AN EXPERIMENTAL STUDY ON COMPRESSED REPRESENTATIONS OF WEB GRAPHS AND SOCIAL NETWORKS BASED ON DENSE SUBGRAPH EXTRACTION

by

Chen Miao

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
April 2016

Table of Contents

List of Tables

List of Figures

Abstract

The compressed web graph and social networks structures proposed by Hernández and Navarro [22] support more queries than alternative approaches, including in-neighbor, out-neighbour, and mining queries. Their main strategy is to extract dense subgraphs from the given graph, and encode them using succinct data structures such as wavelet trees. Previous experimental studies on wavelet trees, however, test their performance using textual data, which are different from the data generated from web graphs. We thus engineer wavelet tree implementations for compressed web graph and social network structures. In particular, we propose a new wavelet tree encoding, called *combined encoding*, which provides better time/space tradeoffs than previous approaches when used in Hernández and Navarro's framework to represent web graphs.

List of Abbreviations Used

CHWTNP pointerless Canonical Huffman-shaped wavelet trees.

HITS Hyperlink-induced topic search.

HWT pointer-based Huffman-shaped wavelet trees.

MPk multiposition linearization of degree k.

RLE run-length encoding.

RLED run-length encoding + δ.

RLEG run-length encoding + γ.

RLEO run-length encoding + ω.

URL Uniform Resource Locator.

WT pointer-based normal-shaped wavelet trees.

WTNP pointerless normal-shaped wavelet trees.

Acknowledgements

First, I would like to thank my supervisor, Dr. Meng He, for his guidance, support, encouragement and patience throughout my graduate program. Not only has he taught a great deal about research, about how to solve the problems, he has also offered me numerous advices on various aspects of academic life, such as academic writing and presentations.

I also want to thank Dr. Peter Bodorik and Dr. Abdel Aziz Farrag to be my readers. Thanks for their comments and time to help me bring this research to this level.

I would like to thank Dalhousie University to provide such a great environment to do research and allowing me to finish my work.

I am proudly grateful to my supportive and loving family for their whole-hearted support of my graduate studies.

Chapter 1

Introduction

The World Wide Web and social networks have been the focus of many academic and industrial research centres. Both the web and social networks can be viewed as graphs: The web can be modelled as a directed graph in which each vertex represents a web page, and there is a directed edge from vertex u to vertex v if there is a hyperlink in the page represented by u that refers to the page represented by v. Such a graph structure is commonly called a *web graph*. Social networks are commonly viewed as a graph in which vertices represent social entities such as individuals and/or organizations, and edges represent relationships between these entities. Many graph algorithms have thus been developed for various applications to extract information from web graphs and social networks. For instance, many ranking algorithms which orders, by relevance and importance, the web pages retrieved by search engines are essentially algorithms on web graphs, e.g., the PageRank [6] and HITS (hyperlink-induced topic search) [27] algorithms. Algorithms that detect link spamming often relies on computations on web graphs [39], such as extracting strongly connected components, enumerating maximal cliques and computing minimum cut. Social network structures are frequently used for data mining and analytics, and more specifically, to identify interest groups or communities [40], to detect important persons over real social networks [40], and to understand information propagation [30, 31, 10].

These graph algorithms typically require the underlying web graphs or social networks to be encoded as data structures residing entirely in the main memory.

Traditionally, graphs have often been represented by either adjacency lists or adjacency matrices. However, as the sizes of web graphs and social networks have been increasing rapidly, these standard representations often use too much space to fit in main memory, and performance will be sacrificed if secondary storage is used as virtual memory. At present, the indexed web contains at least 4.6 billion pages (`http://www.worldwidewebsize.com/`), and Facebook, a popular social network platform, had 1.01 billion daily active users on average in September 2015 (`http://newsroom.fb.com/company-info/`). For large graphs of this scale, designing a representation that fits in the main memory is a challenging task.

Much work has thus been done to design compact representations of web graphs and social networks [41, 1, 37, 5, 9, 2, 4, 14, 29, 13, 22, 7]. In most previous work, the main strategy of representing a web graph is to compress its adjacency lists by taking advantage of properties such as *locality* (pages tend to have hyperlinks to other pages with the same domain names in their URLs) and *similarity* (similar pages tend to have similar adjacency lists). This includes the early work of Suel and Yuan [41] which achieves compression by using different codes including Huffman for hyperlinks pointing to pages with different domain names and those pointing to pages in the same domain, and the work of Adler and Mitzenmacher [1] which discovers pages that share common hyperlinks and encodes the difference between the corresponding adjacency lists. Subsequent work renumbers web graph vertices based on the URLs of corresponding pages to exploit locality and the similarity between nearby adjacency lists [37, 5], designs vertex ordering based on a breadth-first traversal of the graph [2], or uses virtual nodes to represent bicliques or frequent pairs of symbols in adjacency lists [9, 14]. Similar approaches have also been used to represent social networks, and ideas such as shingling order and link reciprocity have been proposed [11]. See [22] for a thorough survey.

Typically, these pieces of work support *out-neighbour queries*, which return the

out-neighbours of a given vertex. As they are all based on adjacency lists, they usually store the transpose of the graph to provide efficient support for *in-neighbour queries* which return the in-neighbours of a vertex. Storing a graph and its transpose roughly doubles the space cost. To improve space efficiency, Brisaboa et al. [7] developed a web graph data structure called k^2-tree which takes advantage of large empty areas common in the adjacency matrix of a web graph to achieve compression. A single k^2-tree can support both in- and out-neighbour queries efficiently. The social networks structure by Maserrat and Pei [29] called MPk which is an Eulerian data structure based on the multiposition linearizations of directed graphs can also answer both queries efficiently.

Recently, Hernández and Navarro [22] proposed an approach that can support even more operations. This method first extracts dense subgraphs, including cliques and bicliques whose sizes are above certain thresholds, from the given web graph or social network. These subgraphs are then represented using succinct data structures, and the remaining graph is represented using an existing graph encoding method such as k^2-trees or MPk. Not only does this approach achieve great compression and query performance in experimental studies, it also supports a richer set of operations than previous results. In addition to the support of in- and out-neighbour queries, it also supports a set of queries that can be used to retrieve information of the extracted dense subgraphs. These queries include returning the cliques and bicliques stored in the compressed structure, counting the number of cliques (or bicliques) that contain a given vertex, and listing the dense subgraphs whose density is above a given threshold. These are called *mining queries*, as they are useful for data mining applications.

The succinct data structures that Hernández and Navarro used include wavelet trees [20], whose space cost dominates the cost of the representation of the extracted dense subgraphs. A wavelet tree can represent a sequence $S[1..n]$ drawn from the

alphabet $[\sigma]$ in $n \lg \sigma + o(n \lg \sigma)$ bits[1] to support the following three operations in $O(\lg \sigma)$ time: $\texttt{access}(S, i)$ which returns $S[i]$, $\texttt{rank}_\alpha(S, i)$ which returns the number of occurrences of symbol α in $S[1..i]$, and $\texttt{select}_\alpha(S, i)$ which returns the position in S that corresponds to the i-th occurrence of α. As a wavelet tree is a key component of many succinct data structures, a lot of efforts have been made to test its practical implementations [28, 3, 21, 16], and Hernández and Navarro used popular existing implementations in their experimental studies on compressed web graphs and social networks. However, previously the performance of wavelet tree implementations were typically tested for textual data only. A natural question to ask is whether they be further improved for purpose of representing extracted dense subgraphs. We thus study this problem and engineer the wavelet tree implementations used in compressed web graphs and social networks.

1.1 Our Work

A wavelet tree is conceptually a tree structure constructed over the alphabet of the sequence it represents, and a bit vector supporting `rank` and `select` operations is constructed for each tree level (see Chapter 3).

In Hernández and Navarro's experimental studies [22], the wavelet tree used to encode the extracted dense subgraphs is implemented as a complete binary tree encoded implicitly, and the bit vector for each level is encoded using a practical implementation [15] of the bit vector structure by Raman et al. [36]. In preliminary studies, we found that previous approaches of using different wavelet tree shapes to improve efficiency for textual data can not be applied directly to improve dense subgraph encoding. These approaches include Huffman-shaped [28] and canonical Huffman-shaped wavelet trees [16]. The reason is that in [22], a wavelet tree is used to encode a sequence over a very large alphabet, which requires a huge amount of structural data

[1] In this thesis, $[\sigma]$ denotes $\{1, 2, \ldots, \sigma\}$ and $\lg x$ denotes $\log_2 x$.

for these alternative tree shapes. To see why, observe that in [22], a wavelet tree is used to encode a sequence whose entries are vertices of the extracted dense subgraphs. As dense subgraphs have many vertices, the alphabet size is large compared to the alphabet of natural language texts. A large alphabet requires more structual data such as pointers to be stored to encode the tree shape, and for some approaches, extra information such as the Huffamn codes. For example, a Canonical Huffman-shaped wavelet tree, in our studies, triples the space cost of the wavelet tree, and the other tree shapes are even worse. We also used different implementations of bit vectors in wavelet trees, and achieved tradeoffs different from those in [22]. However, these tradeoffs are not attractive because they sacrifice too much compression or query performance. We discovered that bit vector implementations based on run-length encodings [21] can achieve better compression ratio, but the query time is increased by about five times to get improved space efficiency. This is consistent with the experimental studies on textual strings [21], but the query time is disappointing. Therefore, despite the existence of extensive works on wavelet tree encodings of textual data, it is hopeless to directly apply them to improve Hernández and Navarro's compressed web graph implementations.

To achieve better results, we design a strategy based on the following observation: bit vectors at different levels of the wavelet tree constructed for dense subgraphs have different properties. If we group bit vectors with similar properties and use the most suitable bit vector implementations for each group, we can potentially achieve better compression and/or query performance. Based on this idea, we engineer wavelet tree implementations used in compressed web graphs and social networks, and obtained a rich set of time/space tradeoffs that can not be achieved using existing approaches. The following four new tradeoffs are particularly interesting:

- A new wavelet tree encoding scheme that decreases the space cost of Hernández and Navarro's compressed web graph structure by 9% to 19% (more than 13%

for all but one graph in our study), while only doubling the query time.

- A new wavelet tree encoding scheme that decreases the space cost of their compressed web graph structure by 4% to 12% (10% or more for most graphs), with roughly the same query time.

- A new wavelet tree encoding scheme that decreases the space cost and the query time of their compressed web graph structure by about 2% and $1\% - 9\%$ (5% or more for most graphs), respectively.

- A new wavelet tree encoding scheme that decreases the query time of Hernández and Navarro's compressed social networks structure by 1% to 8%, with roughly the same space cost.

The idea of using different bit vector encoding schemes at different parts of a wavelet tree has been used before in the study of compressed text indexes [24, 26, 34]. Our work is the first to applies similar ideas to web graph representations.

1.2 Organization

The rest of the thesis is organized as follows. Chapter 2 describes the background knowledge of the research area. We introduce the notion of entropy and information-theoretic lower bound. We also discuss different prefix coding algorithms which are used in wavelet tree and bit vector data structures. Chapter 3 provides a survey on succinct representation of bit vectors and wavelet trees. Chapter 4 describes details about dense subgraph extraction, and how to represent dense subgraphs using compact data structures. In chapter 5, we perform experimented studies on wavelet tree representations of dense subgraphs, and engineer wavelet tree implementation for compressed web graph and social network representations. Chapter 6 is the conclusion of this thesis.

Chapter 2

Preliminaries

This chapter introduces some concepts in information theory and some prefix codes which are used in our work.

2.1 Succinct Data Structures, Information-Theoretic Lower Bounds and Entropy

Succinct data structures can represent data using less space than alternative solutions, though it typically requires more steps to perform an operation over a succinct data structure which may sacrifise the performance when the size of the data set is small. However, when the data are so large that they do not fit in main memory in their raw form, succinct data structures allow us to represent more data in main memory instead of spending a large amount of time accessing the disk, which may improve the performance. When analyzing succinct data structures, researchers often compare the space cost to information-theoretic lower bounds. Indeed, this kind of analysis was performed by Jacobson [25] when he first introduced succinct data structures to encode bit vectors, trees and graphs. The information-theoretic lower bound of representing a combinatorial object consisting of n elements is $\lceil \lg k \rceil$ bits, where k is the number of such objects. For instance, there are 2^n different bit vectors of length n. Thus the lower bound of representing a bit vector of length n is $\lceil \lg 2^n \rceil = n$ bits. There are σ^n different string of length n drawn from alphabet $[\sigma]$, so the information-theoretic lower bound is $\lceil n \lg \sigma \rceil$ bits.

As natural language text is typically compressible, researchers also use empirical

entropy to measure the compactness of succinct data structures for strings and text. The following is the definition of the zero-order empirical entropy of a string:

Definition 1 *The Zero-order empirical entropy of a string S of length m over alphabet $[\sigma]$ is defined as* $H_0(S) = \sum_{\alpha=1}^{\sigma} (p_\alpha \lg \frac{1}{p_\alpha}) = -\sum_{\alpha=1}^{\sigma} (p_\alpha \lg p_\alpha)$

where p_α is the frequency of character α in S, and $0 \lg 0$ is defined to be 0.

We use $H_0(S)$ to refer to H_0 when it is clear from the context. It is well-known that nH_0 is the minimum number of bits required to encode a string if we assign a unique fixed-length code to each alphabet symbol

Note that $nH_0(S) \leq n \lg \sigma$, and the maximum value is reached when all characters have the same frequency.

Researcher have further defined the k-th order empirical entropy for any positive integer k.

Definition 2 *Consider a String S of length n over alphabet $[\sigma]$. For any string $w \in [\sigma]^k$, the string w_s is the concatenation of all the single characters directly following one of the occurrences of w in S. The k-th order empirical entropy of S is*

$$H_k(S) = \frac{1}{|S|} \sum_{w \in [\sigma]^k} |w_s| H_0(w_s)$$

We use $H_k(S)$ to refer to H_k when it is clear from the context. $nH_k(S)$ is a lower bound on the compression that can be achieved by compressors that assign a code to a character based on the k characters that precede it.

2.2 Prefix Coding

A prefix code is a variable-length code system with the property that no codeword is a prefix of any other codeword in the system. This property guarantees that an encoded sequence, when stored as a bit string, can always be uniquely decomposed

into codewords without requiring a special sign between codewords. Here we survey several popular prefix codes: Huffman codes and universal codes.

Huffman codes Huffman codes were developed by David A. Huffman [23]. The idea behind Huffman coding is to use fewer bits to encode symbols that are more frequent. Given a set of symbols and their weights, we can find a set of codewords with minimum expected codeword length. The algorithm that generates Huffman codes works by creating a binary tree called Huffman tree. In this tree, all alphabets symbols are stored in leaf nodes, whose weights are the numbers of occurrences of the symbols in the text. The weight of an internal nodes is the sum of the two child nodes' weights. To generate a Huffman tree, first we calculate the frequency of each symbol and create a leaf node for each symbol associated with its frequency. We then add all the leaves into a priority queue, and nodes with smaller weights have higher priority. The algorithm then removes two nodes with smallest weights (x and y denote these two nodes), creates a new internal node z with weight equal to the sum of the weights of x and y, makes z the parent of x and y, and then adds z to the queue. We repeat these steps until there is only one root node left in the queue. To compute the Huffman code of a symbol, we traverse the Huffman tree from the root node to the leaf storing this symbol. Each time we traverse to the left child or right child, we append a 0 bit or a 1 bit to the result, respectively. When we reach the leaf containing the symbol, the bits generated will be the Huffman encoding of the symbol. For instance, Figure 2.1 is a Huffman tree constructed for a string S=AAAAAAAAAABBBBBCCCCDDDDDDDDDDDDDDDD. You can see that the most frequent letter 'D' only requires one bit to encode. 'C' and 'B' use 3 bits each.

The decoding procedure is simple. We re-use the Huffman binary tree. The decoding algorithm starts with first bit in the stream and determines whether to go

Symbol	Code
D	0
A	11
C	100
B	101

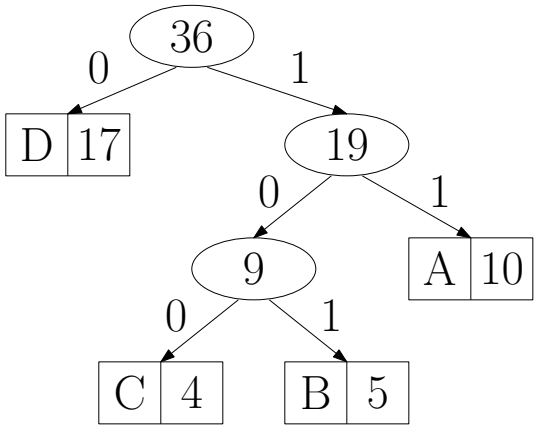


Figure 2.1: Huffman Coding for string S.

left or right in the binary tree based on the bits from the stream. When a leaf of the Huffman binary tree is reached, a character is decoded. Then the algorithm repeats these steps with the next bit as the first bit of the next character.

A Canonical Huffman code is a special type of Huffman code. It ensures that codes of the same length are consecutive and shorter codes have smaller values than longer codes. Figure 2.1 is the normal Huffman codes. To get the canonical version, we sort the Huffman codewords by codeword length and secondly by alphabetical value, and in this example we will have D=0, A=11, B=101 and C=100. Then each of the existing codes is replaced by a new one of the same length as follows: The first symbol in the sorted sequence is assigned a codeword of all 0s, and the number of zeros is equal to the length of its original codewords. Each subsequent symbol is assigned the next greater binary number. If the codeword of the current symbol is larger than the codeword of the previous symbol, we perform a left shift on its codeword. Thus the canonical version of our example will be D=0, A=10, B=110 and C=111.

Universal Codes Universal codes are another type of prefix coding that convert non-negative integers into binary codewords. Here we describe three different schemes of generating universal codes. The first one is Elias gamma coding or gamma coding

which was developed by Peter Elias [17]. To generate the Elias gamma code of a positive integer x, first we calculate $N = \lfloor \lg(x) \rfloor$. Then the gamma code of x is the binary expression of x prepended by N zeros. Thus Elias gamma uses $2\lfloor \lg(x) \rfloor + 1$ bits to represent x. For example, The Elias gamma code of the integer 11 is 0001011. To decode a gamma code, we read and count the number of 0s from the bit stream until we read a 1 bit. N is equal to the number of 0s read. Then we read the remaining N bits which are then prepended by a 1 bit, obtaining the binary expression of x.

The second encoding scheme is Elias delta coding. To code a number $x \geq 1$, we calculate $N = \lfloor \lg x \rfloor$ and separate x into two parts, 2^N and $x - 2^N$. We use Elias gamma code to encode the number $N + 1$. Thus, the delta code of x is the binary expression of the gamma code of $N + 1$ appended by the binary expression of $x - 2^N$. Elias delta code uses $\lfloor \lg(x) \rfloor + 2\lfloor \lg(\lfloor \lg(x) \rfloor + 1) \rfloor + 1$ bits to represent x. For instance, the binary representation $x = 11$ is 1011, and $N = \lfloor \lg(x) \rfloor = 3$. The gamma code of the number $N + 1$ is 00100. Thus the delta code of 11 is 00100 011. To decode a delta code, we first decode $N + 1$ from its gamma code. Then, we read N bits from the stream and prepend them by 1 to get the binary expression of x.

