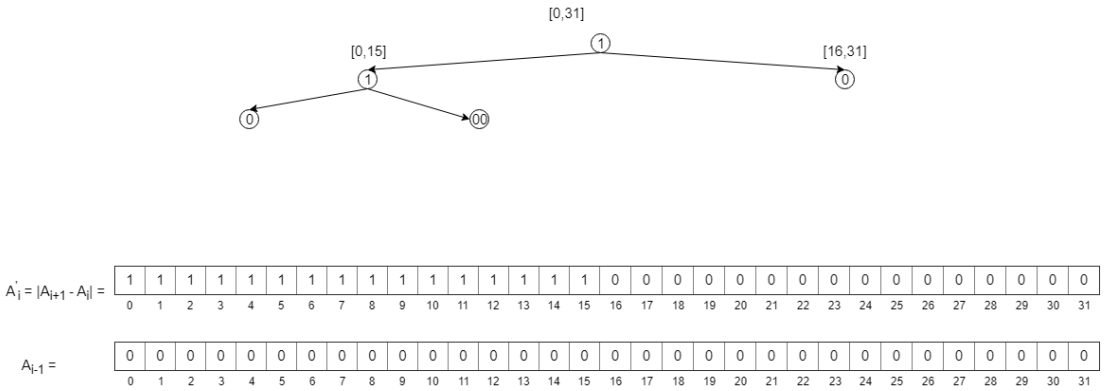

Figure 3.2: A CBT of row A_i at time t. The ones and zeros are both compressed.

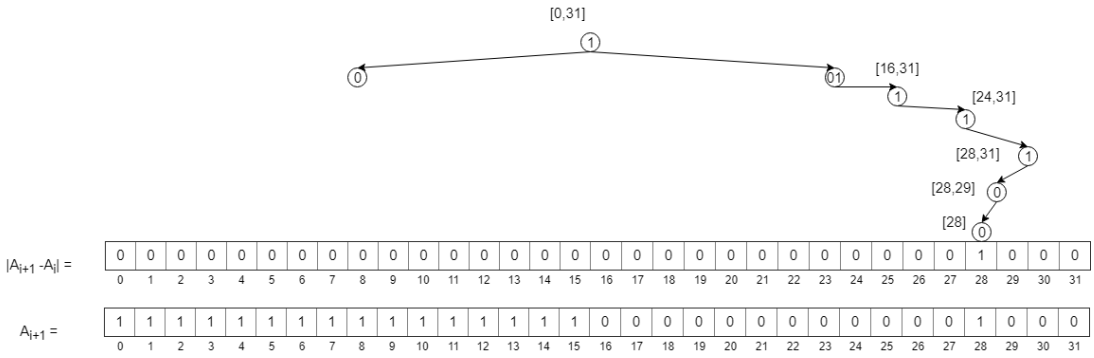

Figure 3.3: A CBT′ with a single arc at [28] being added to the CBT from Figure 3.2.

it to the result of $A_i \cdot x^T$, i.e., $A_j \cdot x^T = ((A_j - A_i) \cdot x^T) + (A_i \cdot x^T)$. Thus, we have a *computation-friendly* compression that is able to compute matrix-vector multiplication in time proportional to the size of the compressed graph, rather than proportional to the number of non-zeros in A. We can then see that an appropriate node-reordering algorithm combined with larger values of W will decrease the size of the compressed graph, and thus further improve the computation time for matrix-vector product.

We also slightly modify the encoding again to include an extra bit after each leaf node corresponding to a change. This bit will allow us to know whether

the change was a removal (0) or an addition (1) of an arc. This modification is needed for our multiplication algorithm to know whether to subtract or add to the product. We now describe Algorithm 3.1, which uses this new encoding to calculate $(A_j - A_i) \cdot x^T$.

Algorithm 3.1: CBT-Vector Multiplication

```
   Input: CBT a, int[] v
   Output: int av
1  begin
2      int av = 0;
3      Node node = a.Root;
4      Visitor vtr = PreOrderTraversal(node);
5      while !vtr.End() do
6          if node.Label == 1 && IsMaxDepth() then
7              av = av + v[node.Index];
8          else if node.Label == 1 && node.Left.Label == 0 &&
               node.Right.Label == 0 then
9              if node.IsPathBit == 1 then
10                 index = node.RelativeBinaryPath();
11                 av = av + v[index];
12             else
13                 av = av + sum(v, node.begin, node.end);
14         else if node.Label == 0 then
15             vtr.Ignore(node);
16         node = vtr.VisitNext(node);
17     return av
```

In Algorithm 3.1, we take CBT a representing some adjacency row A_j and the multiplying vector v, and we output $A_j \cdot v$ as an integer. For ease of reading, we use the visitor pattern with preorder traversal (lines 3-4). First we handle the standard encoding where we only add to our product when we reach arcs at the bottom of the CBT (lines 6-7). Then, we check our improved encoding cases where the node is labeled one, but followed by two zeros (line 8). If the

compression is a relative binary path (line 9), we get the index the path is pointing to and add the v's value at the index to the product (lines 10-11). If the branch contains all ones (line 12), then we add all values in the current node's range (line 13). If the CBT is pruned off with a zero, then we know that we can skip this entire branch (lines 14-15). We return the product on line 17.

Note that with the improved encoding, we can process multiple arcs quickly if the branch is compressed with all ones. In other words, if the current branch spans $[begin, end]$ and it contains all ones (as represented by a one followed by "000") we no longer have to read any more of our tree and we can perform quick sums to add directly to our product av. We now describe Algorithm 3.2 which uses Algorithm 3.1 to perform matrix-vector multiplication using our differential compression.

Algorithm 3.2: CBT Matrix-Vector Multiplication

```
   Input: CBT[] a, int[] v
   Output: int[] av
 1 begin
 2     int[] av = new int[n];
 3     for int i = 0; i < n; i++ do
 4         CBT u = a[i];
 5         if u.IsDiff() then
 6             uv = u.Multiply(v);
 7             av[i] = av[u.Source] + uv;
 8         else
 9             av[i] = u.Multiply(v);
10     return av;
```

In Algorithm 3.2, we take as input an array of CBTs representing the rows of our Boolean matrix A and the multiplying vector v, and we output an array of integers that is the product $A \cdot v$. We start by looping through each CBT u representing row A_u, for $0 \leq u < n$ (lines 3-4). If the CBT is a differential (line 5),

then there is a node *source* from which u was differentially encoded and for which we have already calculated $A_{source} \cdot v$. We then calculate the differential product $(A_u - A_{source}) \cdot v$ (line 6), and add it to the source node's result $A_{source} \cdot v$. This step is one of the main contribution of the chapter where we achieve our speed-up by reusing products. If the CBT is not a differential (line 8), we perform a normal multiplication and set the result in its appropriate position (line 9). Finally, we return $A \cdot v$ on line 10.

Theorem 3.1: Given an $n \times n$ matrix M with $n = 2^k$, m number of 1's, and average number of 1's in each row $\delta = m/n$, our CBT Matrix-Vector multiplication (Algorithm 3.2) runs with worst case time complexity $O(m \log_2(n^2/m))$, or $O(m \log_2(n(n-1)/2m))$ if we are only using the upper triangular matrix (where we assume $M(i,j) = M(j,i)$).

Proof: Given the matrix conditions above, we assume run-time is equal to the number of bits we must process. Thus, we refer to our space complexity given in Theorem 2.1, which gives $O(m \log_2(n^2/m))$ bits. If we are only using the upper triangular matrix, then we substitute n^2 with $(n^2 - n)/2$, which leads to $O(m \log_2(n(n-1)/2m))$ bits. ■

Note again that compression size determines Matrix-Vector multiplication time, and CBT's size is equal to the theoretic minimum of $m \log_2(n^2/m) + O(m)$ [18].

3.5 Matrix-matrix multiplication

In this section, we take the technique from Section 3.4 and adapt it to Boolean Matrix-Matrix multiplication. That is, we perform $A \times B = C$ where A, B,

and C are matrices represented as CBTs. Using CBTs allows us to perform matrix-matrix multiplication with reasonable memory and that includes storing the output directly as CBT. Now, notice that this involves multiplying the rows of A with the columns of B, and that we also want to reuse C in place of B for the next iteration. This means we must encode A such that its CBTs are its rows, and B and C such that their CBTs are their columns. Thus, instead of multiplying to construct C row-by-row, we instead want to multiply such that we can build C column-by-column, top-to-bottom. This is as simple as looping through B's columns, then by A's rows, instead of vice-versa. Also, since this is Boolean multiplication, multiplying vectors involves switching the multiplication with the AND operator, and the addition operation with OR. Although we lose the benefit of reusing our products for calculation as in the cast of Matrix-Vector multiplication where the vector contains real values, we gain the ability to short circuit the multiplication and return true as soon as we find the first true AND term. We now describe Algorithm 3.3, which multiplies two CBTs.

In Algorithm 3.3, for the sake of brevity, we only describe multiplication using the standard encoding. As with Boolean vector multiplication, we take two CBTs representing our input vectors and we output true or false. We begin by traversing both trees at the same time using preorder traversal (lines 1-4). If either of the current nodes is labeled zero (line 5), then we know that they have no arcs in common and can thus ignore that entire branch for both trees (lines 6-7). However, if both nodes are labeled one and we have reached the bottom of the trees (line 8), then we short-circuit and return true for the multiplication (line 9). If we finish traversing either tree (line 4), then we have not satisfied any AND operator and we return false (line 12).

Lemma 3.1: Assume we measure time by how many bits (nodes) we must

Algorithm 3.3: CBT-CBT (Boolean vector-vector) multiplication

```
   Input: CBT a, CBT b
   Output: bool ab
1  begin
2      Node node_a = a.Root, node_b = b.Root;
3      Visitor vtr_a = PreOrderTraversal(node_a) , vtr_b =
        PreOrderTraversal(node_b);
4      while !vtr_a.End() ∧ !vtr_b.End() do
5          if node_a.Label == 0 || node_b.Label == 0 then
6              vtr_a.Ignore(node_a);
7              vtr_b.Ignore(node_b);
8          else if node_a.Label == 1 && node_b.Label == 1 &&
            IsMaxDepth() then
9              return true;
10         node_a = vtr_a.VisitNext(node_a);
11         node_b = vtr_b.VisitNext(node_b);
12     return false;
```

traverse in the binary tree. Given two Boolean arrays A and B of size n, a single CBT-CBT multiplication (Algorithm 3.3) runs with a worst-case time of $O(\delta'(\log_2(n/\delta')))$, where δ' is the maximum number of 1s in A or B.

Proof: Notice that even though we know to ignore the current branch when we encounter a zero in either tree, the other tree's visitor may still need to read through its branch to calculate where the next branch begins. Thus we know that the worst case must read through all the distinct arcs from both trees, i.e., δ', which is the maximum number of 1s in array A or B. Also, notice that nodes near the root are double counted since we are reading two trees at the same time. We then adjust our logic in Lemma 2.3 for our unioned tree, and it gives us $\sum_{j=0}^{\lfloor \log_2 \delta \rfloor}(2^{j+1}) + \delta(\log_2 n - \lfloor \log_2 \delta \rfloor - 1) + \delta = O(\delta'(\log_2(n/\delta')))$. ■

Theorem 3.2: Given two matrices M_a and M_b and number of arcs m_a and m_b, respectively with $|V| = n = 2^k$ and $\delta_a = m_a/n$ and $\delta_b = m_b/n$, the CBT Matrix-

Matrix multiplication in Algorithm 3.3 runs with worst-case time complexity of $O(m' \log_2(n^2/m'))$, where $m' = m_a + m_b$. If we are only using the upper triangular matrix then we have time complexity of $O(m' \log_2(n(n-1)/2m'))$.

Proof: We use the same notion as in Lemma 3.1, where we realize we must traverse each 1 in the cell of each matrix , i.e., $m' = m_a + m_b$. Our worst case is then $O(m' \log_2(n^2/m'))$, and if we are using upper triangular matrix we substitute our n^2 with $(n^2 - n)/2$, which leads to $O(m' \log_2(n(n-1)/2m')))$. ■

3.6 Experiments and results

We implemented all of our programs in C/C++ with an AMD FX(tm)-8350 Eight-Core @ 4.00Ghz and with 32GB of RAM. We ran 10 iterations of matrix-vector multiplication for the Web graphs in Tables 3.2 and 3.3, with v initialized to all ones. We compare the time to calculate these multiplications against our benchmark, Web Graph by Boldi and Vigna (BV) [12]. We do not make any comparisons against the uncompressed matrix-vector multiplication since we would have to hold a 143.1 petabyte Boolean adjacency matrix in main memory.

	Size_{BV}	$\text{Size}_{CBT'}$	n	m
eu-2015-hc	9.7GB	9.5GB	1.07×10^9	9.17×10^{10}
eu-2015-host-hc	145MB	138MB	1.13×10^7	3.87×10^8
gsh-2015-hc	1.3GB	1.2GB	9.88×10^8	3.39×10^{10}
it-2004-hc	194MB	177MB	4.13×10^7	1.15×10^9
uk-2014-hc	5.7GB	5.6GB	7.88×10^8	4.76×10^{10}

Table 3.1: The datasets

Table 3.1 shows the Web crawl datasets used in the experiments along with their sizes in Webgraph [12] and CBT′. Table 3.2 contains the computation time results for Webgraph by Boldi and Vigna [12]. The column m is the number of

	m	m'_{BV}	t_{BV} (s)	t'_{BV} (s)	$S_{BV} = \frac{t_{BV}}{t'_{BV}}$
eu-2015-hc	9.17×10^{10}	1.11×10^{10}	1715.5	591.8	2.9
eu-2015-host-hc	3.87×10^{8}	1.10×10^{8}	5.4	3.97	1.36
gsh-2015-hc	3.39×10^{10}	7.08×10^{9}	973.9	544.1	1.79
it-2004-hc	1.15×10^{9}	2.27×10^{8}	14.7	7.4	1.99
uk-2014-hc	4.76×10^{10}	6.26×10^{9}	1172.3	391.9	2.99

Table 3.2: Experimental results with Webgraph by Boldi and Vigna [12]

1's, m' is the number of differential non-zeros, t is the average time in seconds to compute matrix-vector multiplication *without* differential compression, t' is the time to multiply *with* differential compression, and S is the observed speedup. Table 3.3 contains the same information as in Table 3.2, but using CBT. The results show that $S_{CBT} > S_{BV}$ and $t'_{CBT} < t'_{BV}$. The column S_{CBT}/S_{BV} also show the speedup as a percentage of using CBT vs BV.

We start by compressing each graph to a sequence of CBT′s. Then we calculate m'_{CBT} as the total number of nonzero entries represented by the compressed structure. This m'_{CBT} is different than m'_{BV} since one structure may find it beneficial to use its own differential compression, while the other may not. Note that we keep a CBT_j when there is no i such that $sizeof(CBT_j) > sizeof(CBT_{Diff_{j,i}})$ and $0 \leq j - W \leq i < j$. In other words, it takes less space to normally compress A_j than to compress it differentially.

Examining the results in Tables 3.2 and 3.3, we see that although m'_{CBT} and m'_{BV} have very similar values, m'_{CBT} is slightly smaller. This is because the compression found it more beneficial to reuse more rows of the matrix, meaning that the ratio CBT_{Diff}/CBT is smaller on average than BV_{Diff}/BV. This fact helps explain why $S_{CBT} > S_{BV}$.

Our approach outperforms BV in terms of computation speeds t and t' in all

	m	m'_{CBT}	t_{CBT} (s)	t'_{CBT} (s)	$S_{CBT} = \frac{t_{CBT}}{t'_{CBT}}$	S_{CBT}/S_{BV}
eu-2015-hc	9.17×10^{10}	1.10×10^{10}	1502.9	480.7	3.13	8.0%
eu-2015-host-hc	3.87×10^{8}	1.09×10^{8}	4.6	3.2	1.44	6.0%
gsh-2015-hc	3.39×10^{10}	7.08×10^{9}	853.9	451.2	1.89	5.6%
it-2004-hc	1.15×10^{9}	2.26×10^{8}	11.7	5.5	2.10	5.5%
uk-2014-hc	4.76×10^{10}	6.25×10^{9}	1001.5	311.1	3.22	7.7%

Table 3.3: Experimental results with CBT

graphs. It outperforms in terms of t mainly due to a faster decoding method and better compression brought about by the improved encoding methods from Section 2.4.3. Additionally, it outperforms on t' because m'_{CBT} was smaller than m'_{BV}, meaning we reused more computations.

3.7 Conclusions

We have adapted our previous work involving differential compressed binary trees to allow for matrix-vector multiplication. We have taken advantage of the techniques described in [50] to allow us to reuse results for later computation. This means that our computation time is now proportional to the compressed matrix size, rather than the number of non-zeros in the original matrix. Therefore, the better the compression is able to exploit redundancies, the faster the computation time will be.

Our experiments show that differential CBTs allow for faster Matrix-Vector multiplication than a similar adaptation of Boldi and Vigna. The first reason for this is because our differential compression is succinct enough that the overall compression allows for more reuses of encoded rows. We also massively improved the encoding scheme in Section 3.3. Secondly, Boldi and Vigna's technique also makes use of integer encoders, which is more complex than our tree traversals.

We also extend our technique to support Boolean Matrix-Matrix multiplication which is used to find connected components and transitive closures. This extension loses the ability to reuse products, but gains the ability to short-circuit and return early.

We also observe that an interesting future work would involve reordering the rows of the matrix in order to improve differential compression. Notice that we would also have to reorder the multiplying vector or matrix correspondingly.

Chapter 4

Compressing time-evolving graphs using binary trees

In this chapter, we propose to build a compressed data structure that has a compressed binary tree corresponding to each row of each adjacency matrix of a time-evolving graph. We do not explicitly construct the adjacency matrix, and our algorithms take the time-evolving arc list (i.e., differential contacts) representation as input for its construction. Our compressed structure allows for directed and undirected graphs, fast arc and neighborhood queries, as well as the ability for arcs and frames to be added and removed directly on the compressed structure (streaming operations). We use publicly available network data sets such as Flickr, Yahoo!, and Wikipedia in our experiments and show that our new technique performs as well or better than our benchmarks on all datasets in terms of compression size and other vital metrics.

4.1 Introduction

A time-evolving graph consists of a set of nodes in which the arcs among them change over time. If we had a time-evolving graph of a web or social network graph such as Facebook, we would be able to see how the graph evolved as nodes and arcs are added and removed. In other words, we would be able to examine the graph at any point in time, and directly see which nodes and arcs were active at that time.