The last prefix code is Elias omega coding. To encode a number x, we first put a bit 0 at the end of the encoding as a delimiter. If the number $x = 1$. stop; otherwise we prepend the binary expression of x to the result. Then we count the number, y, of bits we just prepended and set x to $y - 1$. We recursively repeat the previous steps to encode y and prepend the encoding to the result. Take number 11 as an example. We write down a bit 0 in the end of representation. We prepend the binary expression of 11 to get 1011 0. Since we prepended 4 bits in the last step, $y = 4$. Thus x becomes to 3 and its binary expression is 11. We prepend it again and get a new result 11 1011 0. Then number of bits prepended is 2, and x becomes 1, so we stop. Thus, the omega code of integer 11 is 11 1011 0. We can reverse these steps for decoding.

Chapter 3

A survey on Bit Vectors and Wavelet Trees

This chapter is a survey of bit vector and wavelet tree representations in both theory and practice. They are both key data structures used in our graph representations.

3.1 Bit Vector

3.1.1 Theoretical Results

Jacobson [25] first studied succinct representations of bit vectors. Given a bit array B of length n, in which the positions are numbered $0, 1, \ldots, n-1$, and $b \in \{0, 1\}$, Jacobson considered the support of the following operations:

- $\texttt{rank}_B(b, i)$: the number of occurrences of b in $B[0, i]$;
- $\texttt{select}_B(b, i)$: the position of the i^{th} b in B.

Jacobson [25] showed how to represent B using $n+o(n)$ bits to support both operations in $O(\lg n)$ time under the pointer machine model. Later, Clark and Munro [12] showed how to support both operations in constant time under the word RAM model using $n + o(n)$ bits. Raman et al. [36] proposed a succinct indexable dictionaries structure that can support both operations in $O(1)$ time under the word RAM model using $nH_0(B) + o(n)$ bits of space. Later, Sadakane and Grossi [38] showed how to use $nH_k + O(n(\log \sigma + \log\log_\sigma n + k)/\log_\sigma n) + o(n \log \sigma)$ bits to support `rank` and `select` in $O(1)$ time. Recently, Mihai Pătraşcu [35] showed how to represent a bit vector of length n with m 1s using $m \log \frac{n}{m} + O(nt^t/\log^t n + n^{3/4})$ bits, and support both `rank` and `select` operations in $O(t)$ time.

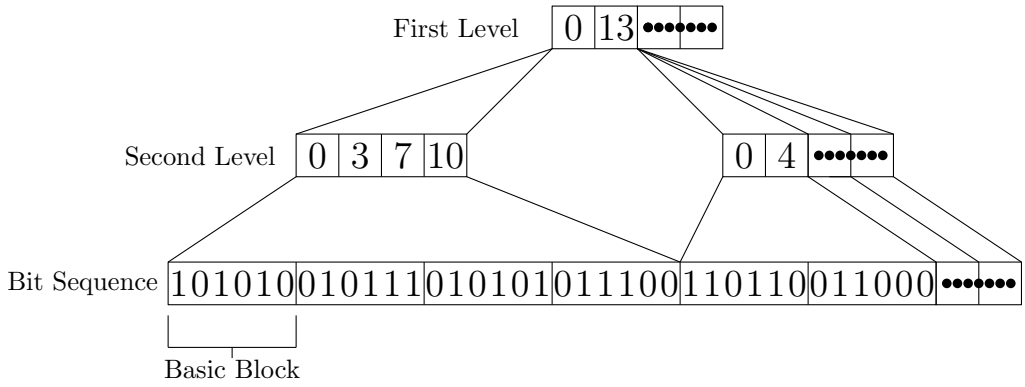


Figure 3.1: Rank Structure with two layers

3.1.2 A General Framework of Implementation

As bit vectors supporting `rank` and `select` are key structures used in many succinct data structures, much work has been done to engineer the implementation of succinct representations of bit vectors [18, 36, 33]. In most of the previous experimental studies, researchers use the following general framework to support `rank`, as summarized in [42]:

1. Divide the bit sequence into fixed-length *basic blocks*. These basic blocks are at the lowest level of partitioning, and we need to have an approach that can count the number of bits inside a basic block up to any given position in this block. For instance, the popcounting method is a popular approach that can quickly count the number of ones in an integer.

2. Consecutive basic blocks are grouped into *superblocks*. Superblocks maintain the information about the number of one bits in previous superblocks and basic blocks maintain the number of 1s in previous basic blocks that are in the same superblock.

Figure 3.1 illustrates such a two-level rank structure. In this example, we divide a bit sequence into basic blocks, and each basic block contains 6 bits. Superblocks (first level) contains 4 consecutive basic blocks at the second level.

To answer $\texttt{rank}_B(b, i)$, we perform the following steps:

1. Start by finding which superblock index i belongs to. We can easily know which superblock it is by dividing i by the size of the superblock. Then we know x, which is the number of 1s in the superblocks before the superblock containing i.

2. Next, we look into the second layer to find y, which is the number of 1s that appears before the basic block containing i within the superblock found at the first level.

3. Last, count the numbers of 1 bits, z, from the starting position of the basic block containing position i up to the position i.

4. The result of $\texttt{rank}_B(b, i)$ is $x + y + z$.

It is possible to further group superblocks to design rank structures of more than two levels. However, González et al. [19] did experimental evaluation that showed one or two-level rank structures use less space and support faster operations than structures of three levels or more.

For the `select` operation, practical solutions typically do not directly implement the theoretical solutions that provide constant-time support. There are two general methods, namely rank-based select query and position-based select query [42].

For the rank-based select query, we use the same rank structure. To compute $\texttt{select}_B(b, i)$ we perform binary search among superblocks, and then do sequential search again inside the basic block we found to compute the answer. For example, if we want to know $\texttt{select}_B(1, 9)$ in Figure 3.1, we perform binary search among the superblocks and find it is inside the first superblock. Then we perform binary search again at the second level, and find that it belongs to the third block. Finally, we can perform a search to answer the query.

For position-based selection, they sample answers of `select` queries. The sampled answers divide B into *select blocks*, we perform binary search on samples to find the select block containing the answer, and then perform a sequential search to find the correct answer in the select block.

3.1.3 Practical Implementations

We used different bit vector encoding schemes in our thesis, including run-length encoding [21] (`RLE`, including `RLE`+γ , `RLE`+δ, `RLE`+ω), `RRR` (Raman, Raman and Rao's structure [36]), and `Plain` (bitmap in plain format without any compression) [33].

RLE

Grossi et al. [21] used run-length encodings (`RLE`) to implement bit vectors. `RLE` is a simple compression algorithm that encodes a run of the same symbol as a pair (t, s), where t is number of times that symbol s occurs consecutively. The symbols in a binary string are either 0 or 1. Let $B = B[0 \cdots n-1]$ be a binary string of size n. To compress B using `RLE`, we regard B as a sequence of runs of identical bits $b_1^{t_1} b_2^{t_2} b_3^{t_3} \cdots b_m^{t_m}$ where $m \leq n$, and $b_x \neq b_{x+1}$ for $1 \leq x < m$. Then the `RLE`+γ encoding of B is $b_1 \gamma(t_1)\gamma(t_2)\cdots\gamma(t_m)$, where b_1 identifies the first bit as either 1 or 0, and $\gamma(t_i)$ is the Elias gamma coding of t_i. `RLE` $+ \delta$ and `RLE` $+ \omega$ work in a similar way, and the only difference is that they use Elias delta and Elias omega codes to encode t_i.

To support operations, we divide B into blocks containing the same numbers of runs, and store additional information for each block. *Block size* is defined as the number of runs in a block. For each block, we store information about how many 0s and 1s there are in B before this block. We also store the ending position of each block in the compressed bit vector. To support $\texttt{rank}_B(1, i)$ and $\texttt{select}_B(1, i)$, we then perform binary searches to find the blocking containing $B[i]$, retrieve the number of 1s before this block, and decompress the block and perform a sequential search to

compute the answer.

We use `RLEG`, `RLED` and `RLEO` to refer to bit vectors based on `RLE`+γ, `RLE`+δ and `RLE`+ω encodings, respectively, while `RLE` can refer to any of them. We suffix `RLEG`, `RLED` or `RLEO` by the block size when needed. For example, `RLEG64` is a bit vector implementation based on `RLE`+γ with block size 64.

RRR

We now introduce practical implementations [15, 33] of the bit vector structure by Raman et al. [36]. These implementations store B in a compressed form as follows: B is split into blocks of length $k = \frac{\lg n}{2}$. Blocks are numbered consecutively starting from 0, and block i refers to the ith block. Each block is represented as a pair (c_i, o_i). Here c_i, is called the class of the block, which encodes the number of 1s in the block, while o_i is called the offset of the block, which uniquely identifies the block among all possible blocks within class c_i. Hence o_i can be encoded using $\lceil \lg \binom{k}{c_i} \rceil$ bits. Thus sparse blocks use less space, which achieves compression for spare bit vectors. The implementation by Claude and Navarro [15] constructs three tables: Table E, the universal table, stores all possible bit vectors of length k, sorted by class and by offset within bit vectors of the same each class. Table C stores the concatenation of all c_i. Since the number of 1s in each block is between 0 and k, c_i requires $\lceil \lg (k+1) \rceil$ bits to encode. Table O stores the concatenation of all o_i. The total space of the table O is bounded by $nH_0 + \frac{n}{k}$ as proved in [12], and the space of table C is bounded by $O(n\frac{\lg k}{k})$. In addition, we group every sr blocks into a superblock. Superblocks are numbered consecutively starting from 0, and superblock i refers to the ith superblock. This structure requires two arrays, $sumC$ and $sumO$. The entry $sumC[i]$ stores $\sum_{j=0}^{i \times sr-1} c_i$ which is the number of 1s before superblock i, and $sumO[i]$ contains $\sum_{j=0}^{i \times sr-1} \lceil \lg \binom{k}{c_i} \rceil$ which is the starting position of the encoding of o_i in Table O. In addition, we have a table called $classIndex$ of length k, in which

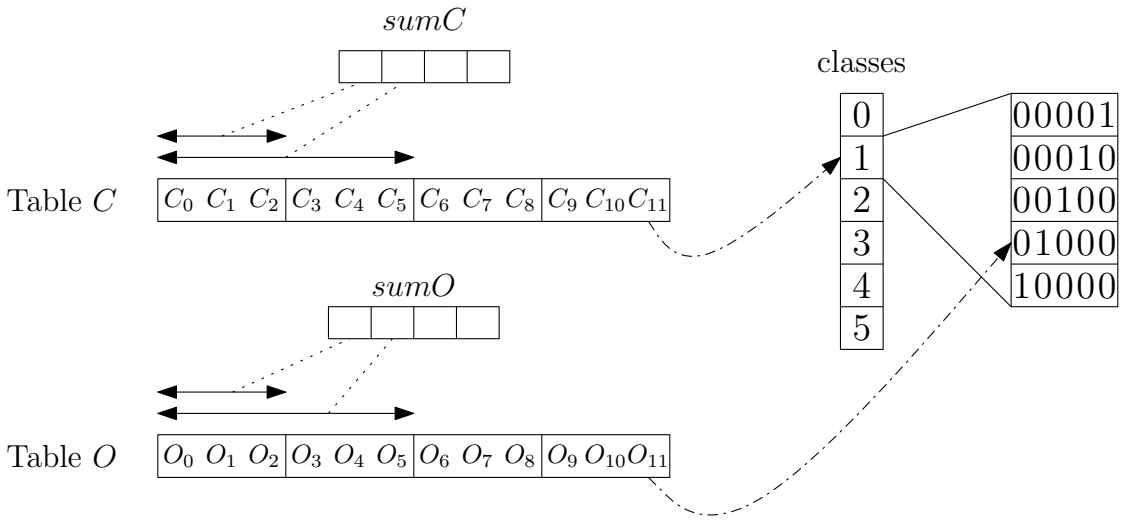


Figure 3.2: RRR Example

$classIndex[i]$ stores the index of the first entry of E that stores a bit vector of class i. In the implementation in [15], they set $k = 15$, so there are 16 classes. Thus each c_i only requires 4 bits. Table E has 2^{15} entries and each entry stores a 16-bit integer. Thus E only takes $2^{15} \times 16$ bits $= 64$ KB.

Figure 3.2 shows an example, in which we have a bit sequence of length 60, and block size is $k = 5$. Every three blocks are grouped into one superblock.

To answer $\texttt{rank}_B(b, i)$:

1. Identify the superblock i_s containing position i by the identity $i_s = \lfloor \frac{i}{sr \times k} \rfloor$
2. Identify the block i_b containing position i by the identity $i_b = \lfloor i/k \rfloor$.
3. Let x be the number of 1s in B that appear before superblock i_s and $posO$ be the starting position of the encoding of o_{i_s} in table O. Then $x = smuC[i_s]$ and $posO = sumO[i_s]$
4. Let y be the number of 1s in B that appear between block $i_s \times sr$ and block $i_b - 1$. Then increase the value of $posO$ by $\sum_{i_s \times sr}^{i_b - 1} \lceil \lg \binom{k}{c_i} \rceil$, and $y = \sum_{i_s \times sr}^{i_b - 1} c_i$. Each c_i can be retrieved from C in constant time, as the entries of C are encoded in the same number of bits.

5. Let z be the number of 1s that appear from the starting position of block i_b to position i. We read o_{i_b} from $O[posO..posO + \lceil \lg \binom{k}{c_i} \rceil]$, and retrieve the content of block i_b from $E[o_{i_b} + classIndex[c_{i_b}]]$. Then scan the bits of the block i_b to compute z.

6. The result of $\texttt{rank}_B(b, i)$ is $x + y + z$.

To answer $\texttt{select}_B(b, i)$ we use the same structures. We perform a binary search among superblocks and blocks, and then perform a sequential search inside blocks until the i-th 1 bit.

Claude and Navarro [15]'s implementation sets $k = 15$, but the space cost of table E is prohibitive for much larger k. Later, Navarro and Providel [33] came up with an implementation that does not store table E, and they could set the block size to 31 and 63. In their implementation, they compute each entry of E on the fly, so that E need not be explicitly stored. Due to technical reasons, k should be at most 1 less than the word size, and thus 63 is the maximum value in 64-bit machines.

We call these implementations `RRR` collectively, and we use `RRR15`, `RRR31` and `RRR63` to refer to implementations in which block sizes are 15 (in [15]), 31 and 63 (in [33]), respectively.

Plain

Navarro and Providel [33] introduced a structure that answers `rank`/`select` queries. It can be viewed as an implementation of Clark and Munro's theoretical result [12] which represents a bit vector $B[1..n]$ in $n + o(n)$ bits to support operations in $O(1)$ time. As in previous implementations, answers for some of the `rank` and `select` queries are precomputed, which are called *rank samples* and *select samples*. The novel part is that when supporting `select`, they not only make use of `select` samples as in previous work, but also use rank samples to speed up the operation. This idea

allows them to use little space overhead in addition to the n bits storing B in its uncompressed form. More precisely, they construct a one-level structure following the general framework in Section 3.1.2 to support `rank`. They then modify position-based selection: Each two consecutive select samples are separated by a fixed number of 1s, so that they can locate the select block containing the answer by simple arithmetic. After that, when performing a linear scan to find the final answer, they do not always scan bit by bit; instead, they use rank samples to compute the number of 1s for these basic blocks that appear before the answer and are entirely contained in the select block. We refer to this implementation as `Plain` in this paper.

3.2 Wavelet Trees

3.2.1 Definitions

Wavelet trees were proposed by Grossi et al. [20] to represent a string $S[1..n]$ over alphabet $[\sigma]$ [1], to support the following operations:

- $\texttt{rank}_c(S, i)$: return the number of occurrences of symbol c in $S[1..i]$;
- $\texttt{select}_c(S, i)$: return the position of the i-th occurrence of symbol c in S;
- $\texttt{access}(i)$: return the symbol at position i.

A wavelet tree stores n bits at each level and the last level contains at most n bits. The height of the wavelet tree is $\lceil \lg \sigma \rceil$. It contains $\sigma - 1$ internal nodes and σ leaves. There are many different variants of wavelet trees, and we describe them in the rest of this section.

[1] $[\sigma]$ denotes $\{1, 2, \cdots, \sigma\}$.

3.2.2 Pointer-based Normal-shaped Wavelet Trees

A pointer-based normal-shaped wavelet tree [20] can be constructed in a top-down fashion as follows (assume that σ is a power of 2 for simplicity): The root corresponds to the entire alphabet, and a bit vector is constructed for the root by iterating through S, writing down a 0 each time we visit an entry of S storing a symbol from $[1..\sigma/2]$, and a 1 otherwise. The left (right) child of the root represents the subsequence of S that corresponds to 0s (or 1s) in the bit vector at the root level, and the subtree rooted at this node is constructed recursively.

Bit sequences in the wavelet trees can be encoded by different compact data structure in order to support fast `rank` & `select` operations. Figure 3.3 gives an example, which is wavelet tree for sequence $S = \{3\ 5\ 1\ 6\ 11\ 0\ 3\ 7\ 10\ 2\ 8\ 0\ 4\ 9\ 4\ 11\ 5\ 6\}$ of length $n = 18$. The alphabet for this case is $\{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\}$, and σ is 12.

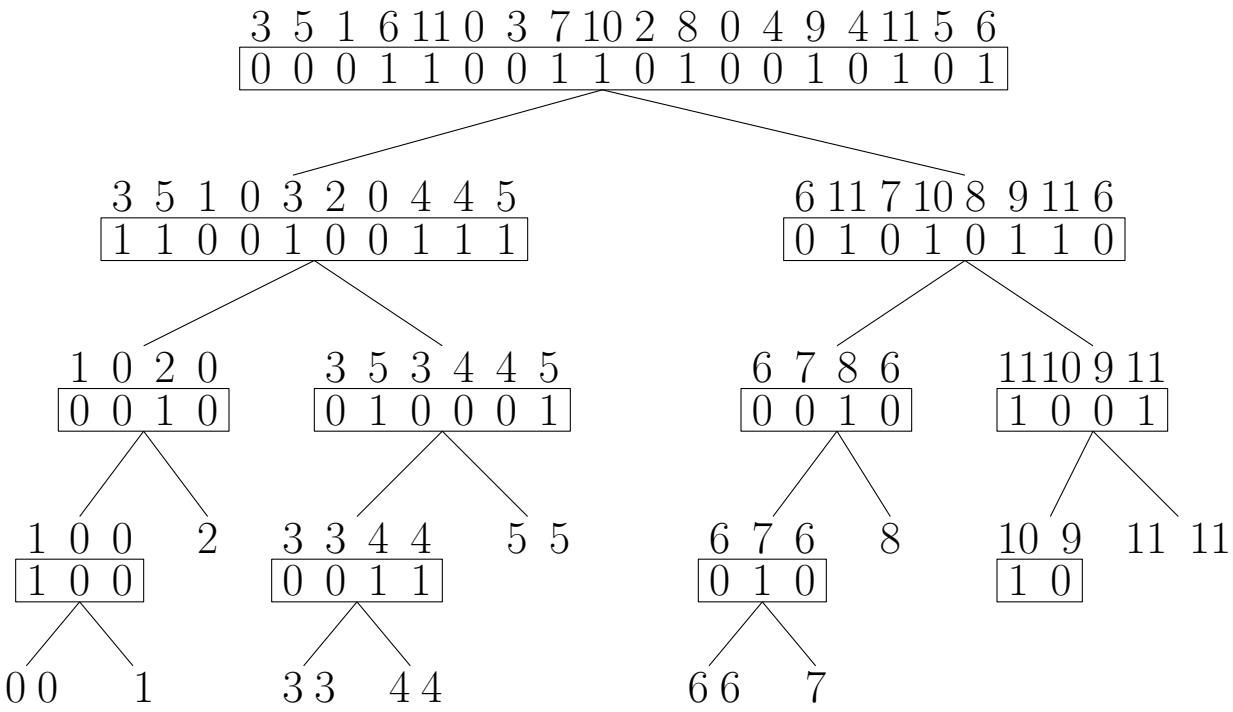


Figure 3.3: A pointer-based normal-shaped wavelet tree.

Let B_v be the bit vector stored for node v, and let v_l and v_r be the left and right children of v, respectively. Algorithm 1 gives the pseudocode of `rank`, and we call $\texttt{rank}(B_r, c, i)$, where r is the root of the wavelet tree, to compute $\texttt{rank}_c(S, i)$. Lines 2-3 check whether the node v is a leaf or not. If v is a leaf, the algorithm returns the value i. Otherwise, lines 4-8 decide whether to visit v_l or v_r depending on whether c is in $alphabetSet(v_l)$, where $alphabetSet(v_l)$ is the alphabet of the subsequence corresponding to v_l. To visit v_l, the algorithm calls $\texttt{rank}(B_{v_l}, c, \texttt{rank}_{B_v}(0, i))$ recursively, where the $\texttt{rank}_{B_v}(0, i)$ is the number of occurrences of 0 in $B_v[0, i]$ (thus $B_{v_l}[\texttt{rank}_{B_v}(0, i)]$ corresponds to $B_v(i)$). Otherwise, the algorithm calls $\texttt{rank}(B_{v_r}, c, \texttt{rank}_{B_v}(1, i))$ recursively, where the $\texttt{rank}_{B_v}(1, i)$ is the number of occurrences of 1 in $B_v[0, i]$ (thus $B_{v_l}[\texttt{rank}_{B_v}(1, i)]$ corresponds to $B_v(i)$).

Algorithm 1 Algorithm for `rank`

```
1: function rank(B_v, c, i)
2: if v is a leaf then
3:     return i
4: else if c ∈ alphabetSet(v_l) then
5:     return rank(B_{v_l}, c, rank_{B_v}(0, i))
6: else
7:     return rank(B_{v_r}, c, rank_{B_v}(1, i))
8: end if
9: end function
```

Algorithm 2 gives the pseudocode of $\texttt{select}(B_v, c, i)$, which computes the position in B_v that corresponds to the i-th occurrence of symbol c in S. Thus, $\texttt{select}(B_r, c, i)$ computes the answer to $\texttt{select}_c(S, j)$. In this piece of pseudocode, lines 2-3 are for the base case, in which v is the leaf corresponding to symbol c. In this case the algorithm returns i. Lines 4-8 are recursive cases. The algorithm decides whether to visit v_l or v_r depending on whether c is in $alphabetSet(v_l)$, where $alphabetSet(v_l)$ is the alphabet of the subsequence corresponding to v_l. If it is, the algorithm calls $\texttt{select}_{B_v}(0, \texttt{select}(B_{v_l}, c, i))$, where the $j_v = \texttt{select}(B_{v_l}, c, i)$ return the position

in B_{v_l} that corresponds to the i-th occurrence of symbol c in the subsequence corresponding to the node v_l. Then $\texttt{select}_{B_v}(0, j_v)$ returns the position in B_v that corresponds to the j_v-th occurrence of bit 0 in B_v. This gives the answer because the occurrences of c in S corresponds to 0 bit in B_v. Otherwise, algorithm calls $\texttt{select}_{B_v}(1, \texttt{select}(B_{v_r}, c, i))$.

Algorithm 2 Algorithm for `select`

```
1: function select(B_v, c, i)
2: if v is a leaf then
3:    return i
4: else if c ∈ alphabetSet(v_l) then
5:    return select_{B_v}(0, select(B_{v_l}, c, i))
6: else
7:    return select_{B_v}(1, select(B_{v_r}, c, i))
8: end if
9: end function
```

A wavelet tree can be used to encode a string using $n\lceil \lg \sigma \rceil + o(n \lg \sigma) + O(\sigma \lg n)$ bits to support queries in $O(\lg \sigma)$ time, because the bitmaps representing the nodes use $n\lceil \lg \sigma \rceil$ bits in total, the little-o term covers the extra cost of the rank and select structures, and the tree pointers and pointers to the bitmaps requires $O(\sigma \lg n)$ bits. If the bit vector at each node is compressed by `RRR`, the overall space cost is bounded by $nH_0 + o(n \lg \sigma) + O(\sigma \lg n)$ bits where the bitmaps representing the nodes use $nH_0 + o(n \lg \sigma)$ bits in total, and the query time remains $O(\lg \sigma)$.

3.2.3 Pointerless Normal-Shaped Wavelet Trees

In a pointerless normal-shaped wavelet tree [15], we do not store the tree structure explicitly using pointers. Instead, we concatenate the bit sequences constructed for the nodes at the same level into a single bit vector of length at most n, and represent each of these $\lceil \lg \sigma \rceil$ concatenated bit vectors succinctly to support `rank` and `select` operation on it.

To be able to use the algorithm given in Section 3.2.2 to support `rank` and `select`

operations over the string represented by the wavelet trees, it suffices to show that, given a node v representing symbols $[i, i+1, \cdots, j]$, we can locate, in $O(1)$ time, the starting and the ending positions of B_v in the concatenated bit vector constructed for v's level. To achieve this, we construct two bit vectors Occ and SA. $SA[1, \sigma]$ identifies which symbols occur in the sequence. $SA[i] = 1$ if and only if a symbol i appears in the sequence. If all the alphabet symbols appear in S, then SA is not needed. Let n_i be the number of occurrences of symbol i, and p be the ith smallest symbol in σ that occurs in the sequence S. $Occ[1, n]$ stores n_i by storing a 1 bit at $\sum_{i=1}^{p} n_i$. All the remaining bits in Occ are 0s. For example, for the sequence *aaccbdb*, $Occ = 0101011$. By performing `rank`/`select` operations on SA and Occ, we can find out, given a symbol i, the number of characters in S that are smaller than i. Such information can be further used to determine the starting and ending positions of B_v for a node v in the concatenated bit vectors constructed for v's level.

Figure 3.4 is a pointerless normal-shaped wavelet tree. Using just one bitmap at each level, we do not need pointers, $O(\sigma \lg n)$, for the topology. Thus a pointerless normal-shaped wavelet tree can be used to encode a string using $n\lceil \lg \sigma \rceil + o(n \lg \sigma) + \sigma + o(\sigma) + n + o(n)$ bits to support queries in $O(\lg \sigma)$ assuming `rank` and `select` operations in constant time. The terms $\sigma + o(\sigma) + n + o(n)$ account for the space cost of SA and Occ.

3.2.4 Pointer-based Huffman-Shaped Wavelet Trees

Claude et al. [16] proposed a pointer-based Huffman-shaped wavelet trees. To construct a pointer-based Huffman-shaped wavelet tree to represent S, we first construct a Huffman tree for S using the approach describe in Section 2.2. This Huffman tree is encoded explicitly using pointers, and it gives us the structure of the wavelet tree. We also store the Huffman code of each symbol. We then construct bit vectors for wavelet tree nodes as follows: The root corresponds to the entire alphabet, and a bit

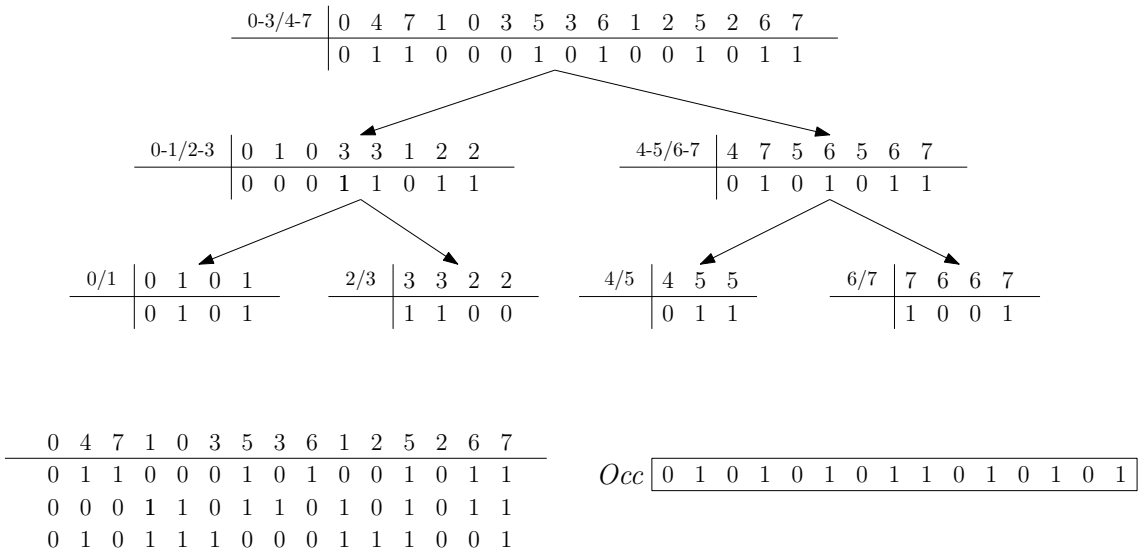
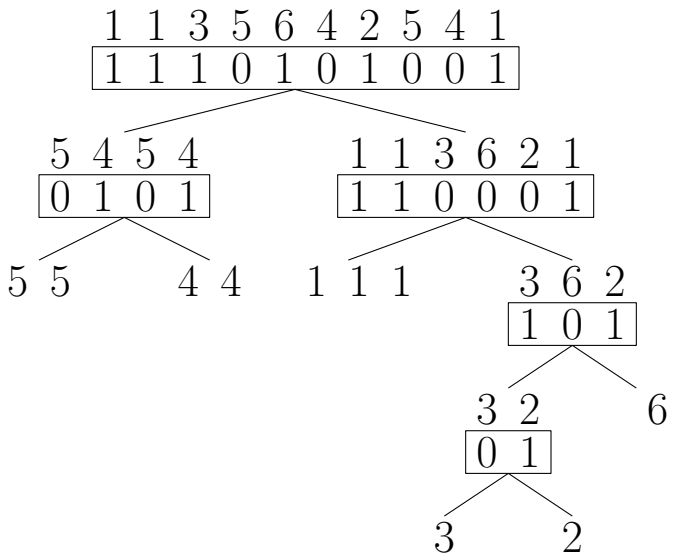


Figure 3.4: A pointerless normal-shaped wavelet tree

vector is constructed for the root by iterating through S, writing down a 0 each time we visit an entry of S storing a symbol if the first bit of its Huffman code is a bit 0, and a 1 otherwise. The left (right) child of the root represents the subsequence of S that corresponds to 0s (or 1s) in the bit vector at the root level, and the subtree rooted at this node is constructed recursively by using the next bit of the Huffman code of each symbol. As the expected length of a root-to-leaf path in a Huffman tree is upper bounded by $H_0(S) + 1$, the average time for `access`, `rank` and `select` is $O(1 + H_0(S))$ using the same algorithm in section 3.2.2. This is better than the $O(\lg \sigma)$ query time of a normal-shaped wavelet tree when the entropy is small. The total number of bits stored in a Huffman-shaped wavelet tree is exactly the output size of the Huffman compressor, which is upper bounded by $n(H_0(S) + 1)$. Therefore, using the plain bit vectors representations, the total space is upper bound by $(H_0(S) + 1)(n + o(n)) + O(\sigma \lg n)$ bits. The term $O(\sigma \lg n)$ accounts for the pointers and for the permutation of symbols induced by the code. Figure 3.5 is an example of a pointer-based Huffman-shaped wavelet tree and the Huffman code of each symbol.



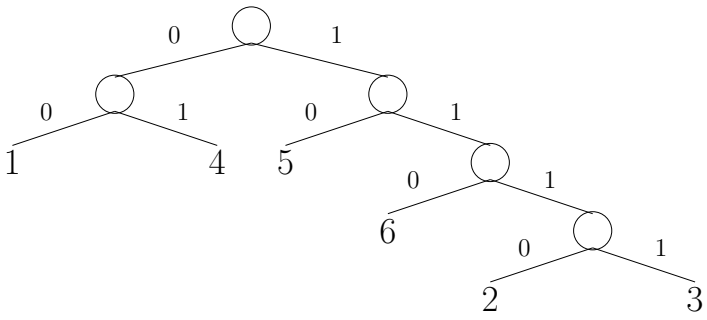
symbol	Code
5	00
4	01
1	11
6	100
3	1010
2	1011

Figure 3.5: Pointer-based Huffman-shaped wavelet tree example

3.2.5 Pointerless Canonical Huffman-shaped wavelet trees

Claude et al. [16] proposed a Canonical Huffman-Shaped wavelet tree. A Huffman tree can be viewed as bit sequence of Huffman codes, and a Canonical Huffman tree can also be viewed as a trie constructed over Canonical Huffman codes which are introduced in Section 2.2. It is also possible to construct a Canonical Huffman tree without using the approach described in Section 2.2 to compute canonical Huffman codes first. We first compute a Huffman tree of the string. Then we transform it into a Canonical Huffman tree as follows: We sort the symbols by their frequencies in decreasing order to create a Huffman tree first, and then we traversal the Huffman tree level by level to sort the symbols by alphabet order at the same level

A Canonical Huffman-shaped wavelet tree is essentially a Huffman-shaped wavelet tree built over the Canonical Huffman tree of S instead of an arbitrary Huffman tree of S. Properties of Canonical Huffman trees can be used to save space overheads: At any level of a Canonical Huffman tree, all the leaf nodes appear to the left of internal nodes. Therefore, to encode the tree structure, we need not use pointers to encode



Symbol	Code
1	00
4	01
5	10
6	110
2	1110
3	1111

Figure 3.6: Canonical Huffman Codes

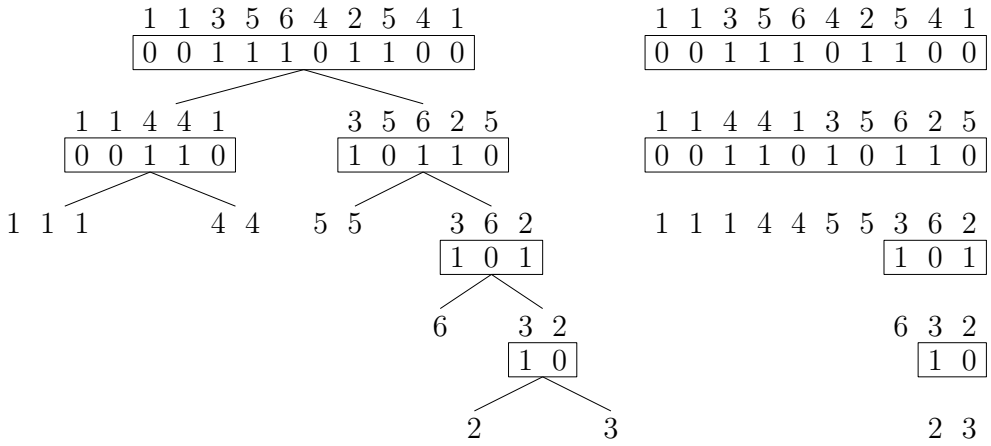


Figure 3.7: Canonical Huffman Wavelet Tree

edges. Instead, it suffices to encode the number of leaves at each level, requiring only $\lg n$ bits per level. Claude et al. [16] further bounded the length of Huffman codes by $\lg n$, by rebalancing Huffman trees below a certain level. This guarantees that there are at most $\lg n$ levels in a Canonical Huffman tree, and it requires at most $\lg^2 n$ bits to encode the tree structure. They also use various approaches proposed by Navarro and Ordóñez [32] that can encode Canonical Huffman codes in a compact manner. Figure 3.7 is an example by using Canonical Huffman codes to build the Canonical Huffman-shaped wavelet trees.

A Canonical Huffman-shaped wavelet tree can still support `access`, `rank` and `select` in $O(H_0(S)+1)$ time. Thus it provides the same support for operations while using less space, when compared to a Huffman-shaped wavelet tree.

Chapter 4

Compressed Graph Representations via Dense Subgraphs Extraction

In this chapter, we describe the compressed representations of web graphs and social networks proposed by Hernández and Navarro [22]. As mentioned before, their strategy is to extract dense subgraphs and then use succinct data structures to represent them. Thus, in Section 4.1, We described their approach to dense subgraph extraction, which was originally proposed by Buehrer and Chellapilla [9]. We then describe the compressed representations in Section 4.2 and the query algorithms in section 4.3.

4.1 Extracting Dense Subgraphs

Let $G = (V, E)$ be a directed graph. Hernández and Navarro [22] gave the following definition of dense subgraphs:

Definition 3 *A dense subgraph $H(S, D)$ of a directed graph $G = (V, E)$ is a subgraph whose vertices are $S \cup D$ and edges are $S \times D$, where $S, D \subseteq V$.*

They then augment each vertex of G with a self-loop, and use a mining algorithm based on [9] to detect dense subgraphs, including bicliques (when $(S \cap D = \emptyset)$), cliques with self-loops on each vertex (when $S = D$) and general dense subgraphs where S and D are neither equal nor disjoint. If no self-loops are present in G , then all dense subgraphs are bicliques. Indeed, this mining algorithm was initially proposed to find bicliques only [9]. Hernández and Navarro [22] choose to add a self-loop,

(s, s), for each vertex s in G that is not incident to a self-loop. They then discover dense subgraphs of all types in the augmented graph. This strategy allows them to extract more subgraphs. In addition, they add a bitmap of length $|V|$ to identify which nodes have self-loops in the original graph. The mining algorithm contains two major phases for finding dense subgraphs, namely the clustering phase and the pattern mining phase [9].

4.1.1 Clustering

Clustering aims to group together vertices that have similar outlinks. The algorithm in [8] first chooses k independent hash functions. Here k is called the *hash factor* of the mining algorithm. Then the algorithm iterates through each adjacency list, and computes a hash value H corresponded with each edge of the adjacency list k times and choose the k smallest hashes associated to each adjacency list; each item in the adjacent list is the ID of the terminal vertex of the corresponding edge. Next, the algorithm generates a $|V| \times k$ matrix F, in which $F[i, j]$ stores the minimum among all the hash values that the items in the i-th adjacency list are mapped to by the j-th hash function. We then sort the rows of the adjacency matrix by the rows of F in lexicographic order, and cluster the sorted adjacency matrix into groups, such that each group contains vertices and their adjacency lists whose corresponding rows in F store the same values. It requires $O(k|E|)$ time to compute all the hash values, and the sorting requires $O(k|V| \lg |V|)$ time. Figure 4.1 shows an example, in which we have nine adjacent lists and 3 hash functions. We use a line to separate the two groups in Figure 4.1. In the mining step to be described in section 4.1.2 we will use the first group as an example.