While many structures already exist to model time-evolving graphs, few focus on implicitness and succinctness [11]. Now that networks such as web and social networks are reaching such large scales with dynamic lifetimes, we need both more main memory and compact yet fast time-evolving structures. If the user cannot fit the graph into main memory, then they must make access calls to disk, which incurs a high time penalty. During this work, we processed a Yahoo! network flow time-evolving graph that only spanned three days yet occupied 21.5GB of space. This massive amount of data presents a clear problem to areas such as time-evolving graph pattern analysis.

When developing a queryable compression technique, it is usually designed to answer a certain set of queries. While one technique may be efficient at answering a query such as whether an arc is active at a particular time, it may be inefficient at a query such as getting all arcs at a given time. A common requirement for most compression algorithms is an intermediate structure, such as an adjacency list, that is built from this list of triplets and used to efficiently build a final compressed structure. Since we can incrementally build our compression, we do not require such an intermediate structure. This, combined with our previously developed technique of compressing directly from a gzipped text file, guarantees

minimal main memory overhead. For example, after preprocessing, our Yahoo! dataset was 21.5GB as a raw text file. After gzipping, it shrunk down to 6.7GB, giving us a main memory saving of 14.8GB.

Our novel compression technique is based on our previous work of using indexed arrays of binary trees [100]. We adapt the original technique to time-evolving graphs by compressing each node's adjacency matrix row in each 2D matrix time slice with *differentially* encoded compressed binary trees (CBT). We also greatly improve the original definition of a compressed binary tree to better compress runs of zeros and ones, as well as branches that contain a single one. We also provide a method for aggregating a series of differential CBTs into a single time spanning CBT, if more space efficient.

Our contributions can be summed up as:

- We adapt the technique introduced in our previous work [100] to compress time-evolving graphs, in a queryable and streaming structure.
- We greatly improve the original notion of a compressed binary tree to compress runs of zeros and ones, as well as branches that only contain a single one.
- We provide a differential compression technique that saves space by only encoding the differences between a node's frames.
- We maintain minimal memory overhead by not requiring any intermediate structure (such as adjacency lists or matrices) to compress and by being able to compress directly from a gzipped file.
- We provided an in-depth space complexity analysis as well as a proof of correctness on our structure.

- We have developed algorithms to execute time-sensitive arc, neighbor, and streaming operations that directly work on the compressed file.
- We provide a detailed empirical study that uses real, massive time-evolving graphs and show that our technique performs as well or better than our benchmarks on all datasets.

The rest of the chapter is organized as follows. In Section 4.2, we better define time-evolving graphs and discuss the operations that can be performed on them. In Section 4.3, we examine existing structures and describe our benchmarks. We improve the notion of a compressed binary tree and apply it to time-evolving graphs in Section 4.4. We also provide an analysis, a proof of correctness, and supported operations with algorithms in this section. Finally, we report empirical results in Section 4.5 and conclude in Section 4.6.

4.2 Preliminaries

In this section, we better define time-evolving graphs and the operations that can be performed on them.

4.2.1 Time-evolving graphs

A time-evolving graph is a graph where arcs appear and disappear as time passes [103]. Additionally, it retains the history of the graph at each time frame. Therefore, we should be able to see the graph exactly as it was at each time frame. A perfect example of this phenomenon is the popular social network, Facebook, and how users are constantly adding/removing friendships.

Given a web or social network, it can be represented as a graph $G = (V, E)$, where V is a set of nodes (individuals) and E is a set of arcs (relationships). Thus, a time-evolving graph would be a series of graphs $G_T = G_1, G_2, ..., G_\tau$, where τ is the number of time frames, i.e., the *lifetime*. Since a simple graph $G = (V, E)$ can be represented as a 2D matrix, then a time-evolving graph could be represented as a 3D matrix.

When downloading a time-evolving graph off the web, we usually find them represented as a *contact list* [23]. A contact is a 4-tuple (u, v, t_i, t_j) meaning that the edge (u, v) existed (inclusively) between the time frames t_i and t_j, where $0 \leq i < j < \tau$ and τ is the lifetime of the graph.

From here, we provide several definitions that will be helpful in understanding the different concepts surrounding time-evolving graphs. We denote $n = |V|$ as the number of vertices, $m = |E|$ as the number of arcs, and τ as the number of time frames, i.e., the *lifetime*.

Definition 4.1. An *aggregated graph* of a time-evolving graph G_T is the static graph defined by all arcs that have been active during the lifetime of G_T. We can also aggregate the graph from a given time-interval $[t_i, t_j]$, where $0 \leq i < j < \tau$.

Definition 4.2. An *incremental graph* is a time-evolving graph where once an arc has been activated, it remains so until the end of the lifetime.

Definition 4.3. An *interval graph* is a time-evolving graph where arcs exist in time intervals $[t_i, t_j)$, where $i < j$. An arc may exist in several intervals and intervals may not overlap.

Definition 4.4. A *point-contact graph* is a time-evolving graph where arcs only appear at a single time frame, i.e., $[t_i, t_{i+1})$. arcs may appear at multiple points.

4.2.2 Operations on time-evolving graphs

Time-evolving graphs have many different applications, such as web and social network graphs, communication networks, citation networks, etc. In each area, the queries of interest are similar to those of static graphs, but with the addition of the time dimension. We can see that these queries can be divided into four categories shown in Table 4.1. Each of these queries can be targeted towards a particular point in time, or a time interval. First, we have queries about vertices, including retrieving the direct/reverse neighbors of a vertex. Next, we have queries about arcs, such as determining whether an arc is active or not. We also have queries about the graph as a whole, such as retrieving all active arcs. Finally, we have queries about changes in the graph. For example, return all arcs that changed state during the requested time period. We describe these operations in Table 4.1.

4.3 Related work

Intuitively, a time-evolving graph can be represented as a sequence of static graphs (snapshots), with each snapshot representing the graph at a particular point in time. Since a snapshot can be represented as a 2D matrix, a time-evolving graph can thus be represented as a 3D matrix, also known as a *presence matrix* [47]. Formally, a *presence matrix* is a 3D binary matrix of size $n \times n \times \tau$, where an entry (u, v, t) represents whether the arc (u, v) is active at time t. The main problem with this 3D representation is its large space requirements, and that the arcs may remain unchanged for long time intervals. For example, as a 3D matrix, the Wiki-Links dataset would require $22608064 * 564224135 * 414347809 = 26.5$ zettabytes.

Table 4.1: Operations on time-evolving graphs

Class	Time-evolving operation
Vertices	**DirectNeighbors(u,t):** returns active adjacent neighbors of **u** at time **t**
	ReverseNeighbors(v,t): returns active reverse neighbors of **v** at time **t**
arcs	**arc((u,v),t):** returns true if arc **(u,v)** is active at time **t**, false otherwise
	arcNext((u,v),t): returns the instant of the next activation of **(u, v)** after **t**, or **t** if it is active; otherwise returns ∞
Graph	**Snapshot(t):** returns all active arcs at time **t**
Changes	**ActivatedArcs(t):** return all arcs that were activated at time point **t**
	DeactivatedArcs(t): return all arcs that were deactivated at time point **t**
	ChangedArcs(t): return all arcs that were activated or deactivated at time point **t**

In 2016, Caro et al. [23] developed ck^d-trees, which we use as our main benchmark. We do this not only because they have top compression rates, but also because their technique is similar to ours in that they also use compressed trees. They define a contact as a quadruplet (u, v, t_i, t_j) and then compress the 4D binary matrix corresponding to the time-evolving graph defined by a set of these contacts. They do this by representing the 4D matrix as a k^d-tree and then distinguishing white nodes as those without any contacts, black nodes as ones that only contain contacts, and gray nodes as those that contain only one contact. This work was preceded by Brisaboa et al's k^2-trees [16] in 2014.

When compressing time-evolving graphs, two popular strategies are to either (i) store the events of activation/deactivation of arcs or (ii) store changes between snapshots by having representative snapshots and the log of events between them. These are often referred to as log and copy+log, respectively.

Continuing on the snapshot representation idea, the G^* database [78] is a distributed index that solves the space issue of the presence matrix by only storing new versions of an arc when its state changes, i.e., as a log of changes. They do this by storing versions of the vertices as adjacency lists and maintaining pointers to each time frame. If an arc changes in the next frame, they create a new adjacency list for that vertex's arc and add a pointer to the new frame. The DeltaGraph [75] is also a distributed index, but it groups the different snapshots in a hierarchical structure based on common arcs.

EveLog [22] is a compressed adjacency log structure based on the *log of events* strategy. It consists of two separated lists per vertex - one for the time frames and another for representing the arcs related to the event. The time frames are compressed using gap encoding, and the arc list is compressed with a statistical model. We can see that query times suffer because we need to sequentially scan

the log.

If we want to check if an arc is active at a particular time frame in the log strategy, we must sequentially read the log of events (possibly deactivating/reactivating the arc) until the time frame is reached. We can see that this approach is slow for large time-evolving graphs, as it is done in linear time. Ferreira et al. [47] follow this strategy by providing a quadruplet $(u, v, t, state)$ for each time an arc changes. In order to improve query times, in [20] the same authors also present a data structure of adjacency lists where each neighbor has a sublist indicating the time intervals when the arc is active. In arcLog [22], this idea is compressed using gap encoding.

When querying copy+log at a time t, we must select the snapshot closest to t and process the log of changes that occurred from the snapshot to t. In [115], the authors develop the FVF (Find-Verify-Fix) framework which includes a copy+log compression that also supports shortest-paths and closeness centrality queries. More preliminary work is done in [10] [51], which describes three different methods to index time-evolving graphs based on the copy+log strategy.

Two *log of events* strategies, CAS and CET, are proposed in [22] to address the problem of slow query times when processing a log. CAS orders the sequence by vertex and adds a Wavelet Tree [54] data structure to allow for logarithmic time queries. CET orders the sequence by time, and the authors develop a modified Wavelet Tree called Interleaved Wavelet Tree to also allow logarithmic time queries.

In 2014, Brisaboa et al. [14] adapt compressed suffix arrays (CSA) [22] for use in temporal graphs (TGCSA) by treating the input sequence as the list of contacts. They use an alphabet consisting of the source/destination vertices and the starting/ending times.

4.4 Time-evolving graphs as differentially compressed binary trees with improved encoding

As described in Section 4.2.1, a time-evolving graph can be represented as an $n \times n \times \tau$ 3D matrix, where $n = |V|$. Additionally, it can be represented as a series of graphs $G_0, ..., G_t, ..., G_{\tau-1}$, where $G_t = (V_t, E_t)$ and $0 \leq t < \tau$.

Our compression represents each node u ($0 \leq u < n$) at each time frame t ($0 \leq t < \tau$). In other words, we compress the adjacency row representing the neighbors of u at each time frame, *differentially*. In Figure 4.1 (a), we have a graph of size $n = 6$ with $\tau = 5$. Then, for each u, we represent each u_t with differential compressed binary trees (b). That is, the compressed binary trees only represent neighbor changes since *the previous time frame*. For the sake of brevity, only the first three nodes are compressed and illustrated with their respective CBTs. However, the reader can find the final bit-strings for all nodes at the bottom (c). We describe this more in Section 4.4.2. We can see that our structure requires 130 bits, whereas the 3D matrix representation needs 180 bits. For undirected graphs, our binary tree would only represent the 3D upper triangular matrix, giving 72 bits for our structure and $((n^n - n)/2) \times \tau = 90$ bits for the 3D upper triangular matrix.

Our compression technique for time-evolving graphs is as follows:

- Compressed binary trees (CBT) are implemented with an improved encoding to better compress **consecutive arcs** and **branches that only contain one arc**. (Section 4.4.1).

- Given a node i at time frames $t - 1$ and t where $1 \leq t < \tau$, the adjacency

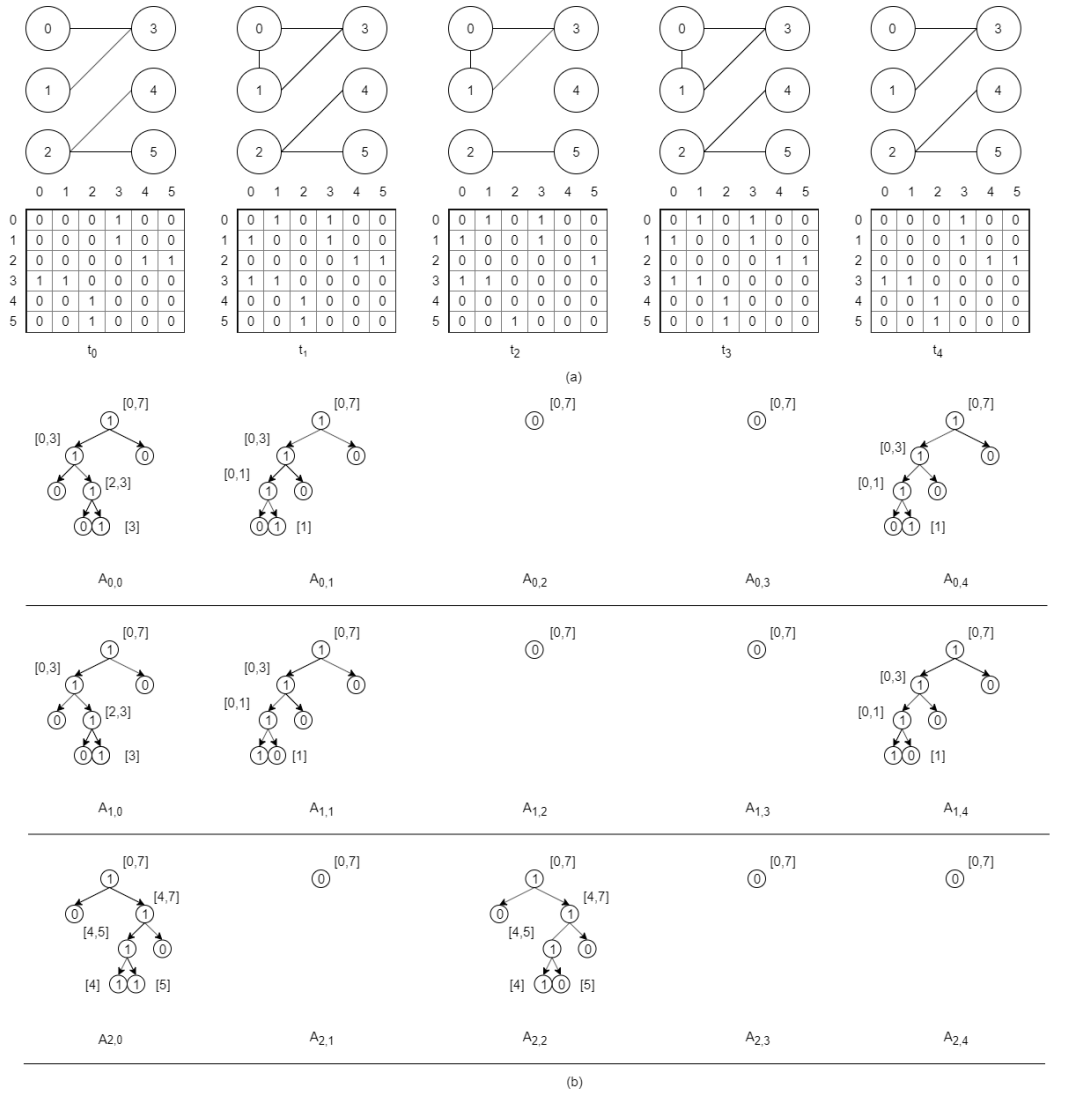

Figure 4.1: A time-evolving graph with $n = 6$ and $\tau = 5$ represented as a series of differential compressed binary trees

row $A_{i,t}$ is encoded differentially from $A_{i,t-1}$. That is, a CBT is encoded to represent the absolute difference between the adjacency rows $|A_{i,t} - A_{i,t-1}|$. Only changes since *the previous time frame* are encoded. (Section 4.4.2)

- A window W is examined to determine if a series of CBTs can be more efficiently represented as a single time spanning compressed binary tree (T-CBT) (Section 4.4.3).
- The structure is outputted as a series of CBTs and T-CBTs with indexes pointing to the beginning positions of each node (Section 4.4.4).

We also provide analysis in Section 4.4.5, a proof of correctness in Section 4.4.6, and supported operations with algorithms in Section 4.4.7.

4.4.1 Improved compressed binary trees

In this section, we introduce the compressed binary tree with the improved encoding techniques illustrated in Figures 4.2 and 4.3. In Figure 4.2, we have a row A_i at a time t compressed with a binary tree using our new encoding scheme. We can see that the branches containing only zeros are pruned off and compressed with a single zero bit as usual. Now, notice that if a node is marked with a one, it should normally never have its two children both marked with zero [23]. We take advantage of this fact by saying that if a node is followed by two children marked with zeros, then an additional bit must follow. If this next bit is zero, then the branch is filled with all ones. If the bit is one, then the branch leads to a single arc, as in Figure 4.3.

In Figure 4.3, we have row A_i again, but at time $t+1$ with one additional arc at position 28. This row is encoded with a CBT *differentially* from the CBT of $A_{i,t}$

in Figure 4.2. We explain this more in Section 4.4.2. This example demonstrates when a node's two children are both marked zero, and the following bit is a one. In this case, it means that the current branch leads to a single arc to which we then provide a direct binary path, relative to the current branch. That is, while the example's change is encoded at the root of the tree ($d = 0$), it could occur at any depth $d < \log_2(n) - 3$ with a saving of $\log_2(n) - 2d - d - 3 = \log_2(n) - d - 3$ bits. We now describe Algorithm 4.1 which outputs a relative binary path given a target index j and a node's beginning and ending range.