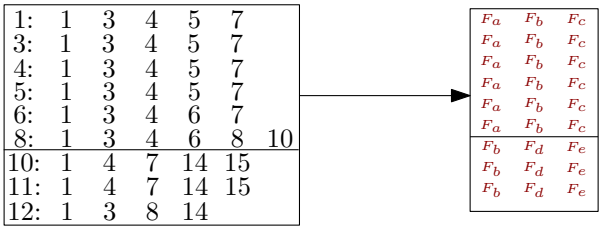


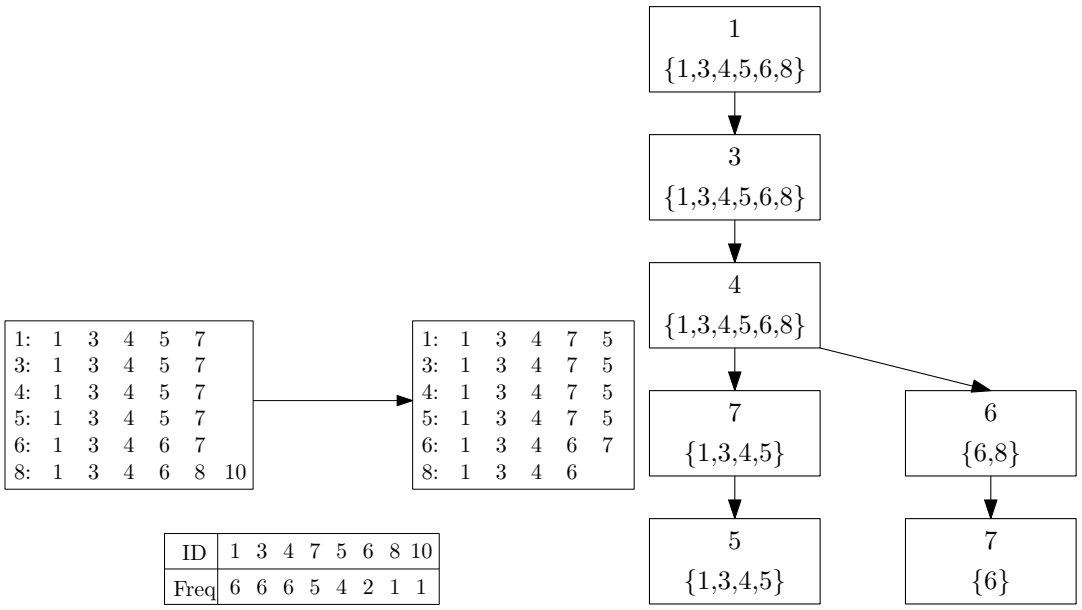
Figure 4.1: Clustering: Hashing Example

4.1.2 Pattern Mining

In this stage, the algorithm computes, for each group, the number of times each vertex appears as items in the adjacency lists in the group. This calculated value for each vertex is called the *frequency* of this vertex in the group. We then modify each adjacency list in the group as follows. First, we remove all the items whose frequencies are 1. Then we sort the items in this adjacency list in decreasing order of their frequencies. This step uses $O(|E| \lg |E|)$ time. Figure 4.2 (a) shows an example. The two tables in the top show the content of the adjacency lists of the first group in Figure 4.1 before and after this step, and the table in the bottom shows the frequencies of each vertex in the group.

The next step of the mining phase is to add all the sorted adjacency lists in each group to a prefix trie. Each node, x, of the trie is associated with a set of vertices of the given graph, such that the adjacency list of each vertex in the set is prefixed with the labels of the path from the root to x. Figure 4.2 (b) shows the prefix trie built for the sorted adjacency lists shown in Figure 4.2 (a). The *edge saving* of each node, x, of the trie is defined as $depth(x) \cdot set(x)$, where $depth(x)$ is defined as the depth of x in the trie and $set(x)$ is the size of the vertex set associated with x. For example, the edge saving of the leftmost leaf in the trie in Figure 4.2 (b) is 20, as its depth is 5 and the size of its associated vertex set is 4.

Note that Each node x in the prefix trie has a label which is the node id, and it represents the sequence, $list(x)$, of labels from the root to x. Node x corresponds to



(a) Re-order clusters (b) Prefix tree for cluster

Figure 4.2: Mining Phase

a dense subgraph $H(S, D)$, where S is the set of graph vertices associated with x, and D is the set of graph vertices in $list(x)$. For instance, the leftmost leaf in the trie in Figure 4.2 (b) corresponds to dense subgraph $H(S, D)$ where S is $\{1, 3, 4, 5\}$ and D is $\{1, 3, 4, 7, 5\}$. The *edge saving* value of $H(S, D)$ is defined to be $|S| \times |D|$, which is the number of edges in $H(S, D)$. The reason why this value is called edge saving is that we need not encode each of the $|S| \times |D|$ edges between S and D after we identify S and D, thus saving the cost of storing these edges explicitly.

To identify dense subgraphs, we choose two parameters: an integer value *threshold*, and another parameters *edgeSaving*, which is a sequence of integers $< e_1, e_2, ..., e_t >$, where $e_1 > e_2 > \cdots > e_t$. We then construct the prefix tree described in this section. We perform a traversal of the tree. When we visit a node x, if its corresponding dense subgraph has more than e_1 edges, we add this dense subgraph into a vector *vec*. Each time after we add a dense subgraph into *vec*, we search *vec* to find out where there is a dense subgraph in *vec* that shares an edge which the newly added dense subgraph.

If there is, we remove the dense subgraph whose edge saving value is smaller between these two dense subgraphs. Once we visit all the nodes in the prefix tree, we remove the edges of the dense subgraphs in *vec* from the original graph and repeat this process. When the total number of dense subgraphs added to *vec* minus the number of subgraphs removed from *vec* is less than *threshold* in a single iteration, we repeat the same process for $e_2, e_3, \cdots, e_t$. After we use up all the values in edgeSaving, *vec* contains all the dense subgraphs discovered.

4.2 Representation of Dense Subgraphs and Remaining Graphs

The mining algorithm divides the original graph into two parts: dense subgraphs and the remaining graph. The remaining graph is represented by a k^2-tree or MPk. Let $H_{all} = \{H(S_1, D_1), \cdots, H(S_d, D_d)\}$ denote the set of extracted dense subgraphs, where $H(S_i, D_i)$ is the ith dense subgraphs extracted by the mining algorithm. Two sequences represent H_{all}. The first is a sequence of vertex identifiers $X = X_1 X_2 \ldots X_d$, and the second is a bit vector $B = B_1 B_2 \ldots B_d$. X_i and B_i are substrings defined for $H(S_i, D_i)$ as follows: X_i is the concatenation of three components, L, M and R. L contains the elements in $S_i - D_i$, M stores $S_i \cap D_i$, and R stores $D_i - S_i$. The bitmap $B_i = 10^{|L|}10^{|M|}10^{|R|}$ has the information that can be used to divide X_i into L, M and R. Figure 4.3 shows how to represent a subgraph that containing only one extracted dense subgraph using the sequences X and B.

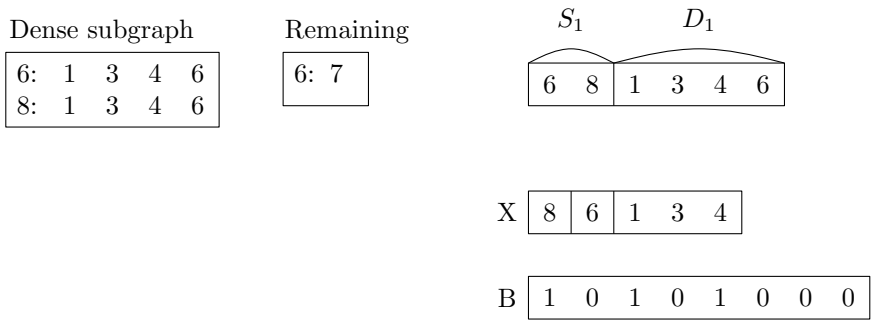


Figure 4.3: Compact Representation

4.3 Out/In-Neighbours Query

To support queries, X and B are represented by succinct data structures supporting `access`, `rank` and `select`. To list the out-neighbours of a vertex x, we first report its out-neighbours in the remaining graph using the k^2-tree for web graphs or MPk for social networks, and then report its out-neighbours in the dense subgraphs as follows: Use `select` to locate all the occurrences of x in X. For each occurrence, we perform a `select` operation to find its corresponding 0 bit in B. We can use the number of 1s, o, before this 0 to further tell whether x is in an L, M or R component by calculating o mod 3. If the result is 0, then the symbol is located in R. If the result is 1, then the symbol is located in L. If the result is 2, then the symbol is located in M. If it is in an L or M component, we perform `rank` and `select` operations over B and X to identify the starting and ending positions of the related M and R components in X, and report all the vertices in M and R. If it is in an R component, then report nothing. Similar ideas can be used to support in-neighbour and mining queries.

The same paper presents another approach which replaces each extracted dense subgraph by a virtual node to mine dense subgraphs recursively. It yields a structure which can outperform the approach described in previous paragraphs in terms of compression and the support for in-/out-neighbour queries. However, it does not support mining queries, and we do not try to improve it. We refer to it as `DSM`.

Chapter 5

Experimental Studies

We have performed experimented studies on compressed representations of web graphs and social networks based on dense subgraphs extraction. In Section 5.1, we use Hernández and Navarro's algorithm for dense subgraphs extraction [22], and repeat their experimental studies to test how various parameters of the algorithm affects compression results. We then report the compression ratios and query times of Hernández and Navarro's compressed representation by using it to represent a set of web graphs and social networks. In Section 5.2, we further study the encoding of dense subgraphs, and in particular the encoding of the sequence X defined in Section 4.2, using various encoding schemes of wavelet trees. Most of these encoding schemes were not studied in [22]. Finally, in Section 5.3, based on the properties of the wavelet trees constructed over X, we propose a new approach called *combined encoding* for the wavelet tree encoding of X to improve previous results in experimental studies.

Since we use different wavelet tree shapes in our experiments, we define the following list of abbreviations:

- `WT`: pointer-based normal-shaped wavelet trees
- `WTNP`: pointerless normal-shaped wavelet trees
- `HWT`: pointer-based Huffman-shaped wavelet trees
- `CHWTNP`: pointerless Canonical Huffman-shaped wavelet trees

We may further combine these abbreviations to specify complete encoding schemes.

For instance, `WTNP-RRR15` stands for a pointerless normal-shaped wavelet tree with its bit vectors at each level encoded by `RRR15`.

The source code of some data structures was made available and used in our studies, including the code for dense subgraphs extraction and `DSM` provided by Hernández and Navarro [22], `WTNP`, `RRR15`, Huffman encoding and `CHWTNP` implementation from the compact structures library libcds (`http://recoded.cl/`), the code for `WT`, `HWT`, `RLEG` and `RLED` provided by Grossi et al [21], the latest k^2-tree representation [7], and the MPk representation from `http://webgraphs.recoded.cl/index.php`. We implemented all the other data structures, including `RRR31`, `RRR63`, `Plain` and `RLEO`. For `RRR`, sr is set to 64 (see section 3.1.3 for the definition of sr), and in `Plain`, the size of a basic block is set to 1024 and the size of a select block is set to 8192. These parameter values are the recommended values given by their authors. The code was written in C++ and compiled by gcc with the -O3 flag turned on for optimization. We used a PC with 16GB RAM installed with 64-bit Ubuntu Linux 14.04 LTS, whose processor is Intel Core i5-3550 at 3.30 GHz with 6MB L3 cache.

The data used in our experimental studies are from the website of the WebGraph Framework project (`http://webgraph.di.unimi.it/`), and the natural order of these graphs are used, i.e., the vertices are sorted by the corresponding URL (webgraphs) or member name (social networks). The only exception is the file soc-Livejournal, which is available from the SNAP project (`http://snap.stanford.edu/data/`). Table 5.1 lists the data sets that we use in our experimental studies, in which the first five are web graphs, and the last five are social networks.

Table 5.1: Data Sets used in our experimental studies and the effectiveness of dense subgraphs mining algorithms

data set	vertices	edges	$\|H\|/\|E\|$	description
eu-2005	862,664	19,235,140	89.47%	a small crawl of the websites with .eu domains
in-2004	1,382,908	16,917,053	86.01%	a small crawl of the websites with .in domains
indochina-2004	7,414,866	194,109,311	93.31%	a fairly large crawl of the country domains of Indochina
uk-2002	18,520,486	298,113,762	89.62%	a large crawl of the websites with .uk domains in 2002
arabic-2005	22,744,080	639,999,458	93.30%	a large crawl of the websites that contain pages written in arabic
enron	69,244	276,143	44.24%	a partially anonymised corpus of e-mail messages exchanged by some Enron employees
dblp-2011	986,324	6,707,236	65.52%	a bibliography service from an undirected scientific collaboration network
dewiki-2013	1,532,354	36,722,696	60.03%	a snapshot of the German part of Wikipedia as of late February 2013
soc-LiveJournal	4,847,571	68,993,773	55.96%	a free on-line community with almost 10 million members
ljournal-2008	5,363,260	79,023,142	56.34%	a virtual-community social site started in 1999

5.1 Dense Subgraphs Mining Algorithm

Recall that the algorithm of Hernández and Navarro's [22] uses the following three parameters: *edgeSaving*, *hash factor*, and *threshold*, which are described in section 4.1 of this thesis.

The smaller the value of *threshold* is, the more thoroughly the mining algorithm discovers dense subgraphs that are sufficiently large before using the next smaller value in the *edgeSaving* sequence. Therefore, having a small *threshold* value helps to extract more dense subgraphs. However, setting the threshold to a smaller value will also increase the running time of the mining algorithm. Therefore, Hernández and Navarro [22] used larger threshold values for large graphs to avoid using too much time for mining, while using smaller threshold values for small graphs to discover large dense subgraphs more thoroughly. We follow the same strategy and use the same values for threshold as in their work. More precisely, $threshold = 10$ for eu-2005, in-2004, dblp-2011 and enron, $threshold = 100$ for dewiki-2013, uk-2002, indochina-2004, ljournal-2008, and soc-livejournal, and $threshold = 500$ for arabic-2005.

Table 5.1 also show how effectively the dense subgraphs mining algorithm discovers dense subgraphs by setting $edgeSaving = <500, 100, 50, 30, 15, 6>$. The fourth columns of this table gives the ratio of the number of edges of the dense subgraphs

Table 5.2: Dense subgraph extraction using different k with in-2004

hash factor	dense subgraphs	nodes	edges	edge-vertex ratio
2	97,489	1,783,053	14,539,382	8.15
4	101,819	1,815,298	14,538,813	8.01
8	104,207	1,863,324	14,544,791	7.81

to the number of edges in the entire graph ($|H|$ denotes the number of edges in the dense subgraphs). As shown in the table, for each web graph, $89\% - 94\%$ of the edges of the original graph are included in the extracted dense subgraphs, while for social networks, only $41\% - 65\%$ of the edges are extracted. This means that Hernndez and Navarro's algorithm [22] can potentially achieve more compression for web graphs than for social networks. It also shows a well-known fact that the web graphs are more compressible compared to the social networks.

Next, Hernández and Navarro [22] discovered that setting the hash factor to be 2 is the best. This is because the algorithm discovers smaller dense subgraphs when the *hash factor* increases, which is bad for compression. We run the same experiment for one of the graphs to demonstrate this. We fix *edgeSaving* to be $<500, 100, 50, 30, 15, 6>$ and *threshold* to be 10, and run the algorithm on in-2004. Table 5.2 show the number of dense subgraphs, the total number of nodes in these dense subgraphs, the total number of edges in these subgraphs and the ratio of the number of edges to the number of nodes. It turns out that setting the hash factor to be 2 is better than other values in terms of edge-vertex ratio.

We then run the compression algorithm on all graphs to see how the values of *edgeSaving* affect compression performance. We use `WTNP-RRR15` to compress X and `RRR15` to compress B. We use k^2-tree and MPk for web graphs and social networks to encode the remaining graph, respectively. Tables 5.3 and 5.4 present the results, whose columns correspond to different sequences of values for edgeSaving: To get the

Table 5.3: Compression performance on web graphs for different values of *edgeSaving*

eu-2005						
edgeSaving	500	100	50	30	15	6
$\|H\|/\|E\|$	59.74%	77.23%	81.77%	84.24%	87.29%	89.47%
dense subgraphs	1,386,736	2,396,356	2,862,572	3,207,528	3,732,808	4,262,672
remaining graph	5,312,512	3,883,008	3,268,608	2,895,872	2,420,736	2,035,712
entire Graph	2.79	2.61	2.55	2.54	2.56	2.62
query	3.68	4.67	5.07	5.32	5.60	6.01

in-2004						
edgeSaving	500	100	50	30	15	6
$\|H\|/\|E\|$	64.1%	75.97%	79.12%	81.22%	83.97%	86.01%
dense subgraphs	1,123,096	1,789,160	2,057,080	2,309,084	2,675,912	3,078,556
remaining graph	4,222,976	3,444,736	3,112,960	2,846,720	2,469,888	2,207,744
entire Graph	2.53	2.48	2.44	2.44	2.43	2.50
query	4.06	4.72	4.95	5.12	5.47	5.69

indochina-2004						
edgeSaving	500	100	50	30	15	6
$\|H\|/\|E\|$	81.52%	86.90%	89.36%	90.74%	92.2%	93.31%
dense subgraphs	9,811,748	13,841,488	16,443,616	18,402,072	20,922,280	23,600,612
remaining graph	25,344,000	20,135,936	17,448,960	15,503,360	13,217,792	11,341,824
entire Graph	1.45	1.40	1.40	1.40	1.41	1.44
query	4.85	5.21	5.42	5.56	5.69	5.80

uk-2002						
edgeSaving	500	100	50	30	15	6
$\|H\|/\|E\|$	69.75%	79.76%	83.19%	85.30%	87.71%	89.62%
dense subgraphs	22,200,160	33,637,076	39,944,032	44,867,204	52,322,728	60,006,512
remaining Graph	72,048,640	57,315,328	50,106,368	44,933,120	37,978,112	34,603,008
entire Graph	2.53	2.44	2.42	2.41	2.42	2.54
query	10.81	11.30	11.34	11.40	11.41	11.70

arabic-2005						
edgeSaving	500	100	50	30	15	6
$\|H\|/\|E\|$	79.89%	86.38%	88.74%	90.20%	91.90%	93.30%
dense subgraphs	40,658,064	59,827,716	71,636,072	79,573,748	91,149,776	104,350,720
remaining Graph	110,178,304	84,508,672	72,704,000	64,512,000	53,899,264	40,738,816
entire Graph	1.89	1.80	1.80	1.80	1.81	1.81
query	6.55	7.11	7.46	7.60	7.70	8.27

Table 5.4: Compression performance on social networks for different values of *edgeSaving*