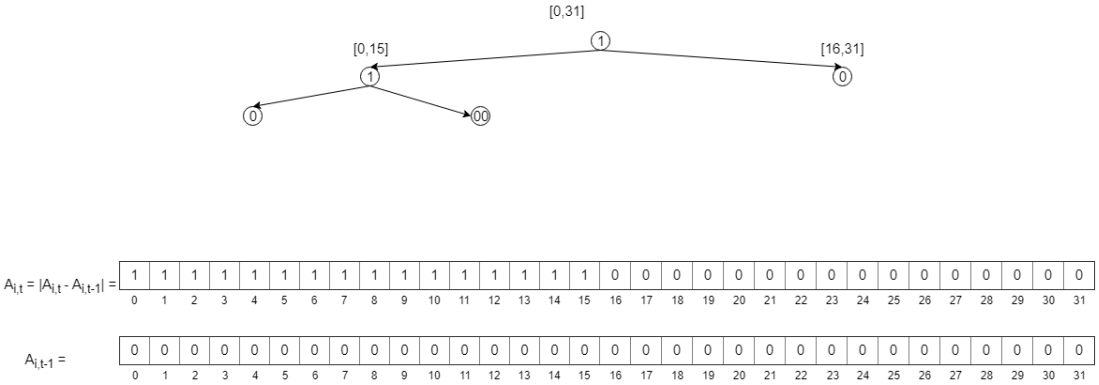

Figure 4.2: A CBT of row A_i at time t. The ones and zeros are both compressed

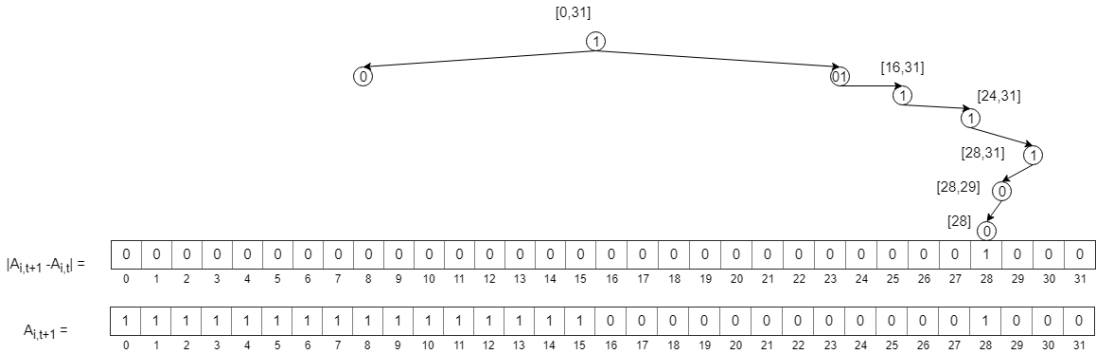

Figure 4.3: A CBT′ with a single arc at position 28 being added to the CBT from Figure 5.2

In Algorithm 4.1, we are given "begin" and "end" which represents the range of our current node, along with the target index j. We calculate the relative depth

Algorithm 4.1: A relative binary path

```
   Input: int begin, int end, int j
   Output: The relative binary path as a bit-string
1  begin
2      BitString s;
3      depth = ⌈log_2 (end − begin)⌉;
4      s.Initialize(depth);
5      for i = 0; i < depth; i++ do
6          mid = ⌈(begin + end)/2⌉;
7          if begin ≤ j < begin + mid then
8              s.AppendBit(0);
9              end − = mid;
10         else
11             s.AppendBit(1);
12             begin + = mid;
13     return s;
```

in line 2, which will be the length of our returned bit-string. In lines 5 through 12, we loop through each level of the remaining depth, and append a one or a zero, depending on which child the path should navigate to (left = 0, right = 1). We return the path's bit-string in line 13.

4.4.2 Differential compressed binary trees

In Figures 4.2 and 4.3, the adjacency rows that the CBTs represent are obtained from $|A_{i,t+1} - A_{i,t}|$. This means that an entry v in the resulting row $A'_{i,t+1} = |A_{i,t+1} - A_{i,t}|$ indicates that $A_{i,v,t} \neq A_{i,v,t+1}$. In a differential CBT, a zero indicates that the branch does not contain any change, and a one indicates that the branch does contain a change. We now describe Algorithm 4.2 which also includes the improved encoding scheme described in Section 4.4.1.

In Algorithm 4.2, we are given as input a node's compressed binary tree (CBT) and a list of arc targets. If a target is already a neighbor of the node, it is to

Algorithm 4.2: Differential binary tree compression

Input: $\text{CBT}_{i,t}$, and a list of arc change targets
Output: $\text{CBT}'_{i,t+1}$ as a bit-string

```
1  begin
2      BitString s;
3      Node node = cbt.Root;
4      Visitor vtr = PreOrderTraversal(node);
5      while !vtr.End() do
6          nodeTargets = targets.Where(x => node.Spans(x));
7          if nodeTargets.Count() > 0 then
8              s.AppendBit(1);
9              if node.arcs() == nodeTargets then
10                 s.AppendBitString('000');
11                 vtr.Ignore(node);
12             else if nodeTargets.Count() == 1 then
13                 s.AppendBitString('001');
14                 path = RelativeBinaryPath(node.begin, node.end, target);
15                 s.AppendBitString(path);
16                 vtr.Ignore(node);
17         else
18             s.AppendBit(0);
19             vtr.Ignore(node);
20         node = vtr.Next();
21     return s;
```

be removed. We use the visitor pattern and some LINQ notation [9] for ease of reading. We start by traversing through the CBT representing the node's current neighbors. On line 6, we get the list of target changes pertaining to the current node. If there are changes in this branch, we have three cases: (i) the branch is removed entirely, (ii) the branch has one change, or (iii) the branch has many changes. The last case is handled simply by line 8. The first case makes use of our new encoding scheme in line 10 by appending '000', indicating that the entire branch has been removed. The second case handled in lines 12 through 16 also uses a new encoding scheme by appending '001', followed by the relative binary

path to the target change. If there were no changes for this node, then we simply append a zero, as in line 18. We return the differential CBT as a bit-string in preorder traversal in line 21.

The bit-strings for Figures 4.2 and 4.3 would be 110000 and 100111100, respectively.

4.4.3 Time-spanning compressed binary trees

Over time, it may be more space-efficient to combine these differential binary trees into one T-CBT structure [101]. This T-CBT is not a differential, but a complete representation of the node during that time-span. The paper in [101] mainly mentions T-ABTs, so when we say T-CBT, we are referring to a node's single T-CBT out of the entire T-ABT. However, we use variants of our improved encoding scheme introduced in this chapter on the original technique [101].

We briefly describe a T-CBT. First, for some node i, we build a *time-aggregated adjacency row* A'_i which is the result of **inclusively or'ing** all $A_{i,t}$ where $begin \leq t \leq end < \tau$ and $[begin, end]$ is the desired time span. That is, A'_i contains all neighbors that node i has ever had within that time-span. Then, we build a CBT representing A'_i. However, once we reach a binary leaf node representing an arc v, we then begin another binary tree spanning the row $A_{i,v,t}$ where $begin \leq t \leq end < \tau$. In other words, this new CBT spans the arc's desired time dimension where a 1 indicates whether or not the arc was active at that time frame. We adopt the same concept involving both of a node's children having zeros from Section 4.4.1. If the third bit is a one, we build a relative binary path directly to the arc, as above, followed by another CBT representing the arc's time dimension. If the third bit is a zero, then the node contains all ones. We then

list all the arcs' time dimension CBTs *in order, differentially.*

In Figure 4.4, we can see that the space required for $CBT(A_{i,t})$+$CBT'(A_{i,t}, A_{i,t+1})$ requires 7 more bits than $T\text{-}CBT(A_{i,t}, A_{i,t+1})$, which conveys the same information [101]. Thus, when compressing from t_i to t_j, where $i < j$, we check to see if $\sum_i^j(Size(CBT(t_i))) > Size(\text{T-CBT}(i, j))$. This trade-off can be explained by the fact that T-CBTs only encode arcs once, while a series of CBT′s accrue extra space because arcs will get repeated. Therefore, we can see that our overall structure for a single node will be a series of T-CBTs, followed by a series of CBT′s.

4.4.4 Overall structure

When describing our technique for compressing, we must first envision each time slice of the graph as its 2D Boolean adjacency matrix representation. For algorithmic simplicity, we expand the matrices to the next highest power of two.

We start by compressing each node entirely, including all of its time dimension. That is, for some node u, we are compressing $A_{u,0}$ through $A_{u,\tau-1}$, where τ is the lifetime of the graph. The first instance of the node u at time t_0 will be a regular $\text{CBT}(A_{u,0})$. From there we append the node at each subsequent time frame with CBT′s, as described in Section 4.4.2. Then, for some window W and t_i to t_j, where $0 \leq j - W \leq i < j$, we check to see if $\sum_i^j(Size(CBT(t_i))) > Size(\text{T-CBT}(i, j))$. When this inequality is satisfied, we substitute CBT′s i through j for a single $\text{T-CBT}_{(i,j)}$, as described in Section 4.4.3. Thus, we can see that our overall structure will be a sequence of T-CBTs, with possibly different time-spans, mixed with CBT′s, for each node.

We then apply indexes coupled with a lifetime to each node for quicker node-

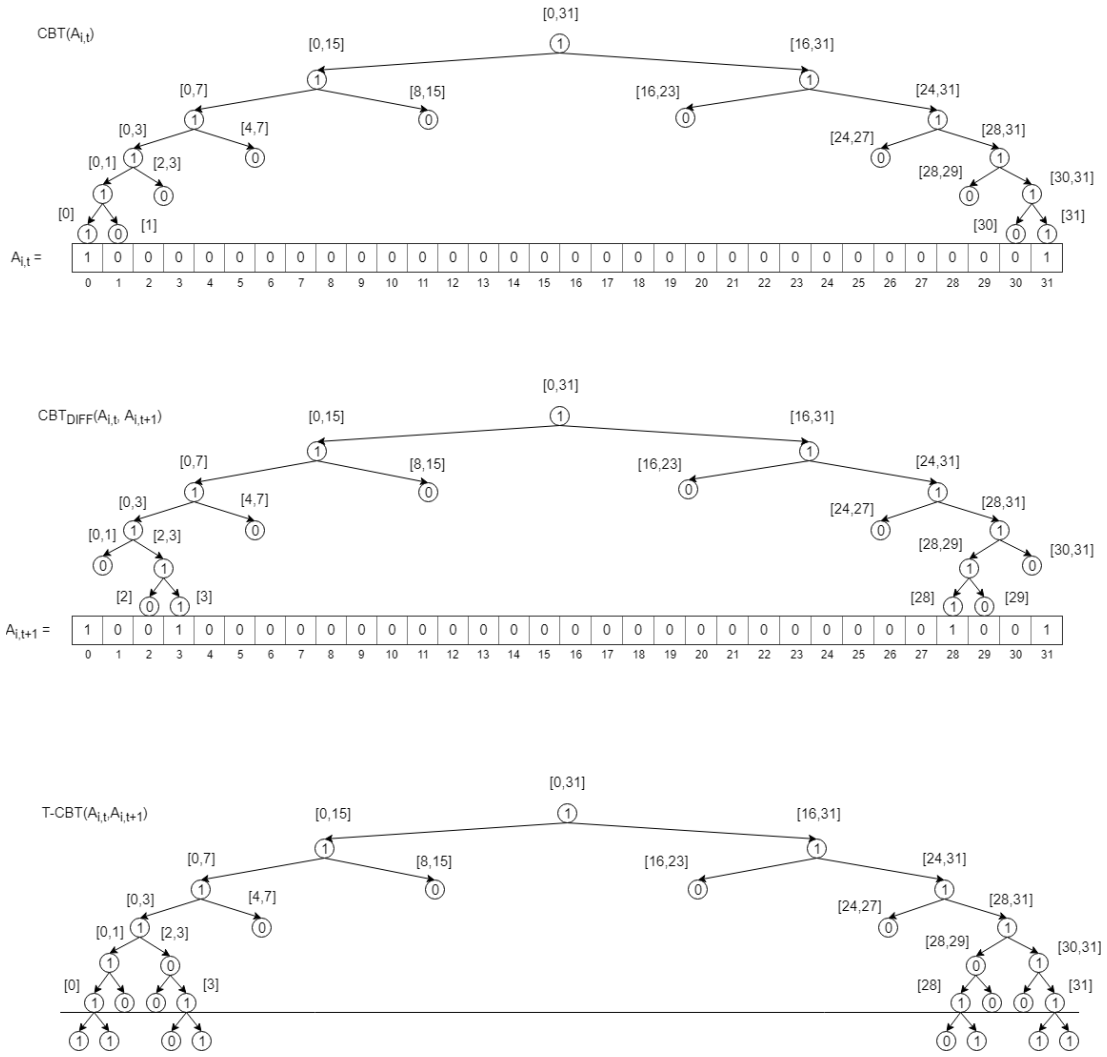

Figure 4.4: A CBT of row $A_{i,t}$ with two arcs (top), a CBT′ from $A_{i,t}$ to $A_{i,t+1}$ with an additional two arcs (middle), and a T-CBT from $A_{i,t}$ to $A_{i,t+1}$ (bottom)

centric queries throughout time. This is a common practice and sacrifices minimal space for a great speed increase on queries [83]. In our case, the space requirement is $O(nd\log(n))$, yet we gain a time complexity of $O(d\log(n))$, where d is the contact degree of the graph.

Input

As mentioned in Section 4.2.1, when downloading a time-evolving graph off the web, we usually find them represented as a *contact list* [23]. A contact is a 4-tuple (u, v, t_i, t_j) meaning that the edge (u, v) existed (inclusively) between the time frames t_i and t_j, where $0 \leq i < j < \tau$ and τ is the lifetime of the graph. Before we compress, we preprocess this input to be a list of 3-tuples (u, v, t) indicating that the edge (u, v) changes state at time $0 \leq t < \tau$. Thus, every 4-tuple contact (u, v, t_i, t_j) becomes two triplets, (u, v, t_i) and (u, v, t_j). We call these triplets *differential contacts*. If an arc already exists and appears again in a later time frame, then the arc is to be deactivated (and vice versa). This list of triplets is to be sorted in ascending order, by u, then v, then t.

These assumptions are only necessary for a direct construction, rather than an incremental one.

Direct construction

If we are given the sorted input graph all at once, we can construct each tree's preorder bit-string directly. This includes the T-CBT's neighbor and time trees, as well as every CBT$'$. This way, we can achieve an initial compression with a time complexity of $O(c\log(n))$. This process is the same as in Algorithm 4.3 and [101].

Algorithm 4.3 uses Algorithm 4.2 and the T-CBT technique from Section 4.4.3

Algorithm 4.3: Time-evolving compression using binary trees

```
   Input: The graph as a sorted list of triplets (u, v, t)
   Output: The compressed graph as a bit-string
 1 begin
 2     diffSize ← 0;
 3     for i= 0 to n − 1 do
 4         for t= 0 to τ − 1 do
 5             node ← GetNeighborsForFrame(n, t);
 6             cbt-diff ← CompressToCBTDiff(node);
 7             tcbt.Add(node);
 8             if tcbt.Timespan() ≥ W then
 9                 tcbt.RemoveLastFrame();
10             if sizeof(tcbt) < sizeof(cbt-diff)+diffSize then
11                 bitstring.Remove(diffSize);
12                 bitstring.Append(tcbt.ToBitstring());
13                 diffSize ← 0;
14                 tcbt.Clear();
15             else
16                 bitstring.Append(cbt-diff.ToBitstring());
17                 diffSize += sizeof(cbt-diff);
18         indexes.Append(sizeof(bitstring));
19     return indexes.ToBitstring() + bitstring;
```

(lines 6 and 7, respectively) to compress each time frame into a CBT′ or a span of time frames into a T-CBT. Notice that while we appear to be running with complexity $n \times \tau$, the nested loops only perform real computation when there is a change from the previous frame. We have only written it this way because frames without any changes still require a zero bit. The window is maintained in lines 8 and 9. Then, on line 10, it checks to see if the W most recent CBT′s could be more efficiently represented with the single T-CBT. If so, it removes the previous CBT′s (line 11) and appends the already calculated T-ABT (line 12). Otherwise, on line 16, it appends the latest CBT′. While doing all of this, it also keeps track of the index position and lifetime of each T-CBT. Finally, it returns

the list of indexes appended with the list of compressed nodes on line 19. For our index's integer encoding scheme, we use Delta Encoding [41].

As described above, the first frame, t_0, will be a regular CBT representing the entire graph at time t_0. Then, while we are building and appending CBT′s, we are also maintaining the current T-CBT that spans those CBT′s within some window W. Thus, when our inequality $\sum_i^j (Size(CBT(t_i))) > Size(\text{T-CBT}(i,j))$ is satisfied, we already have the T-CBT ready and can replace the CBT′s with it. Note that this representation is also needed to know what the T-CBT's actual size is for the inequality. Otherwise, we would have to approximate.

Compressing directly from a gzip compressed arc list

During our experiments, we encountered such large graphs that even the raw arc list format required over 20GB of RAM. Since most computers these days do not have access to large amounts of main memory, we devised a way to compress directly from a much smaller gzipped file. It is important to note that the gzipped file must also be sorted.

The technique uses the zLib library [34] that gzip is built on. This library allows us to partially inflate (decompress) the file in chunks. Since the file is sorted, we can decompress a single node before we recompress it with our method. Obviously, this technique only affects compression time and memory required for compression.

4.4.5 Analysis

Recall that a graph has n nodes, m static arcs, and c' differential contacts. Each differential contact is stored as a 1 in the 3D Boolean matrix, with the rest of

the entries being 0.

Lemma 4.3: Given a time-evolving graph with $n = 2^k$ $(k > 0)$, c' differential contacts, a lifetime of τ, an average differential contact per frame $\delta = c'/\tau$, and an average node-differential contact degree of $\alpha = \delta/n$, a single node in a frame from the time-evolving graph can be encoded with a total space of $\alpha(\log_2(\tau n^2/c')) + O(\alpha)$ bits.

Proof: Using Lemma 2.2 and the graph conditions above, a compressed node would yield a total of $k\alpha + 3\alpha$ bits. However, not all of the paths to each differential contact may be unique. We can see that the worst case is when every node in the tree is present up to depth $\log_2(\alpha)$. After that level, the worst case follows with each path to the α differential contacts being unique, giving a total space of $\sum_{j=1}^{\lfloor \log_2 \alpha \rfloor}(2^j) + \alpha(k - \lfloor \log_2 \alpha \rfloor - 1) + 3\alpha = \alpha(\log_2(\tau n^2/c')) + O(\alpha)$ bits. ■

Lemma 4.4: Given a time-evolving graph with $n = 2^k$ $(k > 0)$, c' differential contacts, a lifetime of τ, an average differential contact per frame $\delta = c'/\tau$, and an average node-contact degree of $\alpha = \delta/n$, a single frame can be compressed with $\delta \log_2(\tau n^2/c') + O(\delta)$ bits, or $\delta \log_2(\tau n(n-1)/2c') + O(\delta)$ bits if we are only using the upper triangular matrix.

Proof: Given the graph conditions above, we must use the formula in Lemma 4.3 for all n nodes, giving $n(\alpha(\log_2(\tau n^2/c'))) + O(\alpha) = \delta \log_2(\tau n^2/c')) + O(\delta)$. ■

Proposition: Given a time-evolving graph with $n = 2^k$ $(k > 0)$ nodes, c' differential contacts, and a lifetime of τ, the information-theoretic lower bound for storage is $c' \log_2(\tau n^2/c') + O(c')$ bits [18].