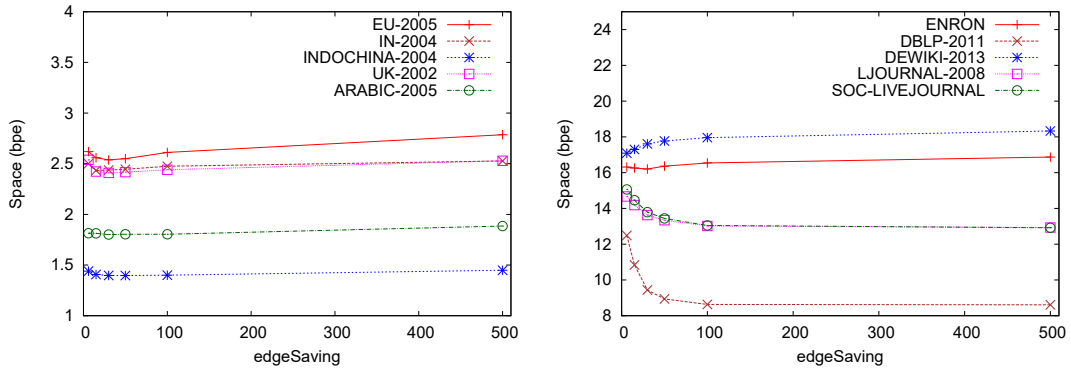
enron						
edgeSaving	500	100	50	30	15	6
$\|H\|/\|E\|$	4.77%	13.82%	20.99%	25.74%	29.14%	44.24%
dense subgraphs	9,684	31,052	50,504	61,552	74,468	125,924
remaining Graph	572,796	539,972	514,304	497,876	486,796	437268
entire Graph	16.87	16.54	16.36	16.21	16.26	16.32
query	5.10	5.61	5.68	5.78	5.98	7.01

dblp-2011						
edgeSaving	500	100	50	30	15	6
$\|H\|/\|E\|$	2.59%	9.48%	18.52%	28.71%	48.52%	65.52%
dense subgraphs	53,792	236,460	683,048	1,328,640	3,020,004	5,073,188
remaining Graph	7,162,688	6,999,844	6,808,312	6,583,820	6,066,496	5,401,864
entire Graph	8.61	8.63	8.94	9.44	10.84	12.49
query	4.34	4.81	5.31	5.91	7.75	9.77

dewiki-2013						
edgeSaving	500	100	50	30	15	6
$\|H\|/\|E\|$	2.38%	14.75%	23.84%	31.88%	45.76%	60.03%
dense subgraphs	614,844	6,302,064	11,143,876	15,621,136	24,129,072	34,491,136
remaining Graph	83,524,912	76,130,700	70,412,816	65,233,072	55,304,780	43,951,620
entire Graph	18.33	17.96	17.77	17.61	17.30	17.09
query	9.72	10.19	10.69	11.23	12.50	13.88

soc-livejornal						
edgeSaving	500	100	50	30	15	6
$\|H\|/\|E\|$	4.65%	10.80%	19.57%	27.99%	42.08%	55.96%
dense subgraphs	551,608	4,316,268	11,456,108	18,999,288	34,105,256	51,947,268
remaining Graph	110,914,812	108,207,824	104,412,244	99,914,556	90,631,420	77,911,280
entire Graph	12.92	13.05	13.44	13.79	14.46	15.06
query	7.73	8.77	10.08	10.35	12.55	14.28

ljournal-2008						
edgeSaving	500	100	50	30	15	6
$\|H\|/\|E\|$	6.07%	12.97%	21.47%	29.54%	42.80%	56.34%
dense subgraphs	771,620	5,619,300	13,563,840	21,750,104	37,496,304	56,508,108
remaining Graph	126,975,036	123,024,380	118,238,164	112,907,628	102,621,808	88,409,780
entire Graph	12.93	13.02	13.34	13.63	14.19	14.67
query	7.05	8.99	9.93	10.62	12.06	13.46



(a) web graphs (b) social networks

Figure 5.1: Space cost for different edgeSaving values.

results for columns labeled 500, 100, 50, 30, 15, and 6, we set the values of edgeSaving to <500>, <500, 100>, <500, 100, 50>, <500, 100, 50, 30>, <500, 100, 50, 30, 15>, and <500, 100, 50, 30, 15, 6>, respectively. Hence the label of each column is the last value in the sequence of values assigned to *edgeSaving* for that column. The second row of this table presents the total number of edges in the dense subgraphs divided by the total number of edges in the original graph. The third and fourth row of this table present the number of bytes needed to encode dense subgraphs and the remaining graph, respectively. The fifth row presents the space cost of encoding the entire graph expressed in bits per edge (bpe), i.e., the total space in bits divided by the number of edges in the graph. We used a randomly generated query sequence containing 8×10^6 queries, and the query times are in microseconds per out-neighbour (out-going edge) reported. Here we only show the time of out-neighbor queries since the in-neighbor queries have similar performance. Figure 5.1 illustrates the compression performance recorded in Tables 5.3- 5.4 on web graphs and social networks.

Decreasing the last value in *edgeSaving* may help us discover more dense subgraphs, which may potentially achieve more compression. On the other hand, this may also discover many dense subgraphs that are too small, which may affect the

Table 5.5: Compression and query performance for web graphs compared to k^2-tree.

			Hernández and Navarro [22]		k^2-tree	
data set	vertices	edges	bpe	time	bpe	time
eu-2005	862,664	19,235,140	2.54	5.32	3.46	1.13
in-2004	1,382,908	16,917,053	2.43	5.47	2.65	1.36
indochina-2004	7,414,866	194,109,311	1.40	5.42	1.72	1.76
uk-2002	18,520,486	298,113,762	2.41	11.40	2.82	7.57
arabic-2005	22,744,080	639,999,458	1.80	7.60	2.45	2.80

Table 5.6: Compression and query performance for social graphs compared to MPk.

			Hernández and Navarro [22]		MPk	
data set	vertices	edges	bpe	time	bpe	time
enron	69,244	276,143	16.21	5.78	17.15	5.15
dblp-2011	986,324	6,707,236	8.61	4.34	8.62	5.46
dewiki-2013	1,532,354	36,722,696	17.09	13.88	18.48	9.99
soc-livejournal	4,847,571	68,993,773	12.92	7.73	13.25	8.63
ljournal-2008	5,363,260	79,023,142	12.93	7.05	13.42	8.62

compression performance negatively. Figure 5.1 shows that for each graph, there exists an *edgeSaving* value that maximizes compression and we use this value in our mining algorithm, as is done by Hernández and Navarro [22]. More specifically we fix *edgeSaving* to be $<500, 100, 50, 30, 15, 6>$ for dewiki-2013, *edgeSaving* to be $<500, 100, 50, 30, 15>$ for in-2004, *edgeSaving* to be $<500, 100, 50, 30>$ for eu-2005, uk-2002, arabic-2005 and enron, *edgeSaving* to be $<500, 100, 50>$ for indochina-2004, and *edgeSaving* to be $<500>$ for dblp-2011, ljournal-2008 and soc-livejournal1.

Tables 5.3 and 5.4 already include the best compression performance that can be achieved by Hernández and Navarro's approach [22]. To show how much more compression has been achieved compared to the original k^2-tree and MPk, we report, in Table 5.5 and Table 5.6, the compression and query performance that can be achieved with the *edgeSaving* values that maximize compression, as well as the performance achieved by directly using k^2-tree and MPk to encode web graphs and social networks, respectively.

Table 5.5 shows that Hernández and Navarro's approach achieves more compression for web graphs than k^2-tree does. Similarity, Table 5.6 shows that their approach also achieves more compression for social networks than MPk does. The difference in compression performance is larger for web graphs, and this is because the ratio of the number of edges in dense subgraphs to the number of edges in the original graph is much higher for web graphs than that for social networks, as shown in Table 5.3 and Table 5.4. The improvement in compression performance is achieved at the cost of sacrificing query time for web graphs, though the query performance achieved by Hernández and Navarro's approach is still competitive.

5.2 Wavelet Tree Implementations for Dense Subgraphs

We now perform experimented studies on wavelet tree representations of the sequence X and bit vectors representations of B defined in Section 4.2. Because the encoding of the remaining graph of a given graph is chosen independantly, we only perform experimental studies on the representations of dense subgraphs in this section.

Table 5.7 shows how big these sequences are. We include $|X|$ which is defined as the length of X, σ which is defined as the alphabet size of X, $\lg \sigma$, $H_0(X)$, $\lceil |X| \times \lg \sigma \rceil$ which is the information theoretical lower bound of encoding X, and $|B|$ which is

Table 5.7: Components of dense subgraph representations

data set	$\|X\|$	σ	$\lceil \|X\| \times \lg \sigma \rceil$	$\lg \sigma$	$H_0(X)$	$\|B\|$
eu-2005	1,808,357	862,611	35,657,817	19.72	19.02	1,949,427
in-2004	1,570,515	1,382,895	32,037,344	20.40	19.43	1,731,463
indochina-2004	9,048,551	7,414,862	206,505,926	22.82	21.79	9,507,426
uk-2002	23,716,402	18,520,483	572,576,043	24.14	23.23	25,349,495
arabic-2005	40,259,449	22,744,079	983,900,178	24.44	23.39	42,616,019
enron	27,507	69,236	442,292	16.08	12.96	28,993
dblp-2011	6,510	986,309	129,625	19.91	12.13	7,015
dewiki-2013	12,240,298	1,532,352	251,505,275	20.55	19.55	15,310,176
soc-livejornal	210,340	4,846,369	4,671,330	22.21	15.68	216,016
ljournal-2008	334,163	5,363,089	7,470,091	22.35	16.08	342,825

Table 5.8: The ratio of the space cost of encoding X to the space cost of encoding B

data set	X	B	ratio
eu-2005	24,280,096	130,336	186.29
in-2004	19,746,272	124,648	158.42
indochina-2004	123,920,608	552,208	224.41
uk-2002	336,609,888	1,639,596	205.30
arabic-2005	605,684,288	2,610,920	231.98
enron	421,504	1,756	240.04
dblp-2011	105,920	576	183.89
dewiki-2013	262,155,840	1,599,020	163.95
soc-livejornal	2,661,280	10,668	249.46
ljournal-2008	4,150,752	16,724	248.19

defined as the length of B. The fact that $H_0(X)$ is close to $\lg \sigma$ indicates that X is difficult to compress.

Table 5.8 shows the space cost in bits of X encoded by `WTNP-RRR15` and the space cost in bits of B encoded by `RRR15`. We include the ratio of the space cost of encoding X to the space cost of encoding B for different graphs. We found that the space cost of B is less than 1% of that of X, so we perform experimental studies on the encoding of X only in this section. Let T_X be the wavelet tree representing X. When the context is clear, we say that T_X is encoded by a certain bit vector implementation (e.g., `RRR15`) when the bit vector at each of its level is encoded by this same representation. In this section, compression is measured in bits per symbol which is the space cost of storing X in bits divided by the length of X. These values, when compared to the values of $\lg \sigma$ and $H_0(X)$ given in Table 5.7, tell us how much compression has been achieved. The queries tested were out-neighbour queries performed over dense subgraphs only. We used a randomly generated query sequence containing 8×10^6 queries. The time is measured in μs per edge which is the total time cost of all queries divided by the total number of edges retrieved by these queries.

5.2.1 Pointer-Based Wavelet Trees

Now we use pointer-based wavelet trees to encode X. The bit vector data structures we use to encode a level in a wavelet tree include `Plain`, `RRR15`, `RRR63`, `RLEG256`, `RLED256`, and `RLEO256`. The Wavelet tree shapes that we use are `WT` and `HWT`.

Table 5.9 shows the compression performance of pointer-based normal-shaped wavelet trees combined with different compact bit vectors. We include the cost of X encoded by the original method `WTNP-RRR15` to compare with pointer-based normal-shaped wavelet trees, even though `WTNP-RRR15` is a pointerless wavelet tree. Among the pointer-based wavelet trees tested here, the best compression is achieved by `WT-RLEG256`. `WT-RLED256` and `WT-RLEO256` always use slightly more space than `WT-RLEG256`. `WT-RRR15` and `WT-RRR63` use more space than `WT-RLEG256`, `WT-RLED256` and `WT-RLEO256`. `WT-RRR63` uses slightly more space than `WT-RRR15`. The reason is the bit vectors corresponding to the nodes at lower levels in pointer-based wavelet trees have small lengths and are not compressible, so we actually use more space for Table O at lower levels when we set the length of a block to 63. `WT-Plain` is the worst in space efficiency. Comparing to the original `WTNP-RRR15`, `WT` uses 4 to 30 times the space. The reason is that the alphabet size of X is large, and thus the corresponding wavelet tree structure has many leaves. Therefore, there are many nodes in the wavelet tree, and many pointers have to be stored to encode the tree structure in a pointer-based representation. These pointers use too much space.

Table 5.10 shows the query performance of pointer-based normal-shaped wavelet trees combined with different compact bit vectors. We also include the query performance of the original method `WTNP-RRR15` to compare with all pointer-based normal-shaped wavelet trees. The best query performance is achieved by `WT-Plain`. `WTNP-RRR15` is 1 to 3 times slower than `WT-RRR15` only. The reason why `WT-RRR15` is faster is that we use more space to store the pointers in order to speed up the traversal in

Table 5.9: Compression performance for dense subgraphs using pointer-based normal-shaped wavelet trees, measured in bits per symbol

data set	WT-Plain	WT-RRR15	WT-RRR63	WT-RLEG256	WT-RLED256	WT-RLEO256	WTNP-RRR15
eu-2005	559.97	448.48	450.07	266.49	267.55	267.04	13.43
in-2004	778.02	623.84	624.88	368.90	369.85	369.34	12.57
indochina-2004	745.85	596.63	597.75	352.39	353.51	352.84	13.70
uk-2002	761.93	609.05	609.48	359.56	360.55	360.03	14.19
arabic-2005	607.40	484.36	484.63	286.61	287.75	287.13	15.04
enron	613.68	498.45	501.28	299.67	301.27	300.63	15.32
dblp-2011	1050.84	852.50	854.57	507.93	509.91	508.84	16.27
dewiki-2013	167.34	138.64	139.44	91.84	94.61	93.08	21.42
soc-livejornal	575.53	463.13	465.53	275.77	276.76	276.35	12.42
ljournal-2008	504.37	405.08	406.89	241.34	242.33	241.88	12.65

Table 5.10: Query performance for dense subgraphs using pointer-based normal-shaped wavelet trees, measured in microseconds per edge

data set	WT-Plain	WT-RRR15	WT-RRR63	WT-RLEG256	WT-RLED256	WT-RLEO256	WTNP-RRR15
eu-2005	2.06	3.59	7.54	36.94	36.98	37.42	5.25
in-2004	1.78	3.24	6.14	30.43	30.95	31.52	5.26
indochina-2004	1.80	3.49	6.10	33.60	34.41	34.64	4.82
uk-2002	2.53	4.46	7.76	52.56	53.16	54.02	6.36
arabic-2005	2.42	4.46	7.76	51.90	52.21	53.39	6.09
enron	1.54	2.48	9.14	24.10	25.53	25.31	5.74
dblp-2011	0.57	0.98	5.11	10.05	10.62	10.56	2.81
dewiki-2013	6.01	9.72	23.33	76.24	80.99	75.97	14.84
soc-livejornal	1.02	2.17	5.37	25.44	26.30	26.70	3.56
ljournal-2008	1.17	2.34	4.91	28.01	28.93	29.16	3.66

the tree, and WTNP-RRR15 requires additional computation to determine, for a node v, the starting and ending positions of $B(v)$ in the concatenated bit vector for v's level.

Table 5.11 shows the compression performance of pointer-based Huffman-shaped wavelet trees combined with different compact bit vectors. The best compression performance is achieved by HWT-RLEG256 among all pointer-based Huffman-shaped wavelet trees. However, the space cost of HWT-RLEG256 is much larger than WT-RLEG256. The reason why HWT-RLEG256 uses more space is that HWT stores the Huffman coding table along with the tree structure. Thus it takes more space for larger alphabets.

Table 5.12 shows the query performance of pointer-based Huffman-shaped wavelet trees combined with different compact bit vectors. The best query performance is

Table 5.11: Compression performance of dense subgraphs using Huffman-shaped wavelet trees, measured in bits per symbol

data set	HWT-Plain	HWT-RRR15	HWT-RRR63	HWT-RLEG256	HWT-RLED256	HWT-RLEO256	WTNP-RRR15
eu-2005	675.80	566.02	566.84	384.37	385.63	384.96	13.43
in-2004	940.79	788.16	788.45	533.20	534.56	533.67	12.57
indochina-2004	900.79	753.19	753.13	509.13	510.43	509.59	13.70
uk-2002	920.27	769.00	768.80	519.73	521.06	520.21	14.19
arabic-2005	732.07	610.83	610.56	413.55	414.84	414.10	15.04
enron	742.93	628.69	629.81	430.39	432.51	431.16	15.32
dblp-2011	1274.50	1076.27	1077.35	731.24	733.28	731.86	16.27
dewiki-2013	198.50	171.36	171.52	125.32	128.49	126.53	21.42
soc-livejornal	695.21	585.03	586.76	398.62	400.02	399.28	12.65
ljournal-2008	608.56	511.75	513.23	349.17	350.57	349.82	12.42

Table 5.12: Query performance of dense subgraphs using Huffman-shaped wavelet trees, measured in microseconds per edge

data set	HWT-Plain	HWT-RRR15	HWT-RRR63	HWT-RLEG256	HWT-RLED256	HWT-RLEO256	WTNP-RRR15
eu-2005	1.75	2.02	4.37	18.20	18.20	19.80	5.25
in-2004	1.60	1.87	3.59	14.18	14.91	16.22	5.26
indochina-2004	1.57	1.79	3.34	15.42	16.26	18.46	4.82
uk-2002	2.48	2.91	4.86	26.80	27.20	30.00	6.36
arabic-2005	2.14	2.58	4.44	23.36	24.29	26.74	6.09
enron	1.32	1.58	5.46	14.35	15.26	15.13	5.74
dblp-2011	0.34	0.34	2.58	4.25	5.05	4.94	2.81
dewiki-2013	4.99	6.89	14.67	47.99	50.29	48.38	14.84
soc-livejornal	0.83	1.14	3.09	10.41	10.96	12.24	3.56
ljournal-2008	0.95	1.34	3.24	11.67	12.27	13.75	3.66

achieved by HWT-Plain among all pointer-based Huffman-shaped wavelet trees.

Figure 5.2 and Figure 5.3 show the space/time tradeoffs of all the approaches discussed in this section. The best space performance is achieved by WTNP-RRR15 as shown in the previous tables. The best query performance is achieved by HWT-Plain. It is faster than WTNP-RRR15, but WTNP-RRR15 is only 1-3 times slower than WT-RRR15. Considering the huge space cost of pointer-based wavelet trees, WTNP-RRR15 is more preferable.

Overall, pointers-based wavelet trees are not good at compressing dense subgraphs because of the space cost required by large alphabets. They are not competitive in dense subgraphs compression compared to the original idea of using WTNP-RRR15. Thus we focus on pointerless wavelet trees. We also notice that bit vectors based on run-length encodings achieve good compression performance.