Theorem 4.1: Given a time-evolving graph with $n = 2^k$, c' differential contacts, a lifetime of τ, an average differential contact per frame $\delta = c'/\tau$, and an average node-contact degree of $\alpha = \delta/n$, the entire time-evolving graph can be compressed with $c' \log_2(\tau n^2/c') + O(c')$ bits, or $c' \log_2(\tau n(n-1)/2c') + O(c')$ bits if we are only using the upper triangular matrix, thereby achieving the information-theoretic lower bound.

Proof: Given the graph conditions above, we must use the formula in Lemma 4.4 for each of the τ time frames, giving $\tau(\delta(\log_2(\tau n^2/c'))) + O(\delta) = c' \log_2(\tau n^2/c')) + O(c')$. ■

A 3D-plot of our upper bound for space requirements is given in Figures 4.5 and 4.6 with set values of τ.

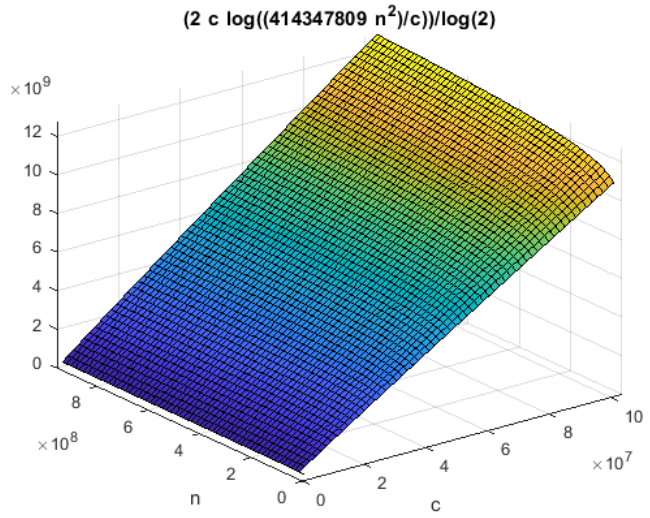

Figure 4.5: 3D-plot of space (bits) requirements given n (nodes) and c (differential contacts) with $\tau = 414347809$

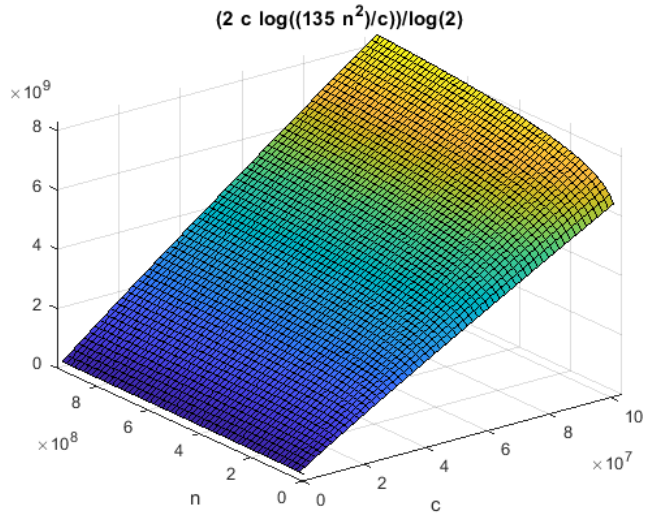

Figure 4.6: 3D-plot of space (bits) requirements given n (nodes) and c (contacts) with $\tau = 135$

4.4.6 Proof of correctness

Theorem 4.2: Given a time-evolving graph $G_\tau = G_0, G_2, ..., G_{\tau-1}$, Algorithm 4.3 provides a lossless compression of G_T.

Proof: If $G'_i = G_i - G_{i-1}$ and $G'_0 = G_0$, where $\tau > i > 1$, then $G_0 + G'_1 = G_1$ and thus $G_t = \sum_{i=0}^{t} G'_i$, where $\tau > t > 1$. Therefore, $\{G'_i\} = G_\tau$, where $\tau > i \geq 0$.

Now, since $\{ABT'_i\} = \{G'_i\}$ and $TABT'_{i,j} = \{ABT'_k\}$, for all $\tau > j \geq k \geq i \geq 0$, our algorithm is correct. ■

4.4.7 Supported operations

We provide four operations that can be performed on our structure: checking arc existence at a given time t, getting a node's neighbors at t, adding/removing arcs at t, and appending/removing frames. While these are all separate operations, they all involve knowing how to efficiently traverse the compressed tree in bit-string form, which we have described for regular ABTs in [100] and for T-ABTs

in [101]. In Algorithm 4.4, we describe how to decode a CBT′, which is the final tool needed to run our supported queries.

Algorithm 4.4: Differential CBT - Decoding

Input: CBT_{t-1} and CBT'_t
Output: CBT_t

```
1  begin
2      Node node_tm1 = cbt_tm1.Root , node_t = cbt_t.Root;
3      Visitor vtr_tm1 = PreOrderTraversal(node_tm1) , vtr_t =
        PreOrderTraversal(node_t);
4      while !vtr_t.End() do
5          if node_t.Label == 1 then
6              if node_tm1.Label == 0 then
7                  node_tm1.Label ← 1;
8                  vtr_tm1.Expand(node_tm1);
9              else if node_t.isLeaf() then
10                 node_tm1.Label ← !node_tm1.Label;
11         else
12             vtr_tm1.Ignore(node_tm1);
13         node_tm1 = vtr_tm1.VisitNext(node_tm1);
14         node_t = vtr_t.VisitNext(node_t);
15     return vtr_tm1.ToBitstring();
```

Algorithm 4.4 takes as input a CBT representing $A_{i,t-1}$ and a CBT′ representing $A'_{i,t} = |A_{i,t} - A_{i,t-1}|$, for some $i \in V$. It outputs the CBT after its CBT′ is applied, thus representing $A_{i,t-1}$. For algorithmic simplicity, we assume that the CBTs are using the original encoding scheme. However, all of experiments use the improved version. We start by traversing both CBT and CBT′ at the same time. At each node in the preorder traversal we check to see if CBT′ indicates a change, as in line 5. If so, we have two cases. If the current CBT's node is labeled with a zero, then we must set it to one and expand he node (lines 6-8). If the current CBT's node is a leaf node, we flip the node's label (lines 9-10). Line 11 means that CBT′ did not indicate a change in the node, and should therefore

be ignored. We return the new CBT in line 15.

Arc query

We shall describe Algorithm 4.5 for checking arc existence (u, v) at a time t in our structure. It begins with lines 2-4 loading the indexes and using them to navigate to u's starting position. Then, on line 5, we begin decoding each T-CBT (line 6) or each CBT$'$ (line 11) in chronological order while also maintaining the latest frame of the node on lines 4, 10, and 12. If the current CBT is a T-CBT (line 6), then we check to see if it spans our selected time frame (line 8). If so, we return whether or not the arc is active on line 9. If the current CBT was a CBT$'$ (line 11), then we apply the CBT$'$ to the current CBT (line 12). Then, if the current CBT spans t (line 13), we return whether the arc exists (line 14). This process is the same as in Algorithm 4.4, except we return true or false when we decode the node representing the requested arc.

Neighbor query

The neighbor query is the same as the arc query, except that it decodes the entire node and returns the active arcs.

Streaming arcs

Streaming arcs involve being able to activate or deactivate an arc in any time frame. Therefore, we present Algorithm 4.6 which assumes that $\text{CBT}_{u,t-1}$ has already been decoded, given a time t and an arc (u,v). The algorithm also assumes that the streaming operation is an arc addition rather than a removal. This assumption is merely for ease of algorithmic reading, as the removal operation is the mirror of addition. Additionally for algorithmic simplicity, we again assume

Algorithm 4.5: Time-Evolving Differential ABT Compression - Arc-Time Query

```
   Input: The compressed time-evolving graph as a bit-string s, u, v, and t
   Output: True or False
1  begin
2      indexes ← s.GetIndexes();
3      position ← indexes.GetStart(u);
4      currentCBT ← null;
5      while true do
6          if s.IsTCBT(&position) then
7              tcbt ← s.GetTCBT(&position);
8              if tcbt.Spans(t) then
9                  return tcbt.Contains(v, t);
10             currentCBT ← tcbt.LastFrame();
11         else
12             currentCBT.Apply(s.GetCBTDiff(&position));
13             if currentCBT.Spans(t) then
14                 return currentCBT.Contains(v);
```

that the CBTs are using the original encoding, rather than our improved one. However, our experiments are coded using the improved version.

Algorithm 4.6 is similar to Algorithm 4.4 in the sense that we must traverse multiple trees at the same time. We must additionally traverse and modify CBT'_{t+1} since a change in t will affect our differential encoding for $t+1$. During this traversal, we only care about nodes that span over the targeted arc (line 5); we ignore them otherwise (lines 25-27). When we encounter a spanning node, we check if CBT'_t or CBT'_{t+1} needs to be expanded (lines 6-11), which we continue to do until we reach a leaf node (line 12). Once we reach the leaf nodes of both CBT'_t and CBT'_{t+1}, we set our labels and recursively update/compress the branch if the label was a zero (lines 15 and 20). Finally, since we have reached the leaf node level, the algorithm is finished and we return the updated CBT'_t and CBT'_{t+1} on

Algorithm 4.6: Differential CBT - Streaming an arc

Input: CBT_{t-1}, CBT'_t, CBT'_{t+1}, and the arc to be added, y
Output: The new CBT'_t and CBT'_{t+1}

```
1  begin
2      Node node_tm1 = cbt_tm1.Root, node_t = cbt_t.Root, node_tp1 =
         cbt_tp1.Root;
3      Visitor vtr_tm1 = PreOrderTraversal(node_tm1), vtr_t =
         PreOrderTraversal(node_t), vtr_tp1 = PreOrderTraversal(node_tp1);
4      while !vtr_t.End() do
5          if node_tm1.Spans(y) then
6              if node_t.Label == 0 then
7                  node_t.Label ← 1;
8                  vtr_t.Expand(node_t);
9              if node_tp1.Label == 0 then
10                 node_tp1.Label ← 1;
11                 vtr_tp1.Expand(node_tp1);
12             if node_t.isLeaf() then
13                 if node_tm1.Label == 1 then
14                     node_t.Label ← 0;
15                     vtr_t.RecursivelyUpdate();
16                 else
17                     node_t.Label ← 1;
18                 if node_tp1.Label == 1 then
19                     node_tp1.Label ← 0;
20                     vtr_tp1.RecursivelyUpdate();
21                 else
22                     node_tp1.Label ← 1;
23                 return vtr_t.ToBitstring(), vtr_tp1.ToBitstring();
24         else
25             vtr_tm1.Ignore(node_tm1);
26             vtr_t.Ignore(node_t);
27             vtr_tp1.Ignore(node_t);
28         node_tm1 = vtr_tm1.VisitNext(node_tm1);
29         node_t = vtr_t.VisitNext(node_t);
30         node_tp1 = vtr_tp1.VisitNext(node_tp1);
31     return;
```

line 23.

Streaming frames

Our final operation of streaming frames applies to adding/removing frames to/from the end of the time-evolving graph. The appending operation is a trivial modification of the construction process of Algorithm 4.3. That is, adding a frame includes inserting or appending a CBT′, and then converting the resulting series of CBT′s to a T-CBT if needed. When removing a frame, if the frame is a CBT′, then we simply remove it and update the next frame. If the frame is spanned by a T-CBT, we break up the T-CBT into CBT′s and perform the same process.

4.5 Experiments and results

Our experiments involve comparing CBT′ compression sizes against various time-evolving compression techniques, particularly ck^d-tree [23]. We compare using the time-evolving graphs in Table 4.2. The original results in [23] were in bpc (bits per contact), but we have converted these compression rates to their final compressed sizes (e.g., megabytes). This is because their definition of a contact is not useful to us as we compress against a 3D representation rather than a 4D one. We now explain each dataset.

The Comm.Net* and Powerlaw* graphs are synthetic and recreated according to the specifications in [23]. Comm.Net* represents short communications between random vertices. Powerlaw* is a power-law degree graph, where few vertices have many more connections than other vertices, but with a short lifetime. We star(*) these graphs because they are not exactly equal to the original graphs, but should yield comparable graphs.

	Type	|V|	|E|	Lifetime	Contacts
I-Comm.Net*	Interval	10000	15940743	10001	19061571
I-Powerlaw*	Interval	1000000	31979927	1001	32280816
I-Wiki-Links	Interval	22608064	564224135	414347809	731468598
I-Yahoo-Netflow	Interval	103661224	321011861	114193	955033901
G-Flickr-Days	Incremental	2585570	33140018	135	33140018
P-Wiki-Edit	Point	21504192	122075170	304002801	266720840

Table 4.2: The dataset stats

	.txt	.txt.gz	$\text{CBT}'_{\%BEST}$	CBT′	ck^d-tree	CAS	CET	TGCSA
I-Comm.Net*	198.3MB	66.2MB	-3.2%	60.1MB	62.0MB	117.2MB	131.0MB	145.8MB
I-Powerlaw*	611.5MB	157.6MB	-4.5%	123.2MB	128.7MB	312.7MB	388.2MB	297.8MB
I-Wiki-Links	16.7GB	4.2GB	+0%	3.2GB	5.6GB	3.2GB	5.3GB	6.1GB
I-Yahoo-Netflow	21.5GB	6.7GB	-4.1%	4.9GB	5.1GB	5.8GB	7.5GB	7.5GB
G-Flickr-Days	776MB	81.0MB	-1.0%	77.1MB	95.3MB	77.9MB	555.9MB	209.6MB
P-Wiki-Edit	6.6GB	1.7GB	+0%	1.3GB	1.4GB	1.4GB	1.3GB	2.4GB

Table 4.3: Compressed graph sizes

Confidence Level = 95%	Compression (hr)	arc_t (μs)	$Neighbor_t$ (μs)	$Stream_t$ (μs)
I-Comm.Net*	2.7m	152.782 ± 132.706	158.722 ± 129.506	185.050 ± 140.876
I-Powerlaw*	5.2m	121.231 ± 99.310	131.243 ± 98.021	156.661 ± 104.384
I-Wiki-Links	1.8h	463.242 ± 337.237	469.517 ± 320.751	478.380 ± 343.446
I-Yahoo-Netflow	2.1h	395.575 ± 235.523	402.236 ± 200.662	412.760 ± 237.522
G-Flickr-Days	5.1m	93.543 ± 40.957	95.399 ± 36.004	105.997 ± 19.890
P-Wiki-Edit	31.1m	225.734 ± 196.512	229.616 ± 184.517	245.673 ± 203.357

Table 4.4: CBT′ execution times

The Flickr-Days dataset is an incremental time-evolving graph with time granularity set to days from 11-02-2006 to 05-18-2007. It represents when users became friends in the Flickr Social Network and can be found at http://socialnetworks.mpi-sws.org/data-www2009.html.

The Wiki-Links (http://dumps.wikimedia.org/enwiki/) interval dataset shows the history of links among articles in the English version of Wikipedia. It has time granularity by second since 03-04-2014. Additionally, Wiki-Edit (http://konect.uni-koblenz.de/) is a bipartite, point-contact graph indicating when a user edits a Wikipedia article. It has time granularity of seconds since the creation of Wikipedia.

Finally, we have the Yahoo-Netflow interval graph which contains communication records between end users in the large Internet and Yahoo servers. It has time granularity by seconds starting at 04-29-2008 and can be found at http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

All datasets are anonymized and require preprocessing. For example, the Yahoo-Netflow dataset contains timestamp, source IP address, destination IP address, source port, destination port, protocol, number of packets, and number of bytes transferred from the source to the destination. The IP addresses were already anonymized with a random permutation algorithm. We also only needed timestamp, source, and destination information, which were converted to frames and vertices, respectively. Additionally, since we pulled both of the Wikipedia dumps after the work in [23], we had to remove records until they matched the original graphs.

We run all of our algorithms on a machine with an Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (4 cores) with 64GB of RAM.

4.5.1 Compression

While examining the compression results, we refer to Table 4.3. The first three columns describe the dataset with its name, size as a raw text file, and size of the text file compressed with gzip. Note that these text file sizes are after we have performed our preprocessing on the graphs. The details of each dataset graph are reported in Table 4.2. The next column gives a percentage of CBT′ with the next best compression method, for easy reading. The following columns represent the size of the graph after each compression technique has been applied.

The results show that CBT′ compresses as well or better than the other techniques on all datasets. When examining this, we see that we achieve better space than the ck^d-tree in all cases. This makes sense, as our trees are similar, but our technique makes use of differential encoding throughout time, giving strictly better compression rates. This also explains why our technique outperforms CAS, even on Wiki-Links and G-Flickr-Days. These graphs contain more active arcs over time, which is a positive case for both CAS and CBT′. CAS no longer needs to store the neighboring changes that deactivate arcs at the end of the lifetime [23] [22], and our compression becomes mostly CBT′s.

For P-Wiki-Edit, which is a point-contact graph, we adjusted our CBT′s such that an arc is automatically assumed to disappear in the next frame unless repeated again. Similar to G-Flickr-Days and I-Wiki-Links, the P-Wiki-Edit graph also compresses to mostly these modified CBT′. It is only when users edit multiple pages many times that it becomes more space-efficient to use T-CBTs. This is because the entire branch has to be constructed each time, rather than once with a time dimension as in [101].

4.5.2 Operation times

Next, we examine the various operation times of CBT′. Since we did not have access to our benchmarks' code, nor an equivalent machine, we can only compare our operation times amongst themselves. When discussing these operation times, we refer to Table 4.4.

The first column indicates the time it took for our technique to compress the graphs. We can see that this time is somewhat proportional to the size of the graphs in terms of contacts. Again, note that our technique does not require any intermediate structure to compress. Thus, our compression times are not affected by access times of these intermediate structures and only depend on the time it takes to read the file and build the trees.

The following columns represent running times of different operations on the compressed graph. These include the time to query an arc at time t (arc_t), retrieving all neighbors of a node at t ($Neighbor_t$), and streaming/activating arcs at t.

Each of these operation times is the average of 1000 queries with random parameters. All these times are measured in microseconds with a confidence level of 95%.

Before examining the times, realize that if we are searching through a T-CBT, regardless of whether we are interested in an arc, once we have reached it, we have no choice but to read its corresponding time tree before we can continue with reading the rest of the tree for the aggregated matrix. This explains why we have high variance on our arc_t and $Stream_t$ operations. This can be avoided by including a number to indicate how long the time tree is, but this increases space required.

We see that query times are mostly affected by τ, which also results in higher variance. arc_t is inherently the fastest of the queries, as it returns as soon as it finds the requested arc at a given time frame. We also manage to keep $Stream_t$ somewhat similar to arc_t by using a modified unrolled linked list to allow for faster bit insertion/deletion.