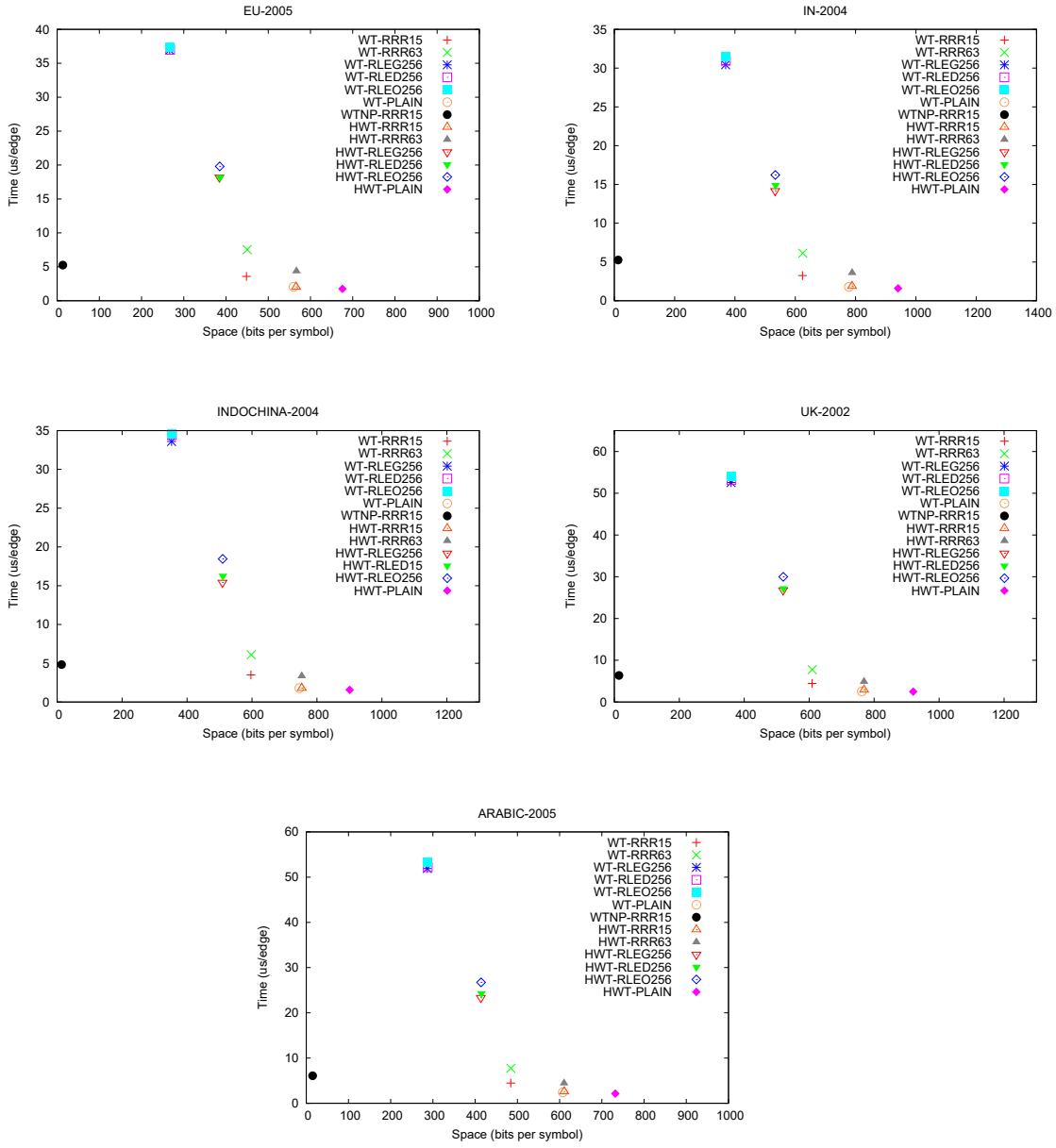


Figure 5.2: Space/time performance in web graphs

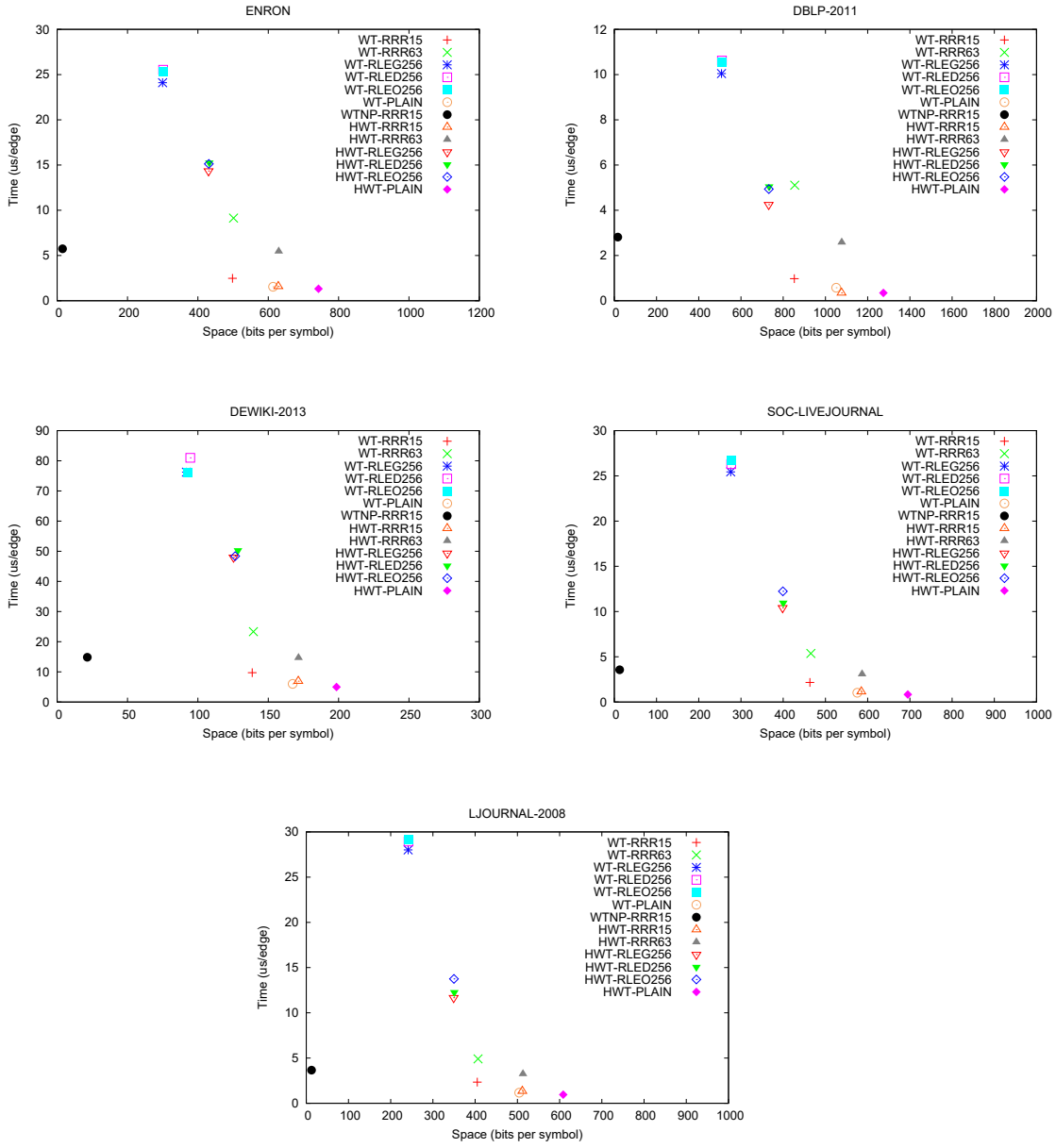


Figure 5.3: Space/time performance in social networks

Table 5.13: Compression performance of dense subgraphs using pointerless normal-shaped wavelet trees, measured in bits per symbol

data set	Plain	RRR15	RRR63	RLEG256	RLED256	RLEO256
eu-2005	21.41	13.43	12.14	10.12	11.05	10.67
in-2004	21.37	12.57	10.99	8.95	9.74	9.42
indochina-2004	24.47	13.70	11.55	9.03	9.83	9.51
uk-2002	25.55	14.19	11.80	9.23	10.05	9.71
arabic-2005	26.59	15.04	12.49	9.95	10.87	10.49
enron	15.86	15.32	15.25	15.31	16.99	16.20
dblp-2011	16.56	16.27	15.83	16.30	18.18	17.13
dewiki-2013	22.19	21.42	20.08	22.56	25.31	23.81
soc-livejornal	18.21	12.65	12.45	10.48	11.43	11.06
ljournal-2008	18.18	12.42	11.96	10.10	11.03	10.67

Table 5.14: Query performance of dense subgraphs using pointerless normal-shaped wavelet trees, measured in microseconds per edge

data set	Plain	RRR15	RRR63	RLEG256	RLED256	RLEO256
eu-2005	3.85	5.25	13.99	107.55	114.02	110.00
in-2004	3.85	5.26	13.09	105.97	113.17	111.31
indochina-2004	3.45	4.82	12.89	117.93	124.66	122.51
uk-2002	4.64	6.36	16.16	140.10	147.73	141.34
arabic-2005	4.38	6.09	15.55	135.43	142.17	135.12
enron	4.34	6.88	21.73	74.77	82.30	78.15
dblp-2011	1.95	4.38	19.51	35.82	41.45	39.03
dewiki-2013	11.08	14.84	39.53	135.25	145.98	135.37
soc-livejornal	2.52	3.56	10.77	81.94	87.37	84.09
ljournal-2008	2.51	7.21	20.39	81.92	87.24	83.92

5.2.2 Pointerless Wavelet Trees

In this section, we focus on two types of pointerless wavelet trees. One is the normal-shaped pointerless wavelet tree [15], and the other one is the canonical Huffman-shaped wavelet tree [32]. The bit vector data structures that we use include `RLEG256`, `RLED256`, `RLEO256`, `Plain`, `RRR15` and `RRR63`.

Table 5.13 and Table 5.14 show the compression performance and the query performance of pointerless normal-shaped wavelet trees combined with different compact bit vectors. The best compression performance is achieved by `RLEG256` for all web graphs. The best compression performance is achieved by `RRR63` for most social

Table 5.15: Compression performance of dense subgraphs using canonical Huffman-shaped wavelet trees, measured in bits per symbol

data set	Plain	RRR15	RRR63	RLEG256	RLED256	RLEO256
Eu-2005	40.95	33.99	33.00	31.26	32.24	31.83
In-2004	50.03	41.83	40.44	38.53	39.35	39.02
Indochina-2004	52.12	42.37	40.55	38.23	39.04	38.71
Uk-2002	55.42	44.80	42.67	40.23	41.09	40.73
Arabic-2005	48.98	38.75	36.53	34.34	35.31	34.89
Enron	35.07	35.40	34.90	36.22	38.16	37.05
Dblp-2011	49.91	49.72	49.38	50.04	51.90	50.78
Dewiki-2013	26.30	26.93	25.57	28.74	31.72	29.96
Soc-livejornal	37.07	33.00	33.02	31.66	32.74	32.31
Ljournal-2008	34.78	30.19	29.91	28.61	29.65	29.23

Table 5.16: Query performance of dense subgraphs using canonical Huffman-shaped wavelet trees, measured in microseconds per edge

data set	Plain	RRR15	RRR63	RLEG256	RLED256	RLEO256
Eu-2005	7.08	7.08	39.28	242.85	274.49	256.92
In-2004	7.65	7.08	39.01	278.31	312.84	297.79
Indochina-2004	7.12	6.64	35.83	278.82	310.28	300.12
Uk-2002	8.87	9.08	47.39	337.80	375.20	358.10
Arabic-2005	8.72	8.64	46.18	302.50	337.12	321.90
Enron	7.40	6.90	52.54	185.79	220.98	188.30
Dblp-2011	4.59	4.59	32.15	103.56	111.25	104.02
Dewiki-2013	12.21	14.17	78.82	276.02	325.26	276.21
Soc-livejornal	5.22	5.13	32.66	190.20	219.81	199.91
Ljournal-2008	5.57	5.34	34.12	198.43	227.12	208.07

networks. `Plain` has the best query performance among all those compact data structures for dense subgraphs.

Table 5.15 and Table 5.16 show the compression performance and the query performance of canonical Huffman-shaped wavelet trees combined with different compact bit vectors. The best compression performance is achieved by `RLEG256` for all web graphs, and `RRR63` for most social networks. `Plain` and `RRR15` have similarity query performance among all those compact data structures for dense subgraphs.

Figure 5.4 and Figure 5.5 illustrate the results recorded in Tables 5.13- 5.16 on web graphs and social networks, respectively. Recall that `WTNP-RRR15` is the original

encoding scheme by Claude and Navarro [15]. The figures show that WTNP-RLEG256 has the best compression performance for web graphs, and WTNP-RRR63 and WTNP-RLEG256 have better compression than other combinations for social networks, but the query time is not competitive comparing to the WTNP-RRR15 structure or WTNP-Plain. WTNP-Plain support faster queries than WTNP-RRR15 does for most graphs, since RRR required more computation when answering queries.

The CHWTNP data structures are not competitive for both web graphs and social networks. Both figures show that they are almost twice larger than the WTNP data structures, since our data has a high entropy and CHWTNP requires us to store some information so that we can compute the Huffman encoding of each symbol, as explained in Section 3.2.5. The query support of RLE-based CHWTNP is slower than RLE-based WTNP, because the query algorithm of CHWTNP needs to decompress symbols. WTNP-RRR15 and WTNP-RRR63 provide the best space/time tradeoff here. WTNP-RRR63 is better than WTNP-RRR15 regarding compression rates, but queries are slower with WTNP-RRR63. In other words, when the block size in RRR are bigger, we have better compression but slower query support. In this experiment, our RLE-based structure's block size is 256, which means RLE-based structure contains many runs in one block. We can decrease the block size of RLE to speed up query performance.

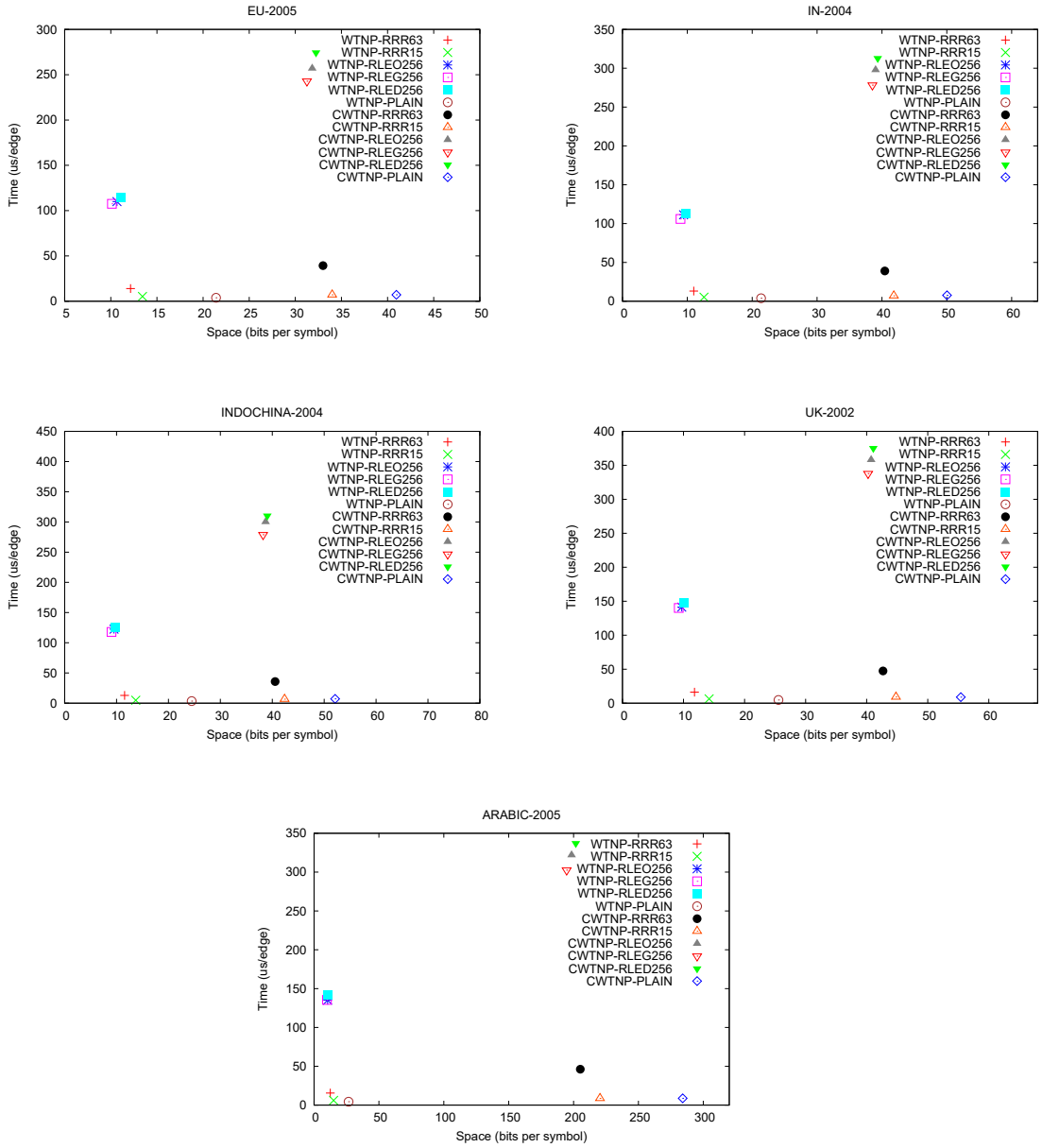


Figure 5.4: Space/time Performance in web graphs

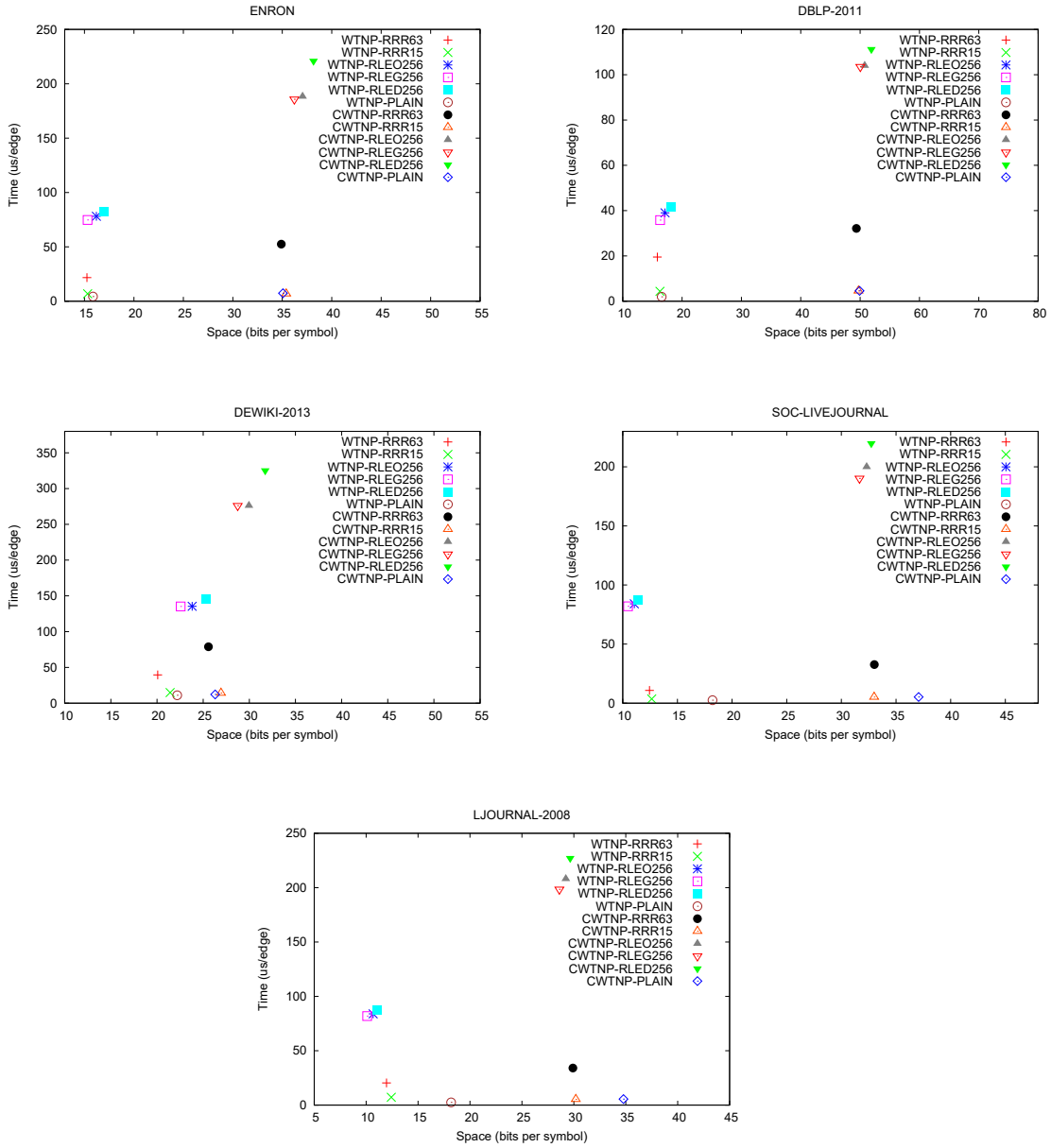


Figure 5.5: Space/time Performance in social networks

5.3 Improving Representation of Dense Subgraphs

5.3.1 Wavelet Tree Implementations for Dense Subgraphs

From the previous experiments, we noticed that the best implementations use pointerless normal-shaped wavelet trees. This is because the large alphabet of X prevents us from achieving improvements using alternative approaches such as those that use different tree shapes. Thus, we focus on investigating the encodings of bit vectors for different levels of wavelet trees, to see if we can make improvements. We only use `WTNP` in this section. Since `RLEG` has better compression and query performance than `RLED` and `RLEO`, we only consider `RLEG` in this section. Thus the bit vector implementations considered in this section are `RRR`, `Plain`, and `RLEG`.