4.6 Conclusion

In this chapter, we have introduced a novel time-evolving web and social network compression using differential compressed binary trees. We also maintain minimal memory overhead while compressing and provide arc, neighbor, and streaming algorithms. We also provide analysis for our compression which proves that our technique requires $c \log_2(\tau n^2/c)) + O(c)$ bits of space, which is asymptotically twice the theoretical minimum. We compare this technique using several real-world datasets against several benchmarks from [23]. We find that our technique performs as well or better than the best benchmark compression in every dataset, and we provide description and analysis of the results.

Chapter 5

Algorithms on compressed time-evolving graphs

Time-evolving graphs are structures that encapsulate how a graph changes over time. Thus, we not only have to deal with large graphs consisting of nodes and arcs in the billions, but we must also keep track of when these arcs activate and deactivate over long lifetimes. In this age of big historical data, we must make use of efficient time-evolving graph compressions, or we will find ourselves quickly out of main memory.

These time-evolving graph compressions must not only be space-efficient, but must also facilitate fast querying directly on the compressed graph. In this chapter, we define several novel time-evolving graph problems and develop algorithms to solve them directly on various, massive, synthetic and real-world time-evolving graphs compressed using our technique. Our experiments provide details of the compressed graph sizes, algorithm run-times, and other metrics.

5.1 Introduction

In this chapter, we adapt our time-evolving compression from Chapter 4 to run time-evolving algorithms. Our motivation is similar to that of Chapter 3, but we instead claim that it is also desire-able to run algorithms directly on the compressed *time-evolving* graphs. We compute the earliest arrival paths, transitive closure, incremental transitive closure, and our own novel definition of time-evolving transitive closure. We provide the definition and motivation for each of these problems in Section 5.2.2.

Our contributions can be summed up as:

- We compress time-evolving graphs in a queryable and streaming structure and provide different ways of ordering the structure to better accommodate certain algorithms.
- We define our novel problem of time-evolving transitive closure.
- We provide an algorithm to solve the existing problem of finding earliest arrivals paths.
- We provide an algorithm to solve our novel problem of time-evolving transitive closure. In doing so, we also provide algorithms for transitive closure and incremental transitive closure.
- All algorithms keep all data compressed to give efficient run-times and memory usages.
- We provide a detailed empirical study that uses real, massive time-evolving graphs and report the findings on our algorithms' run-times.

The rest of the chapter is organized as follows. In Section 5.2, we define time-evolving graphs and the problems that we will be solving on them. We describe our compression technique using compressed binary trees in Section 5.3. We define the algorithms to run directly on our compressed graph in Section 5.4. In Section 5.6, we discuss related work. Finally, we report empirical results in Section 5.5 and conclude in Section 5.7.

5.2 Preliminaries

In this section, we define time-evolving graphs and the operations that can be performed on them.

5.2.1 Time-evolving graphs

As described in Chapters 1 and 4, a time-evolving graph is a graph where arcs appear and disappear as time passes [103]. A static graph (snapshot) can be represented as a graph $G = (V, E)$, where V is a set of nodes and E is a set of arcs. Thus, a time-evolving graph would be a series of graphs $G_T = G_1, G_2, ..., G_\tau$, where τ is the number of time frames, i.e., the *lifetime*. Since a simple graph $G = (V, E)$ can be represented as a 2D matrix, then a time-evolving graph could be represented as a 3D matrix, also known as a *presence matrix* [47].

When downloading a time-evolving graph off the web, we usually find them represented as a *contact list* [23]. A contact is a 4-tuple (u, v, t_i, t_j) meaning that the edge (u, v) existed (inclusively) between the time frames t_i and t_j, where $0 \leq i < j < \tau$ and τ is the lifetime of the graph. Before we compress, we pre-process this input to be a list of 3-tuples (u, v, t) indicating that the edge (u, v) changes state at time $0 \leq t < \tau$. Thus, every 4-tuple contact (u, v, t_i, t_j) becomes

two triplets, (u, v, t_i) and (u, v, t_j).

We also notice that representing a time-evolving graph as a static graph (by union-ing all arcs) loses temporal information that is vital in understanding certain properties of the time-evolving graph. For example, if we have two temporal arcs $(i, j, 1)$ and $(j, k, 0)$, a union-ed version of the graph would show that i can reach k through j. However, such a path is not actually possible in the time-evolving graph because it would require going backwards in time from frame $t = 1$ to $t = 0$.

5.2.2 Problems on time-evolving graphs

In this section, we define several problems of interest on time-evolving graphs. We address these problems by developing algorithms to run directly on the compressed time-evolving graph. In fact, we keep all data compressed throughout every algorithm, giving efficient run-times and memory usages.

Earliest arrival paths

We start by defining what a path is on a time-evolving graph, since a path that travels through time does not translate directly from a path in a static graph.

Definition 5.1 (*Time-evolving path*). Note that given a time-evolving graph $G_T = G_1, G_2, ..., G_\tau$, each arc $e \in \bigcup_{i=1}^{\tau}(E(G_i))$ can be represented as a triplet $e = (u, v, t)$, meaning that u is connected to v at time t. Thus, we define a time-evolving path as a series of arcs $(e_1, e_2, ..., e_k)$, where $v(e_i) = u(e_{i+1})$, $t(e_i) \leq t(e_{i+1})$, and $1 \leq i \leq k$. In other words, the path can not go backwards in time.

We see that this definition preserves temporal information throughout the path.

Definition 5.2 (*Earliest arrival path*). A path $p \in P$ is an earliest arrival path if for all paths $p' \in P$, $t_{end}(p) \leq t_{end}(p')$, for some time range $[t_{start}, t_{end}]$. Thus, we are given a source vertex s, and a time interval $[t_{begin}, t_{end}[$ and we output a path to every other vertex in G.

A classic motivational example for the earliest arrival paths problem is to find the earliest at which you can arrive at any airport given some source airport. We provide an algorithm to solve this problem in Section 5.4.1.

Time-evolving transitive closure

Definition 5.3. (*Transitive closure*). Given a static graph $G = (V, E)$ with $|V| = n$, $|E| = m$, we aim to output an $n \times n$ matrix where $C(u, v) = 1$ iff v is reachable from u.

Definition 5.4. (*Time-evolving (dynamic) transitive closure*). Assume we are given a time-evolving graph $G_T = G_1, G_2, ..., G_\tau$, where τ is the number of time frames. A time-evolving transitive closure is a structure that efficiently gives the transitive closure of G_T at any time frame in the interval $[t_{begin}, t_{end}]$.

Not all graphs are just one massive strongly connected component (SCC) [35]. Time-evolving graphs that contain many SCCs may add edges over time which change the transitive closure. Thus, it would be useful to see how a graph's transitive closure changes over time.

Clearly, the graph at each time frame can have a different C value. A naïve approach would be to calculate C_i for each G_i. However, a more interesting and efficient approach would be to use $G_j - G_i$, where $i < j$, to update C_i, giving us C_j. Thus, our input is G_T and a time interval $[t_{begin}, t_{end}]$. So we can see that not only do we have to develop efficient transitive closure algorithms, but we would also like to reuse a time frame's previously computed transitive closure

when computing the transitive closure for the next time frame.

Using the airport example again, time-evolving transitive closure shows us which airports are reachable to each other. While they may all be reachable to each other given an unlimited amount of time, airport B may not be reachable from airport A during the time frame $[t_i, t_j]$. We provide algorithms that address these problems in Sections 5.4.2, 5.4.3, and 5.4.4.

5.3 Time-evolving graphs as implicit differential compressed binary trees

As described in Section 5.2.1, a time-evolving graph can be represented as a $n \times n \times \tau$ 3D matrix, where $n = |V|$. Additionally, it can be represented as a series of graphs $G_0, ..., G_t, ..., G_{\tau-1}$, where $G_t = (V_t, E_t)$ and $0 \leq t < \tau$.

Our compression represents each node u ($0 \leq u < n$) at each time frame t ($0 \leq t < \tau$). In other words, we compress the adjacency row representing the neighbors of u at each time frame, *differentially*. In Figure 5.1 (a), we have a graph of size $n = 6$ with $\tau = 3$. Then, for each u, we represent each row $A_{i,t}$ with differential compressed binary trees as shown in Figure 5.1 (b). That is, the compressed binary trees only represent neighbor changes since *the previous time frame*. We describe this more in Section 5.3.2. We can see that our structure requires 90 bits, whereas the 3D matrix representation needs 108 bits. For undirected graphs, our binary tree would only represent the 3D upper triangular matrix; however we would need a larger example to illustrate the space benefits.

Our compression technique for time-evolving graphs is as follows:

- Compressed binary trees (CBT) are implemented with an improved encod-

ing to better compress *consecutive arcs* and *branches that only contain one arc* (Section 5.3.1).

- Given a node i at time frames $t-1$ and t where $1 \leq t < \tau$, the adjacency row $A_{i,t}$ is encoded differentially from $A_{i,t-1}$. That is, a CBT is encoded to represent the absolute difference between the adjacency rows $|A_{i,t} - A_{i,t-1}|$. Only changes since *the previous time frame* are encoded (Section 5.3.2).
- The structure is outputted as a series of CBTs in one of many possible orderings (Section 5.3.3).

5.3.1 Compressed binary trees

In this section, we describe compressed binary trees with the improved encoding techniques illustrated in Figures 5.2 and 5.3. In Figure 5.2, we have a row A_i at a time t compressed with a binary tree using an improved encoding scheme. Each node in the tree spans its corresponding range of indexes (e.g., the root spans $[0, n-1]$) and contains a bit indicating whether or not that index range contains a one.

We can see that the branches containing only zeros are pruned off and compressed with a single zero bit. Now, notice that if a node is marked with a one, it should normally never have its two children both marked with zero [23]. We take advantage of this fact by saying that if a node is followed by two children marked with zeros, then an additional bit must follow. If this next bit is zero, then the branch is filled with all ones. If the bit is one, then the branch leads to a single arc, as in Figure 5.3.

In Figure 5.3, we have row A_i again, but at time $t+1$ with one additional arc at position 28. This row is encoded with a CBT *differentially* from the CBT of $A_{i,t}$

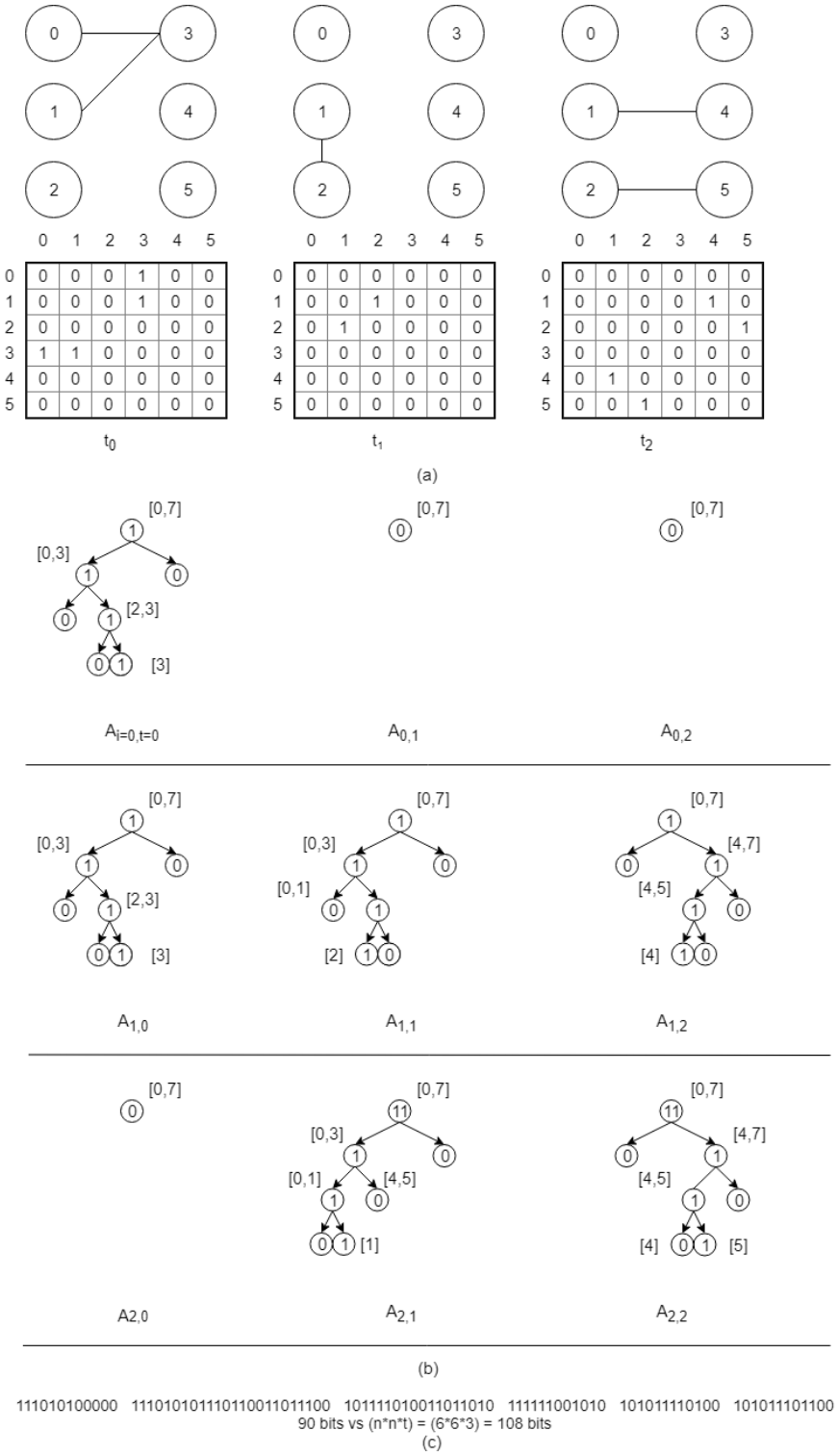

Figure 5.1: A time-evolving graph with $n = 6$ and $\tau = 3$ represented as a series of differential compressed binary trees

in Figure 5.2. We explain this more in Section 5.3.2. This example demonstrates when a node's two children are both marked zero, and the following bit is a one. In this case, it means that the current branch leads to a single arc to which we then provide a direct binary path, relative to the current branch. That is, while the example's change is encoded at the root of the tree ($d = 0$), it could occur at any depth $d < \log_2(n) - 3$ with a saving of $\log_2(n) - 2d - d - 3 = \log_2(n) - d - 3$ bits. We now describe Algorithm 5.1 which outputs a relative binary path given a target index j and a node's beginning and ending range.

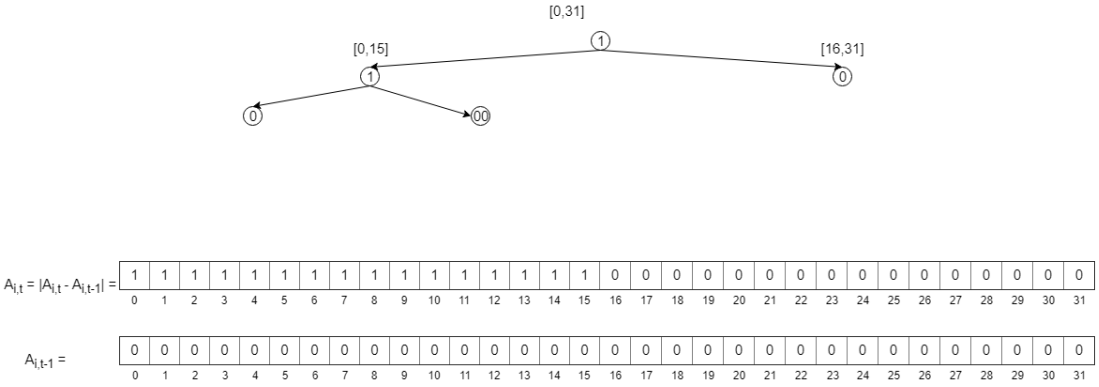

Figure 5.2: A CBT of row A_i at time t

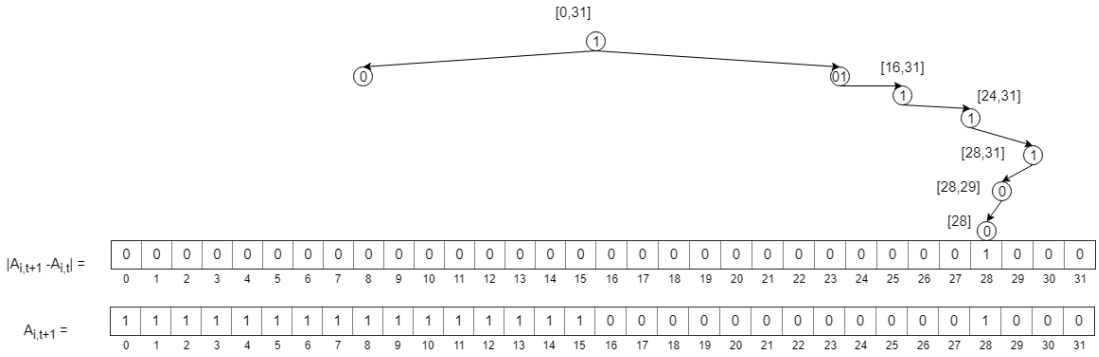

Figure 5.3: A CBT′ with a single arc at position [28] being added to the CBT from Figure 5.2

In Algorithm 5.1, we are given *begin* and *end* which represents the range of our current node, along with the target index j. We calculate the relative depth

Algorithm 5.1: A relative binary path

```
   Input: int begin, int end, int j
   Output: The relative binary path as a bit-string
1  begin
2      BitString s;
3      depth = ⌈log_2 (end − begin)⌉;
4      s.Initialize(depth);
5      for i = 0 to depth − 1 do
6          mid = ⌈(begin + end)/2⌉;
7          if begin ≤ j < begin + mid then
8              s.AppendBit(0);
9              end − = mid;
10         else
11             s.AppendBit(1);
12             begin + = mid;
13     return s;
```

in line 2, which will be the length of our returned bit-string. In lines 5 through 12, we loop through each level of the remaining depth, and append a one or a zero, depending on which child the path should navigate to (left = 0, right = 1). We return the path's bit-string in line 13.

5.3.2 Differential compressed binary trees

In Figures 5.2 and 5.3, the adjacency rows that the CBTs represent are obtained from $|A_{i,t+1} - A_{i,t}|$. This means that an entry v in the resulting row $A'_{i,t+1} = |A_{i,t+1} - A_{i,t}|$ indicates that $A_{i,v,t} \neq A_{i,v,t+1}$. In a differential CBT, a zero indicates that the branch does not contain any change, and a one indicates that the branch does contain a change. We now describe Algorithm 5.2 which also includes the improved encoding scheme described in Section 5.3.1.