Tables 5.17 and 5.18, with the exception of the last two columns in each table, present the compression ratio and query performance achieved by using different bit vector implementations in T_X. From these tables, we can see that the fastest query time is achieved by `Plain`, though its space cost is the highest. Hernández and Navarro's implementation is `RRR15`; increasing the block size in `RRR` can achieve more compression, though the queries will be slower. `RLE` with big block size can be used to achieve even more compression, but the query is even slower. This study shows that new tradeoffs can be obtained, though none of these new tradeoff is particularly

Table 5.17: Compression performance of dense subgraphs using wavelet trees, measured in bits per symbol

data set	Plain	RRR15	RRR63	RLEG32	RLEG64	RLEG256	RRR15-P	RL32-RRR15-P
eu-2005	21.41	13.43	12.14	17.12	13.12	10.12	13.05	10.93
in-2004	21.37	12.57	10.99	15.17	11.62	8.95	12.11	9.20
indochina-2004	24.47	13.70	11.55	15.29	11.71	9.03	13.27	9.31
uk-2002	25.55	14.19	11.80	15.58	11.95	9.23	13.75	9.57
arabic-2005	26.59	15.04	12.49	16.87	12.92	9.95	14.69	10.57
enron	15.86	15.32	15.25	26.65	20.18	15.31	14.78	14.56
dblp-2011	16.56	16.27	15.83	28.57	21.54	16.30	15.81	15.72
dewiki-2013	22.19	21.42	20.08	41.64	30.74	22.56	21.13	21.13
soc-livejournal	18.21	12.65	12.45	17.66	13.56	10.48	12.27	11.03
ljournal-2008	18.18	12.42	11.96	17.06	13.09	10.10	12.03	10.62

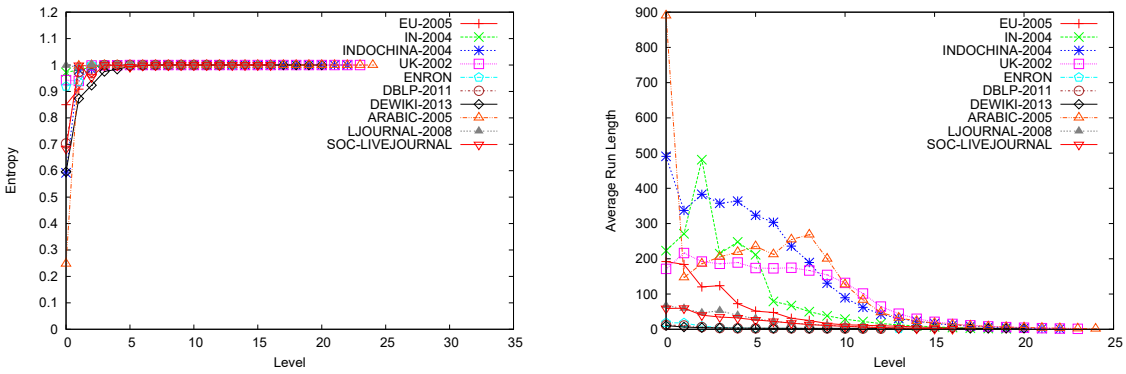
Table 5.18: Query performance of dense subgraphs, measured in microseconds per edge

data set	Plain	RRR15	RRR63	RLEG32	RLEG64	RLEG256	RRR15-P	RL32-RRR15-P
eu-2005	3.85	5.25	13.99	17.46	31.11	107.55	4.93	11.77
in-2004	3.85	5.26	13.09	15.46	26.82	105.97	4.94	11.82
indochina-2004	3.45	4.82	12.89	16.62	30.89	117.93	4.54	12.75
uk-2002	4.64	6.36	16.16	20.25	36.39	140.10	6.03	15.19
arabic-2005	4.38	6.09	15.55	20.32	37.12	135.43	5.81	14.60
enron	4.34	6.88	21.73	15.58	24.36	74.77	6.24	6.39
dblp-2011	1.95	4.38	19.51	11.91	18.56	35.82	3.36	3.47
dewiki-2013	11.08	14.84	39.53	34.09	46.66	135.25	12.76	13.47
soc-livejournal	2.52	3.56	10.77	10.13	18.75	81.94	3.18	6.65
ljournal-2008	2.51	7.21	20.39	18.70	29.93	81.92	6.59	9.51

attractive, as they sacrifice too much query or compression performance.

Thus, to achieve better performance, new approaches are needed. In our experimental studies, T_X has 20-25 levels (the number of levels is the ceiling of $\lg \sigma$, which is reported in Table 5.7). Figure 5.6 shows the entropy and average run length of the bit vector at each level. It shows that the bit vector at the root level has many long runs of 0s and 1s. As we go down the tree, the bit vector at each level has smaller average run length, while its zeroth-order empirical entropy becomes higher. In the last several levels, each bit vector has average run length between 1 and 10, and its zeroth-order empirical entropy is roughly 1. Thus the bit vectors at the bottommost levels are difficult to compress. From this observation, we conjectured that more compression can be achieved if we use `RLE` to represent bit vectors at top levels, and `Plain`, which does not compress bit vectors but has less space overhead, for bottom levels. `RRR` may still be the best option for the bit vectors in between. We then computed the space cost of different bit vector representations at different levels of T_X, which confirmed our conjecture. Figure 5.7 and Figure 5.8 show the space cost of different bit vectors at different levels for the web graphs and social networks, in which the levels are numbered $0, 1, 2, \ldots$ starting from the root level.

We thus use different bit vector implementations for different levels of T_X. As `Plain` has faster query time than `RRR15`, which has faster query time than `RLEG32`, we



(a) Entropy of the bit vector at each level

(b) Average run length of the bit vector at each level

Figure 5.6: Entropy and average run length of the bit vector at each level of the wavelet tree

develop two different ways of combining these implementations. The first approach uses `RRR15` and `Plain` only. It encodes the bit vectors in T_X from top down. We initially use `RRR15` to encode the bit vector at each level, until we reach a level for which `Plain` uses less space. Then, starting from this level downwards, we use `Plain` to encode bit vectors. This approach is referred to as `RRR15-P` in Tables 5.17 and 5.18, which show that its compression and query performance is $3\%-4\%$ and $5\%-6\%$ better than `RRR15` in web graphs, and its compression and query performance is $1\%-4\%$ and $8\%-24\%$ better than `RRR15` in social networks, respectively. The second approach uses three bit vector implementations in a similar fashion: from top to bottom, we use `RLEG32`, `RRR15` and `Plain` to maximize compression. This approach is referred to as `RL32-RRR15-P` in Tables 5.17 and 5.18, which show that its space cost is $19\%-33\%$ less than that of `RRR15` in web graphs and its space cost is $1\%-14\%$ less than that of `RRR15` in social networks. The query time is still competitive. This compression can not be achieved by increasing the block size of `RRR`, and to achieve roughly the same compression, we can use `RLEG256`, but then a query will use 10 times as much time as `RL32-RRR15-P`. Thus `RL32-RRR15-P` is an attractive method when more compression

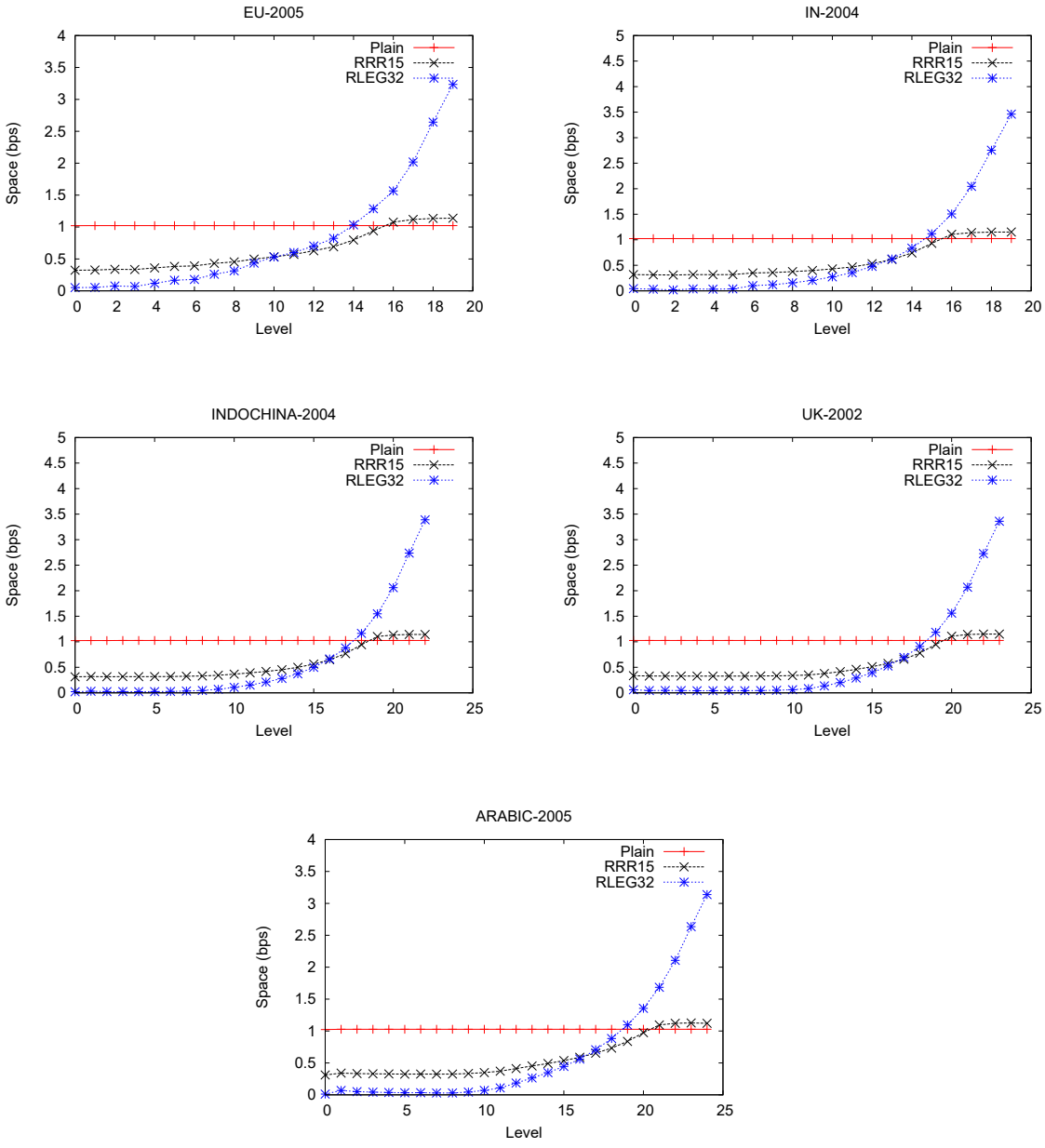


Figure 5.7: The compression performance of the bit vector at each level of the wavelet tree in web graphs

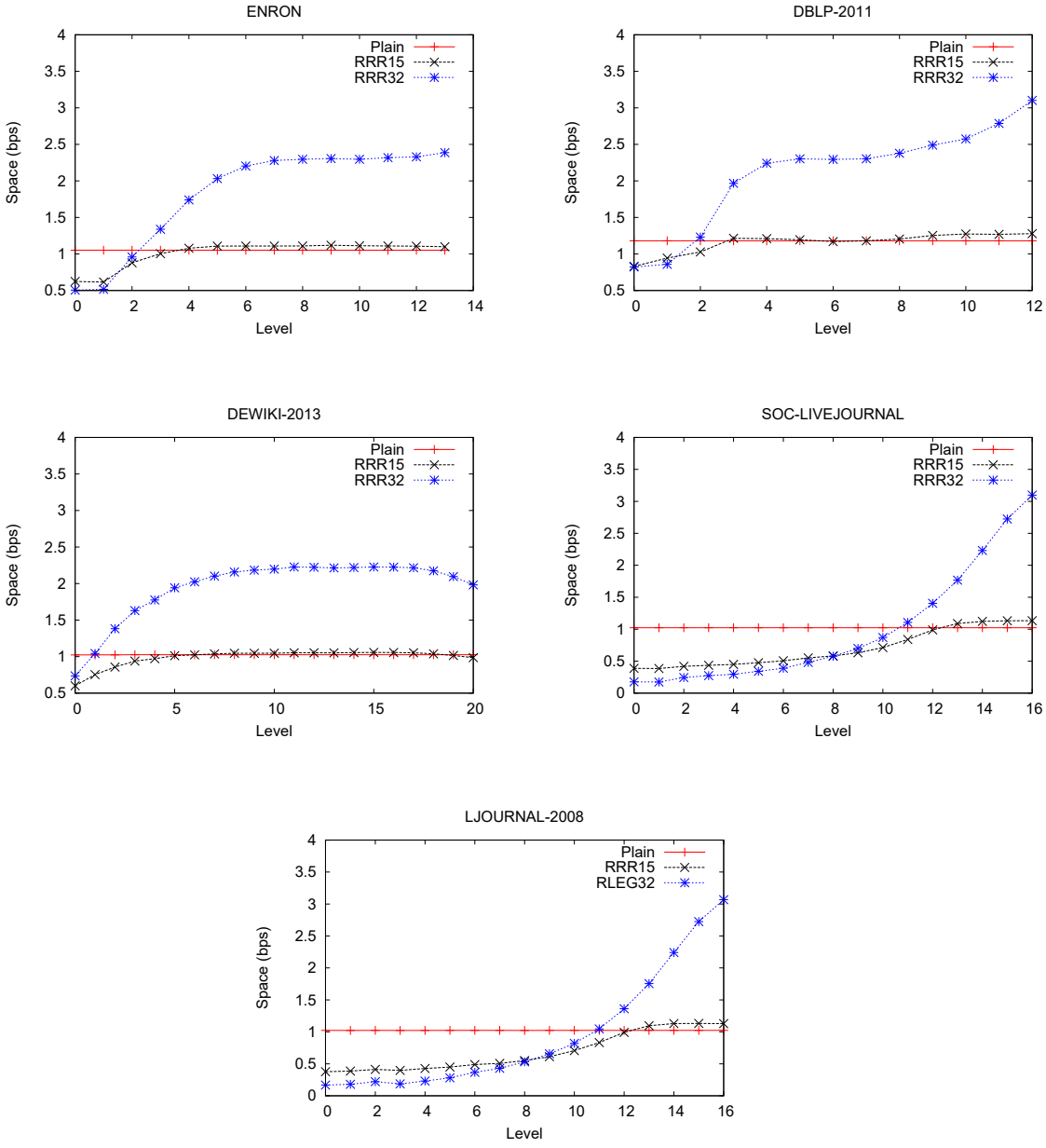


Figure 5.8: The compression performance of the bit vector at each level of the wavelet tree in social networks

Table 5.19: Optimum edgeSaving and level information

	optimum edgeSaving					Level	
Data Set	RRR15	Plain	RLEG32	RRR15-P	RL32-RRR15-P	RRR15-P	RL32-RRR15-P
eu-2005	30	500	50	30	15	15|19	9|15|19
in-2004	15	500	500	15	15	15|19	12|15|19
indochina-2004	50	500	100	30	15	18|22	15|18|22
uk-2002	30	500	100	30	15	19|23	16|19|23
arabic-2005	30	500	100	30	6	20|24	15|20|24
enron	30	30	6	30	30	3|13	1|3|13
dblp-2011	500	500	500	500	500	2|12	1|2|12
dewiki-2013	6	6	500	6	6	6|18|20	6|18|20
soc-livejornal	500	500	500	500	500	12|16	8|12|16
ljournal-2008	500	500	500	500	500	12|16	8|12|16

is desired.

It now makes sense to engineer the implementation of T_X by trying different block sizes of the bit vector implementations used in our new *combined encoding* approach. There is however one more consideration: as we follow Hernández and Navarro's approach and always tune the parameters of their mining algorithm to maximize the compression of the entire graph, having different implementations of T_X may change the parameters of the mining algorithm, which may affect the content of X itself. Therefore, when engineering the implementation of T_X, it makes sense to use the compression and query performance over the entire graph.

5.3.2 Engineering Wavelet Tree Implementation for Compressed Web Graph Representations

We now engineer the implementation of T_X, using the performance achieved over the entire graph to evaluate our approaches. For each bit vector implementation, we follow Hernández and Navarro's approach which uses different values for *edge_saving* and determine the value that achieves the most compression, as shown in Section 5.1. To get the best compression performance, Table 5.19 shows the optimum edgeSaving values for wavelet trees based on different bit vector structures. For combined encoding schemes, it also shows the last wavelet tree level at which each individual

Table 5.20: Space performance with optimum edgeSavings, measured in bits per edge

Data Set	Plain	RRR15	RLEG32	RRR15-P	RL32-RRR15-P
eu-2005	3.16	2.54	2.85	2.50	2.30
in-2004	2.91	2.43	2.60	2.39	2.12
indochina-2004	1.76	1.40	1.46	1.38	1.17
uk-2002	3.02	2.41	2.50	2.37	2.02
arabic-2005	2.29	1.80	1.89	1.78	1.49
enron	16.26	16.21	16.53	16.15	16.13
dblp-2011	8.61	8.61	8.62	8.61	8.61
dewiki-2013	17.09	17.09	18.40	16.99	16.99
soc-livejornal	12.94	12.92	12.94	12.92	12.92
ljournal-2008	12.96	12.93	12.95	12.93	12.93

bit vector encoding scheme is used. These levels are given in the last two columns of the table. As two different bit vector implementation are used in `RRR15-P`, two numbers are used when reporting such level information for `RRR15-P`, separated by "|". Similarly, three numbers are used for `RL32-RRR15-P`. For example, 15|20|24 are given for the `RL32-RRR15-P` encoding of arabic-2005, and this means the bitmaps from level 0 to level 15 are encoded by `RLEG32`, the bitmaps from level 16 to level 20 are encoded by `RRR`, and the bitmaps from level 21 to level 24 are encoded by `Plain`. We have two special cases when using `RL32-RRR15-P` to encode social networks: for dewiki-2013, `RLEG32` uses more space for all levels compared to `RRR15` and `Plain`, so the bitmaps from level 0 to level 6 are encoded by `RRR15`, the bitmaps from level 7 to level 18 are encoded by `Plain`, and bitmaps from level 19 to level 20 are encoded by `RRR15`; for dblp-2011, level 6 is encoded by `RRR15` even through this level is within the range of levels whose bit vectors are supposed to be encoded by `Plain`.