In Algorithm 5.2, we are given as input a node's compressed binary tree (CBT) and a list of arc targets. If a target is already a neighbor of the node, it is to

Algorithm 5.2: Differential binary tree compression

Input: $\text{CBT}_{i,t}$, and a list of arc change targets
Output: $\text{CBT}'_{i,t+1}$ as a bit-string

```
1  begin
2      BitString s;
3      Node node = cbt.Root;
4      Visitor vtr = PreOrderTraversal(node);
5      while !vtr.End() do
6          nodeTargets = targets.Where(x => node.Spans(x));
7          if nodeTargets.Count() > 0 then
8              s.AppendBit(1);
9              if node.arcs() == nodeTargets then
10                 s.AppendBitString('000');
11                 vtr.Ignore(node);
12             else if nodeTargets.Count() == 1 then
13                 s.AppendBitString('001');
14                 path = RelativeBinaryPath(node.begin, node.end, target);
15                 s.AppendBitString(path);
16                 vtr.Ignore(node);
17         else
18             s.AppendBit(0);
19             vtr.Ignore(node);
20         node = vtr.Next();
21     return s;
```

be removed. We use the visitor pattern and some LINQ notation [9] for ease of reading. We start by traversing through the CBT representing the node's current neighbors. On line 6, we get the list of target changes pertaining to the current node. If there are changes in this branch, we have three cases: (1) the branch is removed entirely, (2) the branch has one change, or (3) the branch has many changes. The last case is handled simply by line 8. The first case makes use of our new encoding scheme in line 10 by appending '000', indicating that the entire branch has been removed. The second case handled in lines 12 through 16 also uses a new encoding scheme by appending '001', followed by the relative binary

path to the target change. If there were no changes for this node, then we simply append a zero, as in line 18. We return the differential CBT as a bit-string in preorder traversal in line 21. The bit-string for Figures 5.2 and 5.3 would be 110000 and 100111100, respectively.

Algorithm 5.2 is an implicit compression [11], and since it uses our list of triplets (described in Section 5.2.1), it gives us a differential compression of our time-evolving graph. Depending on which algorithm we want to solve, we can order these triplets by u, then t, then v (for earliest arrivals) or by u, then v, then t (for transitive closure). These orderings will give us either a node-by-node or a snapshot-by-snapshot compression, respectively.

We also notice that for graphs with many vertices, long lifetimes, and few arc changes, we might be spending too many bits to describe that a node's neighbors did not change for long intervals. For example, assume $|V| = \tau = 10^6$, and most nodes' neighbors only have a few changes throughout time. Then we are essentially spending 10^{12} bits to say that the nodes did not change much throughout their lifetime.

We can handle this by layering another compressed binary tree spanning τ where each leaf node corresponds to the root of each differential. Thus, consecutive empty differentials as well as consecutive non-empty differentials are pruned off.

5.3.3 Ordering compressed binary trees

Our differential compressed binary trees technique supports many different ordering schemes. We now describe some orderings and how they are beneficial to our algorithms.

Node-by-node

In this ordering scheme, we enumerate a node's neighbors throughout each time frame before enumerating the next node. We can do this either row-by-row or column-by-column.

In Figure 5.1, row-by-row would mean that we read the binary trees left-to-right then top-to-bottom. In other words, we would have $A_{0,0}, A_{0,1}, ..., A_{1,0}, A_{1,1}, ...$ for our enumeration. Column-by-column would be the transpose of this, where the CBTs are encoded against the columns of the matrix.

This ordering technique lends itself to our earliest arrival algorithm in Section 5.4.1. If we were using the snapshot-by-snapshot ordering as in Section 5.3.3, we would either have to read irrelevant, unreachable nodes that might not be in the same temporal connected component or we would have to inflate the size by providing indexes to each node at each time frame.

Snapshot-by-snapshot

As opposed to the node-by-node ordering, in Figure 5.1 snapshot-by-snapshot means that we would enumerate the trees top-to-bottom then left-to-right. In other words, $A_{0,0}, A_{1,0}, ..., A_{0,1}, A_{1,1}, ...$ Again, we can provide an equivalent column-by-column ordering on the transpose.

Both the row-by-row and column-by-column variants of this ordering facilitate our matrix multiplicaiton and transitive closure operations. If we were to use the node-by-node ordering in Section 5.3.3, then as we process the graph or increment the transitive closure, we would have to keep track of where the next time frame begins for each node.

5.3.4 Matrix-matrix multiplication

In this section, we describe an algorithm to perform Boolean Matrix-Matrix multiplication on our compressed binary trees. That is, we perform $A \times B = C$ where A, B, and C are matrices represented as CBTs. Using CBTs allows us to perform matrix-matrix multiplication with reasonable memory and that includes storing the output directly as CBT. Now, notice that this involves multiplying the rows of A with the columns of B, and that we also want to reuse C in place of B for the next iteration. This means we must encode A row-by-row, and B and C column-by-column. Thus, instead of multiplying to construct C row-by-row, we instead want to multiply such that we can build C column-by-column, top-to-bottom. This is as simple as looping through B's columns, *then* by A's rows, instead of vice versa. Also, since this is Boolean multiplication, multiplying vectors involves switching the multiplication with the AND operator, and the addition operation with OR. This means we gain the ability to short circuit the multiplication and return true as soon as we find the first true AND term. We now describe Algorithm 5.3, which multiplies two CBTs.

In Algorithm 5.3, for the sake of brevity, we only describe multiplication using the standard encoding. We take as input two CBTs representing our input vectors and we output true or false. We begin by traversing both trees at the same time using preorder traversal (lines 1-4). If either of the current nodes are labeled zero (line 5), then we know that they have no arcs in common and can thus ignore that entire branch for both trees (lines 6-7). However, if both nodes are labeled one and we have reached the bottom of the trees (line 8), then we short-circuit and return true for the multiplication (line 9). If we finish traversing either tree (line 4), then we have not satisfied any AND operator and we return false (line

Algorithm 5.3: CBT-CBT (Boolean vector-vector) multiplication

```
   Input: CBT a, CBT b
   Output: bool ab
1  begin
2      Node node_a = a.Root, node_b = b.Root;
3      Visitor vtr_a = PreOrderTraversal(node_a) , vtr_b =
         PreOrderTraversal(node_b);
4      while !vtr_a.End() ∧ !vtr_b.End() do
5          if node_a.Label == 0 || node_b.Label == 0 then
6              vtr_a.Ignore(node_a);
7              vtr_b.Ignore(node_b);
8          else if node_a.Label == 1 && node_b.Label == 1 &&
             IsMaxDepth() then
9              return true;
10         node_a = vtr_a.VisitNext(node_a);
11         node_b = vtr_b.VisitNext(node_b);
12     return false;
```

12).

We use Algorithm 5.3 in Section 5.4.2 to perform matrix-matrix multiplication.

5.4 Compressed time-evolving graph algorithms

Now that we have defined our problems and our compression technique, we describe our algorithms that work directly on our compressed structure to solve the defined problems.

5.4.1 Earliest-arrival time

In this section we describe our algorithm to compute the earliest arrival paths and times.

Algorithm 5.4: Earliest-arrival paths and times

Input: The compressed time-evolving graph G_τ, source vertex x, and the time interval $[t_{begin}, t_{end}]$

Output: A BFS tree parent array giving the earliest-arrival paths from x to every vertex $v \in V$ within $[t_{begin}, t_{end}]$

```
1  begin
2      Initialize t[x] = (−1, t_begin), t[v] = (−1, ∞) for all v ∈ V \ {x}, and a
         Queue Q with one element, x;
3      while !Q.Empty() do
4          y ← Q.Dequeue();
5          yNode = G_τ.GetNodeStart(y);
6          for t = 0 to t_end do
7              foreach v in yNode do
8                  if t ≥ t_begin and t[y].Second ≤ t
9                  and t < t[v].Second then
10                     t[v] ← (t, t);
11                     if t[v].First == −1 then
12                         Q.Enqueue(v);
13             yNode ← yNode.NextFrame();
14     return t[];
```

In Algorithm 5.4 we are given as input the complete time-evolving graph G_τ, a source node x, a time range $[t_{begin}, t_{end}]$, and we output a BFS tree as a parent array representing the earliest paths from x to all other nodes. For this algorithm, G_τ is ordered node-by-node. In other words, it starts with node 0 and all its time frames, then node 1 and all its time frames, and so on. This is opposed to the ordering needed in Algorithm 5.7, where we order snapshot-by-snapshot.

We begin in line 2 by initializing a parent array where each entry v is a pair containing the parent node u and the time-frame t of the arc used. Line 2 also initializes a queue Q starting with x, and line 3 begins looping over Q until it is empty. Thus, we are going to perform a variant of BFS such that once we dequeue a node y, we then process all of y's time dimension, up to t_{end} (lines 4-6). Since G_τ is a differentially compressed time-evolving graph, we must start at $t = 0$ (line 6) so that we can differentially compute y at time t_{begin}. In lines 8 and 9, when we reach t_{begin}, we are now considering an arc (y, v, t) as candidate in the earliest-path BFS. Thus, we must have a parent y that has already been reached at time t and t must be less than our previously selected arc for v. If we select the arc, then we update it in the parent array and enqueue v if it hasn't already been reached (lines 10-12). Finally, when Q is empty, we return the parent array $t[]$ in line 13.

5.4.2 Transitive closure

Recall that Algorithm 5.3 from Section 5.3.4 multiplies two CBTs together to output a 1 or 0 (true or false). In this section, we will apply Algorithm 5.3 and an addition algorithm to calculate the transitive closure of the graph.

While it has been shown that the transitive closure of a graph can be calcu-

lated in $O(n^{2.81})$ time by performing some preprocessing [96], we start this work using a version of the $O(n^3 \log_2(n))$ repeated multiplication algorithm [49]. The $O(n^{2.81})$ algorithm must be carefully considered in compressed graphs and we save this as a future work. Our algorithm also calculates the transitive closure in time $O(Min(\delta, n-\delta)n^2 \log_2^2(n))$ which is already much faster than $O(n^{2.81})$ in extremely sparse and dense graphs.

Also notice that although when calculating the transitive closure, matrix multiplication and addition are repeated up to $\log_2(n)$ times, we can also short-circuit this logic if the current iteration's result is the same as the previous iteration.

Algorithm 5.5: Transitive closure

```
   Input: CBT[] A, CBT[] B
   Output: CBT[] C
1  begin
2      Initialize a column-by-column CBT[] C;
3      for k = 0 to log(n) do
4          for j = 0 to n − 1 do
5              for i = 0 to n − 1 do
6                  product ← Multiply(A, B);
7                  if product then
8                      C[j].Stream(i);
9          C ← Add(B, C);
10     return C;
```

In Algorithm 5.5, we take as input the matrices A and B as CBT[]s and we output C as a CBT[]. We use a version of the $O(n^3 \log_2(n))$ repeated multiplication algorithm, but with compressed binary trees. On line 6, we use Algorithm 5.3 and on line 9, we use a similar algorithm that simply ORs two CBTs together. Line 8 is the same as building the CBT from left-to-right since all incoming arcs are in order. Finally, we return our product matrix on line 10.

5.4.3 Incremental transitive closure

In this section, we describe our algorithm for updating an already calculated transitive closure given the addition of arcs to the underlying graph. While Algorithm 5.5 in Section 5.4.2 already outputs a compressed transitive closure, we need a *differential* compression. This can be done in a simple preprocessing step which merges equivalent rows. We now describe the algorithm.

Algorithm 5.6: Incremental transitive closure

Input: The compressed transitive closure C of a graph $G = (V, E)$, and differential containing the arcs to add G'
Output: The updated compressed transitive closure as a differential, C'

```
1  begin
2      Initialize a new differential graph C' with size n = |V|;
3      foreach new arc (u, v) in G' do
4          ccNodeU ← C.GetCCNode(u);
5          ccNodeV ← C.GetCCNode(v);
6          if ccNodeU != ccNodeV then
7              ccMin ← Min(ccNodeU, ccNodeV);
8              ccMax ← Max(ccNodeU, ccNodeV);
9              newMaxRef ← C'.GetCCNode(ccMax);
10             if ccMin == newMaxRef then
11                 continue;
12             ccRowMax ← C.GetCCRow(ccMax);
13             C'[ccMin].Stream(ccRowMax);
14             C'[ccMax].SetCCNode(ccMin);
15     return C';
```

In Algorithm 5.6, we take as input a differentially compressed transitive closure C of a graph $G = (V, E)$ and a graph differential G' representing the arcs to add, and we output a differential C' which represents the changes to C after G' was added. Thus, we would have to add $C + C'$ to see the complete resulting transitive closure. Note that since C is a static graph, we can only order node-by-node, as opposed to Algorithms 5.4 and 5.7. Additionally, since C is a transitive

closure, we notice that nodes within the same connected components will have the exact same rows in the adjacency matrix of C. Therefore, we encode rows within the same connected component differentially. That is, if $(u, v) \in C$, then we encode u using our compressed binary trees, but only point v to u so that we do not repeat the exact same data multiple times.

We begin in line 2, where we initialize our differential C'. We then loop through each new arc (u, v) in G' (line 3). Since rows u and v may be differentials, we determine which nodes are the representatives for their respective connected components, ccNodeU and ccNodeV (lines 4 and 5). If they are the same representative node for the same connected component, then we know this arc does not change the transitive closure and we can ignore it (line 6). We want to only have y point to x if $x < y$, so we determine whether ccNodeU $>$ ccNodeV or ccNodeU $<$ ccNodeV (lines 7 and 8). We also check if these two connected components have already been merged together by checking if ccMax has already been pointed to ccMin in C' (lines 9-11). If not, we get ccMax's row (line 12), stream it into C'[ccMin] (line 13), and point C'[ccMax] to ccMin (line 14) (signifying that the two connected components have been merged). Finally, we return C' in line 15.

We can see that this algorithm runs in time $O(s_{CC} n \log_2(n))$ per arc, where s_{CC} is the size of the largest strongly connected component. This is because we must find the representative rows of the SCCs for the arc's source and destination (which costs $O(n)$), and we must add a differential equal to s_{CC} which costs $\log_2(n)$ per node in the SCC if the arc connects them.

5.4.4 Time-evolving transitive closure

In this section, we describe an algorithm which processes a time-evolving graph and calculates its time-evolving transitive closure given some time interval.

Algorithm 5.7: Time-evolving transitive closure

Input: A compressed time-evolving graph G_τ, and a time range $[t_{begin}, t_{end}]$

Output: The compressed time-evolving transitive closure $C_{[t_{begin},t_{end}]}$.

```
1  begin
2      Initialize a new time-evolving graph C_[t_begin,t_end] and a static graph of
        the current snapshot G;
3      for t = 0 to t_end do
4          G'_t ← G_τ.ReadNextFrame();
5          if t ≥ t_begin then
6              if t == t_begin then
7                  C ← TransitiveClosure(G);
8                  C_t ← C;
9                  continue;
10             C_t ← IncrementTransitiveClosure(C, G'_t);
11             C.Stream(C_t);
12         else
13             G.Stream(G'_t);
14     return C_[t_begin,t_end];
```

In Algorithm 5.7 we are given as input the complete time-evolving graph G_τ and a time range $[t_{begin}, t_{end}]$, and we output the differentially compressed time-evolving transitive closure $C_{[t_{begin},t_{end}]}$. For this algorithm, G_τ is ordered snapshot-by-snapshot. In other words, it encodes nodes 0 through $n-1$ at t_0, then nodes 0 through $n-1$ at t_1, and so on. This is opposed to the ordering needed in Algorithm 5.4, where we order node-by-node.

We begin in line 2 by initializing a compressed time-evolving graph $C_{[t_{begin},t_{end}]}$ and a static graph representing the current snapshot of G. Similar to Algorithm 5.4, we must begin reading at time $t = 0$ in order to apply our differentials to

compute what the graph is at time $t = t_{begin}$ (line 3). We do this by grabbing the current differential (line 4) and streaming it into the current snapshot (line 13). Once we reach t_{begin} (line 6), we no longer need to maintain the current snapshot. We compute the transitive closure of G at t_{begin} (line 7) and set it as the base frame for the differential frames to be followed (line 8). Once we have our starting transitive closure C, we now only need to use the current differential G'_t to calculate the transitive closure differential for the entire frame (line 10). Additionally, we must apply the transitive closure differential to C in order continue incrementing with the latest transitive closure (line 11). Finally we return $C_{[t_{begin},t_{end}]}$ on line 14.

5.5 Experiments and results

Our experiments involve running the algorithms described in Section 5.4 on the various real-world and synthetic graphs given in Table 5.1. For each graph we provide the number of vertices $|V|$, the number of arcs $|E|$, the lifetime τ, the number of contacts, the number of connected components, and the largest diameter. Our experiments include compressed graph sizes, algorithm run-time, and other metrics. We now describe our datasets.

The Comm.Net* graph is synthetic and represents short communications between random vertices. This results in many disconnected components, and thus a short diameter. Powerlaw* is also synthetic and is a power-law degree graph, where few vertices have many more connections than other vertices, but with a short lifetime. Both graphs have a constant number of active arcs at any given time frame.

The Flickr-Days dataset is an incremental time-evolving graph with time gran-

	Type	|V|	|E|	Lifetime	Contacts	Size(CBT′)	Time(CBT′)
I-Comm.Net*	Interval	10000	15940743	10001	19061571	60.1MB	2.7m
I-Powerlaw*	Interval	1000000	31979927	1001	32280816	123.2MB	5.2m
I-Wiki-Links	Interval	22608064	564224135	414347809	731468598	3.2GB	1.8h
I-Yahoo-Netflow	Interval	103661224	321011861	114193	955033901	4.9GB	2.1h
G-Flickr-Days	Incremental	2585570	33140018	135	33140018	77.9MB	5.1m

Table 5.1: The dataset stats (including the compressed size and the time to compress)

Confidence Level = 95%	$t = 0$ (s)	$[t = 1, t = \tau]$ (s)	Earliest Arrival$_{x=10}$ (s)	Earliest Arrival$_{x=100}$ (s)
I-Comm.Net*	157.289	3.174	0.015 ± 0.006	0.019 ± 0.008
I-Powerlaw*	299.515	3.591	1.503 ± 0.047	2.262 ± 0.108
I-Wiki-Links	472.941	5673.228	133.914 ± 3.611	192.330 ± 7.119
I-Yahoo-Netflow	116.307	7102.003	88.275 ± 2.918	95.043 ± 3.881
G-Flickr-Days	26.045	42.745	4.455 ± 0.240	4.861 ± 1.320

Table 5.2: Algorithm run-times

Confidence Level = 95%	TC (s)	TC_{INC} (s)	$TC_{TE,x=10}$ (s)	$TC_{TE,x=100}$ (s)
I-Comm.Net*	20.581 ± 1.278	0.075 ± 0.053	29.445 ± 0.483	34.445 ± 0.494
I-Powerlaw*	291.344 ± 2.215	1.509 ± 0.022	368.117 ± 10.498	475.989 ± 12.476
I-Wiki-Links	9359.146 ± 162.648	0.084 ± 0.073	12247.086 ± 375.751	13413.209 ± 667.124
I-Yahoo-Netflow	4838.398 ± 243.843	1.746 ± 0.012	6828.252 ± 997.631	7744.206 ± 1631.115
G-Flickr-Days	33.684 ± 0.205	0.022 ± 0.008	42.881 ± 8.246	58.874 ± 11.275

Table 5.3: Algorithm run-times (continued)

ularity set to days from 11-02-2006 to 05-18-2007. It represents when users became friends in the Flickr Social Network and can be found at http://socialnetworks.mpi-sws.org/data-www2009.html.

The Wiki-Links (http://dumps.wikimedia.org/enwiki/) interval dataset shows the history of links among articles in the English version of Wikipedia. It has time granularity by second from 03-04-2014 to 01-01-2015.

Both Flickr-Days and Wiki-Links are one connected component throughout time, and thus had to be disconnected in order to run our transitive closure algorithms.

Finally, we have the Yahoo-Netflow interval graph which contains communication records between end users in the large Internet and Yahoo servers. It has time granularity by seconds from 04-29-2008 to 01-01-2015 and can be found at http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

All datasets are anonymized and required preprocessing. For example, the Yahoo-Netflow dataset contains timestamp, source IP address, destination IP address, source port, destination port, protocol, number of packets, and number of bytes transferred from the source to the destination. The IP addresses were already anonymized with a random permutation algorithm. We also only needed timestamp, source, and destination information, which were converted to frames and vertices, respectively.

We run all of our algorithms on a machine with an Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (4 cores) with 64GB of RAM.

5.5.1 Differential processing

In columns $t = 0$ and $[t = 1, t = \tau]$ of Table 5.2, we have times related to our differential processing.

Before examining our earliest arrival execution times, we must mention that even though the desired time frame for the algorithm is $[t_{begin}, t_{end}]$, we must still process the time-evolving graph from time $t = 0$ to t_{begin}. This is because our compression is differential, and we must read and process the previous frames to know what the graph's active arcs are at t_{begin}.

We also must note that some graphs may already be large at $t = 0$ (e.g., Powerlaw and Flickr-Days), and subsequent frames are smaller changes to the graph.

These two facts are why we have included the time to process the graph at $t = 0$ and from $t = 1$ to $t = \tau$.

These processing times are necessary for all our algorithms and thus are kept separate from the algorithms' actual running times. In other words, the algorithm run-times assume the graph has already been read up to t_{begin}.

5.5.2 Earliest arrival

In columns Earliest Arrival$_{x=10}$ and Earliest Arrival$_{x=100}$ of Table 5.2, we provide times to solve the earliest arrival paths problem directly on our compressed structure.

The running times for our earliest arrival algorithm are the average of 100 iterations with randomly chosen source vertices and time intervals of size 10 and 100. Again, these times assume the graph has already been processed up to time t_{begin}.

From Algorithm 5.4, we know that we perform a variant of BFS where we always have to read a given node's time frames to t_{end}. Thus, the main cause of variance is from the size of the strongly connected component of which the source vertex is a member. The larger the SCC, the bigger the BFS tree and the more nodes we have to process.

5.5.3 Transitive closure

In Table 5.3, we provide various times associated with transitive closure, incremental transitive closure, and time-evolving transitive closure.

In our experiments for transitive closure, the running times assume that the left side (row-by-row) and the right side (column-by-column) of the multiplication are already processed and allocated. Thus, the times only reflect the time to repeatedly multiply and add the two matrices.

Before discussing our transitive closure experiments, notice that although transitive closure is repeated up to $\log_2(n)$ times, we can also short-circuit this logic if the current iteration's result is the same as the previous iteration.

In the results we can indeed see that the denser versions of the graphs calculate more quickly than their smaller time frames. The exceptions to this are the synthetic graphs, which contain a constant number of active arcs at any given time frame. These times are essentially the same, but slightly varied most likely because arcs changed locations in the compressed binary trees. Yahoo-Netflow also took longer in its larger time frame. This is because the graph is disconnected enough that the extra arcs did not help the short-circuiting and only added to navigation time.

5.5.4 Incremental transitive closure

Using the transitive closures calculated earlier, we average the time required to stream in 1000 random arcs which connect two strongly connected components (possibly of size 1). Since the number of SCCs does not change between the smallest and largest versions of the graphs (except Yahoo-Netflow), we only have to consider one transitive closure per graph. Graphs which are already one SCC won't change given any arc, but must still determine that the source and vertex are in the same SCC.

We see this fact reflected in the results, where the average time to stream an arc into Powerlaw*, Wiki-Links, and Flickr-Days is only the time needed to check if the source and destination are in the same SCC.

In Comm.Net* and Yahoo-Netflow, the algorithm still has to determine the SCCs for the source and vertex, but must also merge the components by adding differentials for the transitive closure.

5.5.5 Time-evolving transitive closure

Now, with the incremental transitive closure algorithm, we can process a compressed time-evolving graph and give the transitive closure of each time frame as another compressed time-evolving graph. We do this by using the transitive closure and developing a time-evolving transitive closure for the next 10 and 100 frames. The running times begin after the initial transitive closure has been calculated.

In this experiment, we must not only merge connected components and create differentials, but we must also apply these differentials to our current running copy of the transitive closure. The results show that this operation is actually quite

cheap, as our differentials are efficient and our streaming operation is fast with the use of an unrolled linked list for our underlying bit-string data structure.

5.6 Related work

Time-evolving graphs have been extensively researched with many structures [15] [16] [20] [22] [23] [47] [78] and problems/algorithms [24] [63] [65] [133] [135] already defined.

In 2016, Caro et al. [23] developed and tested several time-evolving data structures including the ck^d-tree. They define a contact as a quadruplet (u, v, t_i, t_j) and then compress the 4D binary matrix corresponding to the time-evolving graph defined by a set of these contacts. They do this by representing the 4D matrix as a k^dtree and then distinguishing white nodes as those without any contacts, black nodes as ones that only contain contacts, and gray nodes as those that contain only one contact. This work was preceded by Brisaboa et al.'s k^2-trees [16] in 2014. However, not only did they not reach compression rates and query times as good as our technique, but their compression is not suitable for most node-centric algorithms.

Time-evolving path problems have also been thoroughly defined [24] [63] [65] [133] [135]. However, none of these works have run their algorithms on compressed graphs.

Transitive closure has also been well studied with many improvements over the years [3] [49] [76] [90] [106] [137]. However, none use compressed structures to calculate the transitive closure and none address our novel problem of a compressed time-evolving transitive closure, although a few [26] [66] do examine the benefits of compressing a pre-computed transitive closure.

5.7 Conclusion

In this chapter, we developed a differential time-evolving graph compression and applied algorithms on it. We found that not only were we the first to solve such problems on compressed time-evolving graphs, but that our structure was particularly efficient at doing so. We solved the earliest arrival paths problem, incremental transitive closure problem, and our novel problem of time-evolving transitive closure.

Chapter 6

Conclusion

Finally, we conclude by summarizing the concepts we covered in this dissertation. Aftter summarizing each chapter, we give some closing remarks in Section 6.6.

6.1 Chapter 1

We began in Chapter 1, by introducing the concept of networks and the problems we face when performing analysis on them. When networks become too massive (billions to trillions of nodes and arcs), even basic data structures such as adjacency lists end up requiring petabytes to zettabytes of memory. If we store these networks in secondary memory (rather than main memory), any analysis will require I/O access (i.e., disk access), thus drastically slowing analysis time. Therefore, if we want to perform analysis on massive networks, we must either provide enough main memory or supply some data structure which sufficiently compresses the data while still allowing fast access times.

In Section 1.1, we described many real-world social, information, technological, and biological networks to give the reader more context for the dissertation.

Each category of network was also supplied with sample networks and references to relevant works.

After our context of real-world networks was established, we continued in Section 1.2 by formally defining how graphs represent networks, including time-evolving networks. Tables 1.1 and 1.2 listed out some real-world network graphs and their properties. We also provided a discussion of basic representations such as the Boolean adjacency matrix, adjacency lists, and arc lists in Section 1.2.2. Common properties and queries of static and time-evolving graphs were enumerated in Sections 1.2.3 and 1.2.4.

After defining all of our graph concepts, we familiarized ourselves with some basic compression concepts in Section 1.3. This included a formal definition of lossless compression, Kolmogorov complexity, and information-theoretic lower bounds (Section 1.3.1), as well as formal definitions of queryable compression (Section 1.3.2) and incremental compression (Section 1.3.3).

We then moved on to graph-specific compression in Section 1.4. We discussed some common exploitable structural properties (Section 1.4.1), node reordering techniques (Section 1.4.2), and the popular CSR technique (Section 1.4.3).

Prior work was discussed in Section 1.5. Here, we not only defined what it means for a compression to be implicit, succinct, and compact, but we also provided Table 1.4 which summarized many succinct arbitrary graph representations.

Finally, we outlined the rest of the paper in Section 1.6.

6.2 Chapter 2

In Chapter 2, we split a quadtree compression of a static graph's 2D Boolean adjacency matrix into its adjacency rows and compress them with compressed binary trees. Adding indexes to this technique greatly sped-up access times while only barely increasing space. We provided our motivation and exact contributions in Section 2.1.

In Section 2.2, we discussed related work and justified our benchmark selection. We also referred to our previous work using quadtrees and a similar work with k^2-trees. Preliminaries were covered in Section 2.3, where we discussed network properties, compression concepts, and queries.

We described our novel technique in Section 2.4, where we represented each row of the Boolean adjacency matrix with a compressed binary tree. We achieved the information-theoretic lower bound (Section 2.4.4), and we can optionally add indexes to our structure to gain a query complexity proportional to the maximum degree of the graph.

Normally, compression starts with the uncompressed graph in a raw text file as an arc list. A normal technique would then load the arc list into some intermediate structure (such as adjacency lists) for faster compression. Finally, the technique outputs the compressed graph as a bit-string. We not only provided a way to compress the graph without any intermediate structure (Algorithms 2.3 and 2.4), but we can also compress directly from a text file that has been compressed using some general technique such as gzip.

In Section 2.5, we presented our datasets and metrics. We saw from the results that our compression outperforms our benchmark in every metric. Our results also showed that our streaming times were nearly identical to the query times

while using the custom in-memory structure. Finally, we concluded the chapter in Section 2.6.

The results in this chapter were published and presented at the 2017 IEEE International Conference on Big Data (Big Data 2017) in Boston, MA.

6.3 Chapter 3

In Chapter 3, we applied differential compression to the technique introduced in Chapter 2. We found that this differential static graph compression allowed us to reuse previously computed products when performing matrix-vector multiplication. Thus, algorithms such as PageRank are now no longer bound by the size of the graph's 2D Boolean adjacency matrix, but by the size of the compressed graph. We discussed this motivation and outlined the rest of the chapter in Section 3.1. We also discussed related work and chose our benchmark in Section 3.2.

In Section 3.3, we described how to apply a differential compression between compressed binary trees. Once we obtained a differential representation, we showed how to perform matrix-vector multiplication and we proved that its run-time is proportional to the size of the compressed graph (Section 3.4). We also extended this technique to matrix-matrix multiplication and again proved that the run-time is proportional to the size of the compressed graph (Section 3.5).

We presented our datasets and metrics in Section 3.6. We saw that not only are our base query speeds faster than the benchmark, but the speed-up gained from reusing products is greater as well. Finally, we concluded the chapter in Section 3.7. Here, we proposed future work by using matrix-matrix multiplication for transitive closures and by reordering the rows of the matrix to improve

differential compression.

These results were published at the 2019 IEEE International Conference on Information Reuse and Integration (IRI 2019) in Los Angeles, CA.

6.4 Chapter 4

In Section 4.1, we adapted our differential compression technique from Chapter 3 to compress time-evolving graphs, rather than static graphs. We used this differential compression to only encode the differences between a node's time frames. Time-evolving graphs are interesting because they store a graph's complete state at each time frame, and thus, we can run analyses that study patterns throughout time. We discussed this motivation and outlined the rest of the chapter in Section 4.1.

We more formally defined time-evolving graphs and the operations performed on them in Section 4.2. We showed that time-evolving graphs could be incremental, interval, or point, and that the basic operations performed on them were time-evolving versions of the arc existence query, neighbor query, and streaming query. We discussed related work in Section 4.3, including ck^d-trees, suffix arrays, and various log and copy+log techniques.

Our time-evolving technique was defined in Section 4.4, where we apply differential compression between each node's time frames. We also provided an extension that allows us to combine multiple differential compressed binary trees into one T-CBT (Section 4.4.3). We did all of this using only minimal memory overhead again by compressing directly from a gzipped time-evolving arc (contact) list (Section 4.4.4). We also proved that our compression reaches the information-theoretic lower bound, with respect to the 3D Boolean adjacency ma-

trix and the number of contacts (Section 4.4.5). A suite of supported operations was also provided including time-evolving arc existence, neighbors, streaming arcs (throughout any time-frame), and streaming time-frames themselves (Section 4.4.7).

We then ran experiments on our new technique in Section 4.5. The metrics included the compressed graphs' sizes (among many benchmarks), compression time, and supported operation run-times. We saw that our technique compressed to sizes better than or equal to any of our benchmarks. Our compression times also ended up being proportional to the number of contacts in the time-evolving graphs. The supported operations were not run with our benchmarks, but arc existence and neighbor queries ran at nearly identical times, with the streaming operation only slightly slower (while using our custom in-memory structure). We provided discussion of all these topics in Section 4.5.

Finally, we concluded the chapter in Section 4.6.

The results in this chapter were published and presented at the 2018 IEEE International Conference on Big Data (Big Data 2018) in Boston, MA.

6.5 Chapter 5

In Chapter 5, we adapted our time-evolving graph compression technique for solving the earliest arrival paths problem and for our own novel definition of time-evolving transitive closure. We found that not only were we the first to solve such problems on compressed time-evolving graphs, but that our structure was particularly efficient at doing so.

We briefly discussed motivation and outlined the chapter in Section 5.1. After that, we formally defined time-evolving graphs (Section 5.2.1), the earliest-

arrival paths problem (Section 5.2.2), and time-evolving transitive closure (Section 5.2.2). Again, note that our definition of our time-evolving transitive closure is novel, to the best of our knowledge.

In Section 5.3, we began by describing our compressed binary trees, differential technique, and ways that we can order them to achieve different representations of a time-evolving graph. That is, we can order them node-by-node or snapshot-by-snapshot. We then provided Algorithm 5.3 which is the basis of Boolean vector-vector multiplication using our technique.

In Section 5.4, we described how to apply our technique to solve many problems. We used our snapshot-by-snapshot ordering to solve the earliest-arrival problem in Section 5.4.1. Using this ordering allowed us to use a time-evolving version of BFS to find the earliest-arrival paths and times in one pass. Next, we provided Algorithm 5.5, which used repeated multiplication to find the transitive closure of the compressed graph. Continuing with transitive closure, in Section 5.4.3 we provided Algorithm 5.6, which can increment a compressed transitive closure graph matrix given a set of arcs to add (stored as a compressed differential). Finally, we provided Algorithm 5.7 in Section 5.4.4, which used node-by-node ordering and calculated (while keeping everything compressed) the time-evolving transitive closure given a time-evolving graph and a desired time interval.

After developing all of our algorithms, we ran experiments on them in Section 5.5. Our metrics included the time to process different intervals of the graph and run-times for each algorithm. We discovered amazing run-times due to our clever ordering techniques and the fact that our algorithms run in time proportional to the size of the compressed graph. Thus, the smaller we can compress the graph, the better our run-times become. We discussed these results in detail.

Related work was discussed in Section 5.6. We found that we had no appro-

priate benchmarks to compare against, as our novel problem definitions had not yet been adapted by any other technique. Finally, we concluded in Section 5.7.

These finding were published in the 2019 IEEE International Conference on Big Data (Big Data 2019) in Los Angeles, CA.

6.6 Closing remarks

In this dissertation, we developed several succinct representations for both static graphs and time-evolving graphs. This included a suite of supported operations for each technique. Each chapter explored relevant work from which we chose our benchmarks. We provided a detailed analysis showing we achieved the information-theoretic lower bound, and we ran experiments that empirically that showed that we outperformed our benchmarks (where applicable). We believe that these improvements are great contributions to the fields of compression and massive network analysis.

Bibliography

[1] Lada A Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3):211–230, 2003.

[2] Micah Adler and Michael Mitzenmacher. Towards Compressing Web Graphs. In *Proceedings of the Data Compression Conference*, number 9 in DCC '01, pages 203–212, Washington, DC, USA, 2001.

[3] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[4] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the world-wide web. *Nature*, 401(6749):130, 1999.

[5] Luis A Nunes Amaral, Antonio Scala, Marc Barthelemy, and H Eugene Stanley. Classes of small-world networks. *Proceedings of the National Academy of Sciences*, 97(21):11149–11152, 2000.

[6] Baruch Awerbuch and Tripurari Singh. New connectivity and msf algorithms for ultracomputer and pram. In *ICPP*, volume 83, pages 175–179, 1983.

[7] Albert-László Barabási, Réka Albert, and Hawoong Jeong. Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics and Its Applications*, 281(1-4):69–77, 2000.

[8] Albert-László Barabási, Natali Gulbahce, and Joseph Loscalzo. Network medicine: a network-based approach to human disease. *Nature Reviews Genetics*, 12(1):56, 2011.

[9] Brian Beckman. Why linq matters: cloud composability guaranteed. *Commun. ACM*, 55(4):38–44, 2012.

[10] G. D. Bernardo, N. R. Brisaboa, D. Caro, and M. A. Rodrguez. Compact data structures for temporal graphs. In *2013 Data Compression Conference*, number 1, pages 477–477, March 2013.

[11] Maciej Besta and Torsten Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *CoRR*, abs/1806.01799, 2018.

[12] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web*, number 7 in WWW '04, pages 595–602, New York, NY, USA, 2004.

[13] James E Bostick, John M Ganci Jr, Martin G Keen, and Sarbajit K Rakshit. Performing contextual analysis of incoming telephone calls and suggesting forwarding parties, September 18 2018. US Patent 10,079,939.

[14] Nieves R. Brisaboa, Diego Caro, Antonio Fariña, and M. Andrea Rodríguez. A compressed suffix-array strategy for temporal-graph indexing. In *SPIRE*, 2014.

[15] Nieves R Brisaboa, Diego Caro, Antonio Fariña, and M Andrea Rodriguez. Using compressed suffix-arrays for a compact representation of temporal-graphs. *Information Sciences*, 465(24):459–483, 2018.

[16] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39(22):152–174, 2014.

[17] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, 2000.

[18] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. In *SIAM J. Comput.*, Volume 28, Issue 5. Society for Industrial and Applied Mathematics, 1999.

[19] Andre Broido et al. Internet topology: Connectivity of ip graphs. In *Scalability and Traffic Control in IP Networks*, volume 4526, pages 172–187, 2001.

[20] Binh-Minh Bui-Xuan, Afonso Ferreira, and Aubin Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. Technical Report RR-4589, INRIA, October 2002.

[21] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, number 11, pages 233–244, 2009.

[22] Diego Caro, M. Andrea Rodríguez, and Nieves R. Brisaboa. Data structures for temporal graphs based on compact sequence representations. *Inf. Syst.*, 51(C):1–26, July 2015.

[23] Diego Caro, M. Andrea Rodriguez, Nieves R. Brisaboa, and Antonio Farina. Compressed kd-tree for temporal graphs. *Knowl. Inf. Syst.*, 49(2):553–595, November 2016.

[24] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(21):387–408, 2012.

[25] Qian Chen, Hyunseok Chang, Ramesh Govindan, and Sugih Jamin. The origin of power laws in internet topologies revisited. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 608–617, 2002.

[26] Yangjun Chen and Yibin Chen. Decomposing dags into spanning trees: A new way to compress transitive closures. In *2011 IEEE 27th International Conference on Data Engineering*, number 11, pages 1007–1018, 2011.

[27] 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*, number 10 in KDD '09, pages 219–228, 2009.

[28] Fan Chung. The heat kernel as the pagerank of a graph. *Proceedings of the National Academy of Sciences*, 104(50):19735–19740, 2007.

[29] J. Clement. Number of monthly active facebook users worldwide as of 3rd quarter 2019 (in millions), 2019.

[30] Russell C Coile. Lotka's frequency distribution of scientific productivity. *Journal of the American Society for Information Science*, 28(6):366–370, 1977.

[31] Germán Creamer, Ryan Rowe, Shlomo Hershkop, and Salvatore J Stolfo. Segmentation and automated social hierarchy detection through email network analysis. In *International Workshop on Social Network Mining and Analysis*, number 18, pages 40–58. Springer, 2007.

[32] Toby Davies and Shane D Johnson. Examining the relationship between road structure and burglary risk via quantitative network analysis. *Journal of Quantitative Criminology*, 31(3):481–507, 2015.

[33] Don Rutledge Day and Rabindranath Dutta. Using video image analysis to automatically transmit gestures over a network in a chat or instant messaging session, May 2 2006. US Patent 7,039,676.

[34] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

[35] Sergey N Dorogovtsev, José Fernando F Mendes, and Alexander N Samukhin. Giant strongly connected component of directed networks. *Physical Review E*, 64(2):025101, 2001.

[36] Wen-Bo Du, Xing-Lian Zhou, Oriol Lordan, Zhen Wang, Chen Zhao, and Yan-Bo Zhu. Analysis of the chinese airline network as multi-layer networks. *Transportation Research Part E: Logistics and Transportation Review*, 89(8):108–116, 2016.

[37] Jennifer A Dunne, Richard J Williams, and Neo D Martinez. Network structure and robustness of marine food webs. *Marine Ecology Progress Series*, 273(11):291–302, 2004.

[38] Jennifer A Dunne, Richard J Williams, Neo D Martinez, Rachel A Wood, and Douglas H Erwin. Compilation and network analyses of cambrian food webs. *PLoS Biology*, 6(4):e102, 2008.

[39] SC Eisenstat, MC Gursky, MH Schultz, and AH Sherman. Yale sparse matrix package. i. the symmetric codes. Technical report, Yale Univ New Haven CT Dept of Computer Science, 1977.

[40] SC Eisenstat, MC Gursky, MH Schultz, and AH Sherman. Yale sparse matrix package. ii. the nonsymmetric codes. Technical report, Yale Univ New Haven CT Dept of Computer Science, 1977.

[41] P Elias. Universal codeword sets and respresentations of the integers. In *IEEE Transactions on Information Theory*, pages 194–203, 1975.

[42] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262, 1999.

[43] Mehdi Farsi, Massimo Filippini, and Urs Trinkner. Economies of scale, density and scope in swiss posts mail delivery. *Liberalization of the Postal and Delivery Sector*, (10):91–101, 2006.

[44] Arash Farzan and J Ian Munro. Succinct representations of arbitrary graphs. In *European Symposium on Algorithms*, number 11, pages 393–404, 2008.

[45] Gerald Faulhaber. Network effects and merger analysis: instant messaging and the aol–time warner case. *Telecommunications Policy*, 26(5-6):311–333, 2002.

[46] David A Fell and Andreas Wagner. The small world of metabolism. *Nature Biotechnology*, 18(11):1121, 2000.

[47] Afonso Ferreira and Laurent Viennot. A Note on Models, Algorithms, and Data Structures for Dynamic Communication Networks. Research Report RR-4403, INRIA, 2002.

[48] Johannes Fischer and Daniel Peters. Glouds: Representing tree-like graphs. *Journal of Discrete Algorithms*, 36(10):39–49, 2016.

[49] Michael J Fischer and Albert R Meyer. Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, number 3, pages 129–131, 1971.

[50] Alexandre Francisco, Travis Gagie, Susana Ladra, and Gonzalo Navarro. Exploiting computation-friendly graph compression methods for adjacency-matrix multiplication. In *2018 Data Compression Conference*, number 7, pages 307–314, 2018.

[51] S. . Garca, N. R. Brisaboa, G. d. Bernardo, and G. Navarro. Interleaved k2-tree: Indexing and navigating ternary relations. In *2014 Data Compression Conference*, number 9, pages 342–351, March 2014.

[52] Saptarshi Ghosh, Avishek Banerjee, Naveen Sharma, Sanket Agarwal, Niloy Ganguly, Saurav Bhattacharya, and Animesh Mukherjee. Statistical analysis of the indian railway network: a complex network approach. *Acta Physica Polonica B Proceedings Supplement*, 4(15):123–138, 2011.

[53] John Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, number 9, pages 16–25, 1994.

[54] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[55] Jiancheng Guan and Na Liu. Exploitative and exploratory innovations in knowledge network and collaboration network: A patent analysis in the technological field of nano-energy. *Research Policy*, 45(1):97–112, 2016.

[56] JianCheng Guan, KaiRui Zuo, KaiHua Chen, and Richard CM Yam. Does country-level r&d efficiency benefit from the collaboration network structure? *Research Policy*, 45(4):770–784, 2016.

[57] John Guare. *Six degrees of separation: A play*. Vintage, 1990.

[58] Roger Guimera, Brian Uzzi, Jarrett Spiro, and Luis A Nunes Amaral. Team assembly mechanisms determine collaboration network structure and team performance. *Science*, 308(5722):697–702, 2005.

[59] Alexander Halavais. *Search Engine Society*. John Wiley & Sons, 2017.

[60] Edward JS Hearnshaw and Mark MJ Wilson. A complex network approach to supply chain network theory. *International Journal of Operations & Production Management*, 33(4):442–469, 2013.

[61] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, number 9, pages 21–30, 2015.

[62] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[63] Petter Holme and Jari Saramäki. Temporal networks. *Physics Reports*, 519(28):97–125, 2012.

[64] Ralf Hölzer, Bradley Malin, and Latanya Sweeney. Email alias detection using social network analysis. In *Proceedings of the 3rd ACM International Workshop on Link Discovery*, number 5, pages 52–57, 2005.

[65] Silu Huang, James Cheng, and Huanhuan Wu. Temporal graph traversals: Definitions, algorithms, and applications. *CoRR*, 2014.

[66] HV Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems (TODS)*, 15(4):558–598, 1990.

[67] Erik Jenelius and Lars-Göran Mattsson. Road network vulnerability analysis: Conceptualization, implementation and application. *Computers, Environment and Urban Systems*, 49(11):136–147, 2015.

[68] Alan Jennings. Matrix computation for engineers and scientists. *London and New York, Wiley-Interscience, 1977. 340,*, 1977.

[69] Hawoong Jeong, Bálint Tombor, Réka Albert, Zoltan N Oltvai, and A-L Barabási. The large-scale organization of metabolic networks. *Nature*, 407(6804):651, 2000.

[70] VK Kalapala, V Sanwalani, and C Moore. The structure of the united states road network. *Preprint, University of New Mexico*, 2003.

[71] U Kang, Charalampos E Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. Radius plots for mining tera-byte scale graphs: Algorithms, patterns, and observations. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, number 10, pages 548–558, 2010.

[72] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the Ninth IEEE International Conference on Data Mining*, number 10 in ICDM '09, pages 229–238, 2009.

[73] Chinmay Karande, Kumar Chellapilla, and Reid Andersen. Speeding Up Algorithms on Compressed Web Graphs. *Internet Mathematics*, 6(3):373–398, February 2009.

[74] Henry Kautz, Bart Selman, and Mehul Shah. Referral web: combining social networks and collaborative filtering. *Communications of the ACM*, 40(3):63–65, 1997.

[75] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, (11):997–1008, 2013.

[76] Valerie King and Garry Sagert. A fully dynamic algorithm for maintaining the transitive closure. *Journal of Computer and System Sciences*, 65(1):150–167, 2002.

[77] Jon M Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S Tomkins. The web as a graph: measurements, models, and methods. In *International Computing and Combinatorics Conference*, number 17, pages 1–17. Springer, 1999.

[78] Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen, Jr., Sean R. Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The g* graph database: Efficiently managing large distributed dynamic graphs. *Distrib. Parallel Databases*, 33(4):479–514, December 2015.

[79] Kevin D Lafferty, Andrew P Dobson, and Armand M Kuris. Parasites dominate food web links. *Proceedings of the National Academy of Sciences*, 103(30):11211–11216, 2006.

[80] Kasper Green Larsen and Ryan Williams. Faster online matrix-vector multiplication. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, number 7, pages 2182–2189, 2017.

[81] Vito Latora and Massimo Marchiori. Is the boston subway a small-world network? *Physica A: Statistical Mechanics and its Applications*, 314(4):109–113, 2002.

[82] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.

[83] Panagiotis Liakos, Katia Papakonstantinopoulou, and Michael Sioutis. Pushing the envelope in graph compression. *CIKM '14*, (9):1549–1558, November 2014.

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

[85] Yun Liu, Veena B Mendiratta, and Kishor S Trivedi. Survivability analysis of telephone access network. In *15th IEEE International Symposium on Software Reliability Engineering*, number 10, pages 367–377, 2004.

[86] Oriol Lordan, Jose M Sallan, and Pep Simo. Study of the topology and robustness of airline route networks from the complex network approach: a survey and research agenda. *Journal of Transport Geography*, 37(8):112–120, 2014.

[87] Linyuan Lü, Duanbing Chen, Xiao-Long Ren, Qian-Ming Zhang, Yi-Cheng Zhang, and Tao Zhou. Vital nodes identification in complex networks. *Physics Reports*, 650(63):1–63, 2016.

[88] Sebastian Maneth and Fabian Peternek. A Survey on Methods and Systems for Graph Compression. 6(3), 2015.

[89] Peter V Marsden. Network data and measurement. *Annual review of sociology*, 16(1):435–463, 1990.

[90] Daniel P Martin. Dynamic shortest path and transitive closure algorithms: A survey. *CoRR*, 2017.

[91] Tobias Martin, Gerald Lumma, Ulrike Weber, Dieter Wuest, and Soenke Gruetzmacher. Telephony communications system for detecting abuse in a public telephone network, May 14 2019. US Patent 10,291,772.

[92] 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*, number 22 in KDD '10, pages 303–325, New York, NY, USA, 2010. Springiner-Verlag New York.

[93] Stanley Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[94] Erfanmanesh Mohammadamin, Rohani Vala Ali, and Abdullah Abrizah. Co-authorship network of scientometrics research collaboration. *Malaysian Journal of Library & Information Science*, 17(3):73–93, 2017.

[95] James Moody. The structure of a social science collaboration network: Disciplinary cohesion from 1963 to 1999. *American Sociological Review*, 69(2):213–238, 2004.

[96] Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[97] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. Information network or social network?: the structure of the twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web*, number 5, pages 493–498, 2014.

[98] Moni Naor. Succinct representation of general unlabeled graphs. *Discrete Applied Mathematics*, 28(3):303–307, 1990.

[99] Michael Nelson, Sridhar Radhakrishnan, Amlan Chatterjee, and Chandra Sekharan. On compressing massive streaming graphs with Quadtrees. In *2015 IEEE International Conference on Big Data (Big Data)*, 2015.

[100] Michael Nelson, Sridhar Radhakrishnan, Amlan Chatterjee, and Chandra Sekharan. Queryable Compression on Streaming Social Networks. In *2017 IEEE International Conference on Big Data (Big Data)*, 2017.

[101] Michael Nelson, Sridhar Radhakrishnan, and Chandra Sekharan. Queryable compression on time-evolving social networks with streaming. In *2018 IEEE International Conference on Big Data*, number 5 in IEEE BigData '18, pages 146–151. IEEE, IEEE Computer Society, 2018.

[102] Chien-Chun Ni, Yu-Yao Lin, Jie Gao, Xianfeng David Gu, and Emil Saucan. Ricci curvature of the internet topology. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, number 8, pages 2758–2766. IEEE, 2015.

[103] Vincenzo Nicosia, John Kit Tang, Cecilia Mascolo, Mirco Musolesi, Giovanni Russo, and Vito Latora. Graph metrics for temporal networks. *CoRR*, 2013.

[104] Takashi Nishikawa and Adilson E Motter. Comparative analysis of existing models for power-grid synchronization. *New Journal of Physics*, 17(1):015012, 2015.

[105] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Accelerating graph adjacency matrix multiplications with adjacency forest. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, number 8, pages 1073–1081, 2014.

[106] Patrick E O'Neil and Elizabeth J O'Neil. A fast expected time algorithm for boolean matrix multiplication and transitive closure. *Information and Control*, 22(2):132–138, 1973.

[107] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[108] Pradeep Pillai, Andrew Gonzalez, and Michel Loreau. Metacommunity theory explains the emergence of food web complexity. *Proceedings of the National Academy of Sciences*, 108(48):19293–19298, 2011.

[109] János Podani, Zoltán N Oltvai, Hawoong Jeong, Bálint Tombor, A-L Barabási, and E Szathmary. Comparable system-level organization of archaea and eukaryotes. *Nature Genetics*, 29(1):54, 2001.

[110] Derek J De Solla Price. Networks of scientific papers. *Science*, (5):510–515, 1965.

[111] Anabel Quan-Haase, Joseph Cothrel, and Barry Wellman. Instant messaging for collaboration: A case study of a high-tech firm. *Journal of Computer-Mediated Communication*, 10(4):JCMC10413, 2005.

[112] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4):43, 2007.

[113] Vaishali Rampurkar, Polgani Pentayya, Harivittal A Mangalvedekar, and Faruk Kazi. Cascading failure analysis for indian power grid. *IEEE Transactions on Smart Grid*, 7(4):1951–1960, 2016.

[114] 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*, number 9 in DCC '02, pages 122–131, 2002.

[115] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.

[116] John Resig and Ankur Teredesai. A framework for mining instant messaging services. In *Proceedings of the 2004 SIAM DM Conference*, 2004.

[117] Ryan Rowe, German Creamer, Shlomo Hershkop, and Salvatore J Stolfo. Automated social hierarchy detection through email network analysis. (8):109–117, 2007.

[118] Parongama Sen, Subinay Dasgupta, Arnab Chatterjee, PA Sreeram, G Mukherjee, and SS Manna. Small-world properties of the indian railway network. *Physical Review E*, 67(3):036106, 2003.

[119] Olaf Sporns. Network analysis, complexity, and brain function. *Complexity*, 8(1):56–60, 2002.

[120] Olaf Sporns and Richard F Betzel. Modular brain networks. *Annual Review of Psychology*, 67(27):613–640, 2016.

[121] Olaf Sporns, Giulio Tononi, and Gerald M Edelman. Theoretical neuroanatomy: relating anatomical and functional connectivity in graphs and cortical connection matrices. *Cerebral Cortex*, 10(2):127–141, 2000.

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

[123] Jörg Stelling, Steffen Klamt, Katja Bettenbrock, Stefan Schuster, and Ernst Dieter Gilles. Metabolic network structure determines key aspects of functionality and regulation. *Nature*, 420(6912):190, 2002.

[124] Anthony M Townsend. Network cities and the global structure of the internet. *American Behavioral Scientist*, 44(10):1697–1716, 2001.

[125] Jeffrey Travers and Stanley Milgram. An experimental study of the small world problem. In *Social Networks*, number 18, pages 179–197. 1977.

[126] György Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.

[127] Uzi Vishkin. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics*, 9(2):197–207, 1984.

[128] Andreas Wagner and David A Fell. The small world inside large metabolic networks. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 268(1478):1803–1810, 2001.

[129] Duncan J Watts. *Small worlds: the dynamics of networks between order and randomness*, volume 9. Princeton University Press, 2004.

[130] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *Nature*, 393(3):440–442, 1998.

[131] John G White, Eileen Southgate, J Nichol Thomson, and Sydney Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. *Philos Trans R Soc Lond B Biol Sci*, 314(1165):1–340, 1986.

[132] Jeffrey B Winner and Nicholas Galbreath. Online content delivery based on information from social networks, August 6 2019. US Patent 10,373,173.

[133] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, 7(11):721–732, 2014.

[134] Miao Wu, Ting-Jie Lu, Fei-Yang Ling, Jing Sun, and Hui-Ying Du. Research on the architecture of internet of things. In *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, volume 5, pages V5–484, 2010.

[135] B Bui Xuan, Afonso Ferreira, and Aubin Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(18):267–285, 2003.

[136] Bo Yang, Yu Lei, Jiming Liu, and Wenjie Li. Social collaborative filtering by trust. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(8):1633–1647, 2016.

[137] Huacheng Yu. An improved combinatorial algorithm for boolean matrix multiplication. (11):1094–1105, 2015.

[138] Reza Zafarani, Mohammad Ali Abbasi, and Huan Liu. *Social media mining: an introduction*. Cambridge University Press, 2014.