Table 5.20 and Table 5.21 show the compression performance and the query performance of pointerless wavelet trees combined with different bit vector structures by using the optimum edgeSaving values. In this section, the compression is measured in bits per edge which is the space cost of storing the dense subgraph and the remaining

Table 5.21: Query performance with optimum edgeSavings, measured in microseconds per out-neighbours

Data Set	Plain	RRR15	RLEG32	RRR15-P	RL32-RRR15-P
eu-2005	2.73	5.32	15.15	4.87	7.95
in-2004	3.18	5.47	11.72	5.21	11.01
indochina-2004	3.87	5.42	15.05	5.37	12.41
uk-2002	9.33	11.40	23.68	10.86	19.44
arabic-2005	5.18	7.60	19.65	7.20	14.62
enron	5.60	5.78	10.77	5.55	5.77
dblp-2011	4.18	4.34	5.74	4.26	4.29
dewiki-2013	11.12	13.88	11.24	12.73	12.86
soc-livejornal	7.42	7.73	9.37	7.66	9.21
ljournal-2008	6.60	7.05	9.12	6.88	8.60

Table 5.22: Optimum edgeSaving II

Data Set	RLEG4	RLEG6	RLEG8	RLEG16	RLEG64	RLEG128	RLEG256	RRR31	RRR63
eu-2005	500	500	500	500	30	15	15	30	15
in-2004	500	500	500	500	15	15	15	15	15
indochina-2004	500	500	500	500	30	15	15	15	15
uk-2002	500	500	500	500	30	15	15	15	15
arabic-2005	500	500	500	500	30	6	6	6	6

graph in bits divided by the total number of edges of the graph. The queries were out-neighbour queries performed on the entire graph, and the query times reported are in microseconds per out-neighbour reported. For the space performance, RL32-RRR15-P achieves an improvement of around 9% to 19% over RRR15 for web graphs. For social graphs, RL32-RRR15-P does not achieve any significant improvement. For the query performance, RRR15-P achieves a slight improvement over RRR15 for both web graphs and social networks. RL32-RRR15-P is around twice as slow as RRR15 for web graphs, and the query performance is similar for social networks. Overall, combined encoding achieved significant improvements for web graphs only. Thus, the rest of this section will focus on web graphs only.

We first report, in Tables 5.22, 5.23 and 5.24, the optimum edgeSaving values, the compression performance and the query performance achieved by using existing bit

Table 5.23: Space performance over the entire graph of using existing approaches, measured in bpe

data set	Plain	RRR15	RRR31	RRR63	RLEG4	RLEG6	RLEG8	RLEG16	RLEG32	RLEG64	RLEG128	RLEG256	k^2-tree	DSM1	DSM2
eu-2005	3.16	2.54	2.44	2.41	5.13	4.29	3.86	3.23	2.85	2.51	2.32	2.21	3.46	2.02	2.34
in-2004	2.91	2.43	2.32	2.29	4.46	3.76	3.40	2.87	2.60	2.34	2.18	2.10	2.65	1.64	1.67
indochina-2004	1.76	1.40	1.31	1.28	2.89	2.35	2.09	1.69	1.46	1.30	1.19	1.14	1.72	0.81	0.86
uk-2002	3.02	2.41	2.26	2.20	4.48	3.75	3.38	2.83	2.50	2.23	2.06	1.98	2.82	1.44	1.56
arabic-2005	2.29	1.80	1.66	1.60	3.64	2.99	2.66	2.17	1.89	1.67	1.50	1.42	2.45	1.03	1.12

Table 5.24: Query performance over the entire graph of using existing approaches, measured in μs/edge

data set	Plain	RRR15	RRR31	RRR63	RLEG4	RLEG6	RLEG8	RLEG16	RLEG32	RLEG64	RLEG128	RLEG256	k^2-tree	DSM1	DSM2
eu-2005	2.73	5.32	10.13	13.47	4.74	5.26	5.60	7.58	15.15	27.50	48.79	93.67	1.13	12.33	4.84
in-2004	3.18	5.47	10.02	12.38	4.75	5.22	5.78	7.62	11.72	24.04	45.42	91.28	1.36	3.95	1.94
indochina-2004	3.87	5.42	10.90	13.17	6.14	6.69	7.33	10.01	15.05	29.67	57.53	112.46	1.76	5.17	2.27
uk-2002	9.33	11.40	17.60	20.76	12.10	12.64	13.20	15.95	23.68	37.34	67.80	123.59	7.57	16.61	7.56
arabic-2005	5.18	7.60	13.75	16.95	8.65	9.21	9.84	12.72	19.65	35.98	63.87	119.56	2.80	13.21	5.25

vector implementations to encode T_X. We also include three different approaches: k^2-tree, and two tradeoffs of DSM. The first tradeoff, denoted by DSM1, chooses parameters ($ES = 10$ and $T = 10$; see [22]) to maximize compression. The second tradeoff, denoted by DSM2 ($ES = 100$ and $T = 10$), outperforms RRR15 in both query time and space cost. Note that none of these three approaches can support mining queries, so we include them here only to show where our results stand when compared to some state-of-the-art web graph compression approaches that support fewer operations; we will make more comments about them in later part of the section.

From these two tables, we can draw the same conclusion as in Section 5.3.1 on the performance of existing approaches. What is new is that we tested many different choices for the block size of RLEG (we only use RLEG as it outperforms RLED and RLEO): 4, 6, 8, 16, 32, 64, 128, and 256. It is unusual to use small blocks sizes such as 4, 6 or 8, as they require too much space overhead. However, as the bit vectors at the top levels of T_X have very long runs of the same bit, these approaches can still use less space than RRR15 for these levels, and their fast support for queries is attractive.

Table 5.25 shows the optimum edgeSaving values and the last wavelet tree level at which each individual bit vector encoding scheme is used for for RL32-RRR31-P and RL32-RRR63-P. These levels are given in the last two columns of the table. Table 5.26

Table 5.25: Optimum edgeSaving for combined encoding

	optimum edgeSaving		Level	
Data Set	RL32-RRR31-P	RL32-RRR63-P	RL32-RRR31-P	RL32-RRR63-P
eu-2005	15	15	8|15|19	8|15|19
in-2004	15	15	14|*|19	14|*|19
indochina-2004	15	15	15|18|22	17|18|22
uk-2002	15	15	15|18|23	18|19|23
arabic-2005	6	6	14|20|24	13|24|*

Table 5.26: Space Performance of using different block sizes of RRR in combined encoding, measured in bits per edges

Data Set	RL32-RRR15-P	RL32-RRR31-P	RL32-RRR63-P
eu-2005	2.30	2.31	2.34
in-2004	2.12	2.14	2.14
indochina-2004	1.17	1.18	1.18
uk-2002	2.02	2.04	2.05
arabic-2005	1.49	1.49	1.51

Table 5.27: Query Performance of using different block sizes of RRR in combined encoding, measured in microseconds per edge

Data Set	RL32-RRR15-P	RL32-RRR31-P	RL32-RRR63-P
eu-2005	7.95	8.81	9.45
in-2004	11.01	11.86	11.96
indochina-2004	12.41	13.36	14.11
uk-2002	19.44	19.70	21.46
arabic-2005	14.62	15.94	19.89

Table 5.28: Optimum edgeSaving III

	optimum edgeSaving				
data set	RL4-RRR15-P	RL6-RRR15-P	RL8-RRR15-P	RL16-RRR15-P	RL64-RRR15-P
eu-2005	30	30	30	30	15
in-2004	30	30	15	15	15
indochina-2004	30	30	30	15	15
uk-2002	30	30	30	15	15
arabic-2005	30	30	6	6	6
	level				
eu-2005	3|15|19	4|15|19	6|15|19	8|15|19	13|15|19
in-2004	7|15|19	8|15|19	8|15|19	10|15|19	15|*|19
indochina-2004	9|18|22	10|18|22	11|18|22	13|18|22	17|18|22
uk-2002	11|19|23	12|19|23	13|19|23	13|19|23	18|19|23
arabic-2005	11|20|24	11|20|24	11|20|24	12|20|24	17|20|24

and Table 5.27 show the compression performance and the query performance for RL32-RRR15-P, RL32-RRR31-P, and RL32-RRR63-P. In table 5.25, "*" is used in some cells to indicate that the corresponding bit vector encoding scheme is not used at all for a particular files, because at any level of the wavelet tree, this schemes uses more space than at least one alternative approach. For example 14|*|19 in the cell corresponding to in-2004 and RL32-RRR31-P means that RLEG32 is used for levels 0-14, Plain is used for levels 15-19, and RRR15 is not used. We have found that RL32-RRR31-P and RL32-RRR63-P not only provide slower support for queries than RL32-RRR15-P does, but also use more space, which is surprising. We found that this is because, in combined encodings, RRR is used to encode the bit vectors in T_X that are compressible but are not highly compressible (the highly compressible ones are encoded using RLEG), and for these bit vectors, increasing the block size in RRR does not improve compression. Therefore, the best strategy is to use RRR15 in combined encodings.

We now test the performance of combined encodings using different block sizes for RLEG. Table 5.28 shows the optimum edgeSaving values and the last wavelet tree level at which each individual bit vector encoding scheme is used for RL4-RRR15-P, RL6-RRR15-P, RL8-RRR15-P, RL16-RRR15-P and RL64-RRR15-P. These levels are given in

Table 5.29: Space performance over the entire graph of using combined encodings, measured in bits per edge

data set	RRR15-P	RL4-RRR15-P	RL6-RRR15-P	RL8-RRR15-P	RL16-RRR15-P	RL32-RRR15-P	RL64-RRR15-P
eu-2005	2.50	2.46	2.43	2.41	2.35	2.30	2.23
in-2004	2.39	2.28	2.25	2.23	2.17	2.12	2.07
indochina-2004	1.38	1.27	1.25	1.23	1.20	1.17	1.13
uk-2002	2.37	2.23	2.18	2.15	2.08	2.02	1.97
arabic-2005	1.78	1.65	1.61	1.59	1.54	1.49	1.44

Table 5.30: Query performance over the entire graph of using combined encodings, measured in μs/edge

data set	RRR15-P	RL4-RRR15-P	RL6-RRR15-P	RL8-RRR15-P	RL16-RRR15-P	RL32-RRR15-P	RL64-RRR15-P
eu-2005	4.87	4.85	5.08	5.46	6.56	7.95	13.39
in-2004	5.21	5.07	5.36	5.87	7.48	11.01	20.48
indochina-2004	5.37	5.53	5.87	6.19	8.28	12.41	24.46
uk-2002	10.86	11.54	11.70	12.19	14.63	19.44	32.87
arabic-2005	7.20	7.66	8.04	8.47	10.06	14.62	25.14

the bottom of the table. Table 5.29, and Table 5.30 show the compression performance and the query performance for combined encoding schemes. They not only contain the results for RRR15-P and RL32-RRR15-P as defined in Section 5.3, but also contain the performance of several more approaches that use different block sizes for RLEG. In particular, we used block sizes 4, 6, 8, 16, 32 and 64, and the names of these combined approaches are defined similarly as in Section 5.3. To make it easier to see how combined encodings compare to previous approaches, Figure 5.9 illustrates the results reported in the tables in this section. In these figures, the data points corresponding to RLEG use block sizes 16, 32, 64 and 128, the data points for RRR use block sizes 15, 31 and 63, and the data points for RL-RRR15-P includes all the combined encoding approaches in Tables 5.29 and 5.30 that combines RLEG, RRR and Plain.

From these tables and figures, we can tell that our combined encoding approach indeed achieves more desirable tradeoffs than what can be achieved by using a single bit vector encoding for T_X. Our strategy of using three different bit vector encodings to represent T_X can improve the space cost of using RRR only, without sacrificing as much query performance as RLEG would. RL32-RRR15-P, RL6-RRR15-P and RRR15-P

are the three interesting tradeoffs described in Section 1.1.

As with Hernández and Navarro's implementation, our new tradeoffs achieves more compression than k^2-trees, though k^2-trees support faster queries. `DSM1`, which outperforms the original implementation in both query and space, still outperforms the new tradeoffs, though the difference is now smaller. We would like to remind the readers that `DSM` does not support mining queries, so our implementation is still the best if these queries are desired.

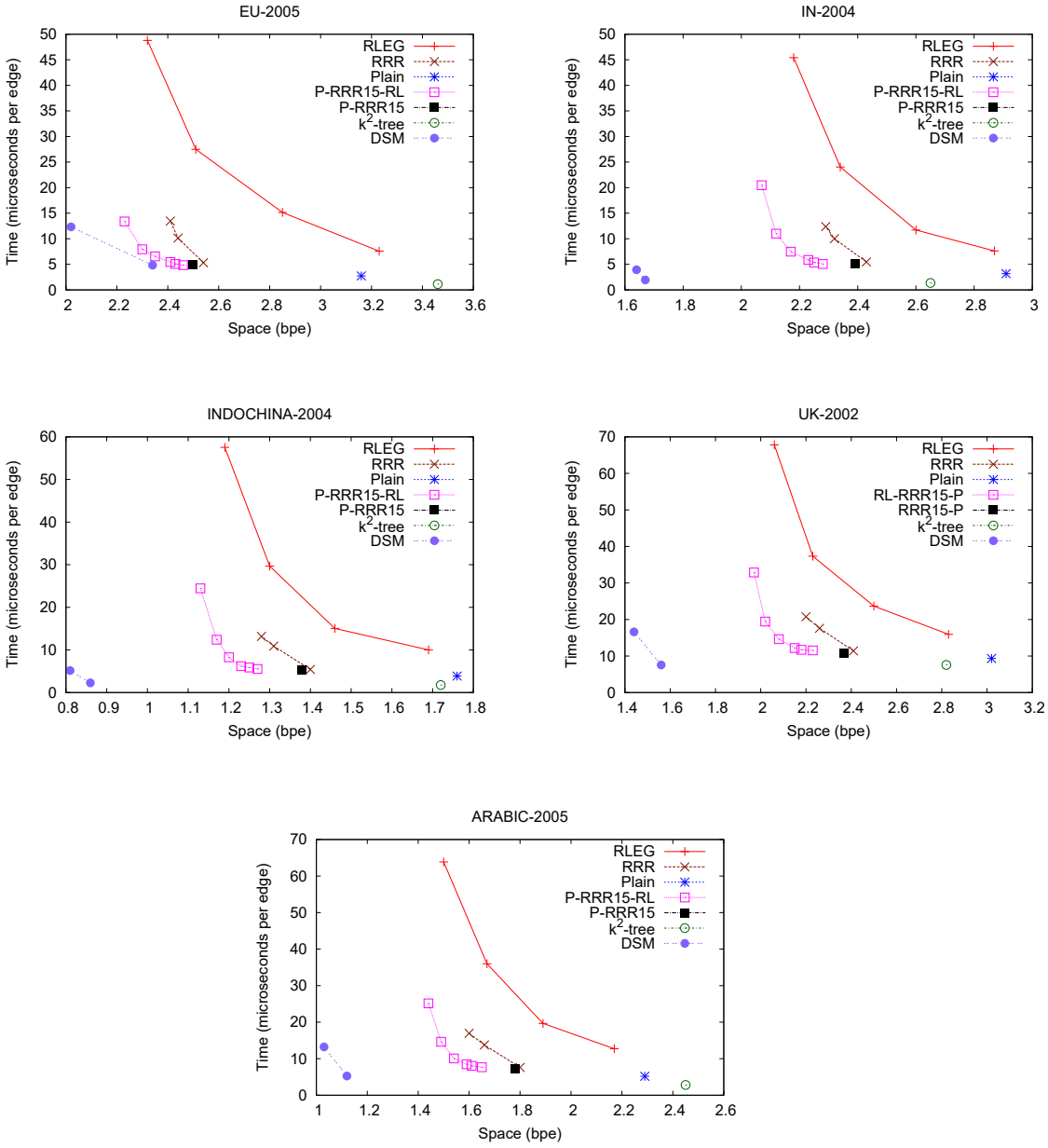


Figure 5.9: Space/Query performance on web graphs

Chapter 6

Conclusions and Future Work

This thesis studies compressed representation of web graphs and social networks based on discovering dense subgraphs, which was originally proposed by Hernández and Navarro's [22]. In these representations, dense subgraphs are represented by succinct data structure such as wavelet trees.

We first conducted a survey on succinct data structures that can be used in these graph structures, including different bit vectors structures, such as `RRR` and `RLE`, and different implementations of wavelet trees such as the pointer-based wavelet trees and pointerless wavelet trees.

Then we used these data structures in Hernández and Navarro's framework to represent web graphs and social networks. We found that pointer-based wavelet trees are not competitive compared to pointer-less wavelet tree in dense subgraph representations due to the hug space overhead incurred by the large size of the alphabets in our experimental studies.

We thus propose a wavelet tree encoding scheme called *combined encoding*. This approach allowed us to achieve new results for web graph representations that can not be achieved by changing wavelet tree shapes or using an existing bit vector implementation for all levels of a wavelet tree. In addition, our structure is simple and easy to implement.

Despite the progress made for web graphs, our approach does not achieve significant improvement over previous results for social networks. Thus, further improving

the compressed representations of social networks is a future research direction. Another possible future research direction is to study the parallel construction of these data structures as the mining phase of the construction algorithm requires a lot of time.

Bibliography

[1] Micah Adler and Michael Mitzenmacher. Towards compressing web graphs, Snowbird, Utah, USA. In *the 11th Data Compression Conference*, pages 203–212, March, 2001.

[2] Alberto Apostolico and Guido Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[3] Jérémy Barbay and Gonzalo Navarro. Compressed representations of permutations, and applications, Freiburg, Germany. In *the 26th International Symposium on Theoretical Aspects of Computer Science*, pages 111–122, February, 2009.

[4] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. Permuting web and social graphs. *Internet Mathematics*, 6(3):257–283, 2009.

[5] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques, New York, NY, USA. In *the 13th International Conference on World Wide Web*, pages 595–602, May, 2004.

[6] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN systems*, 30(1-7):107–117, 1998.

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

[8] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[9] Gregory Buehrer and Kumar Chellapilla. A scalable pattern mining approach to web graph compression with communities, Palo Alto, California, USA. In *the 1st International Conference on Web Search and Data Mining*, pages 95–106, February, 2008.

[10] Meeyoung Cha, Alan Mislove, and Krishna P Gummadi. A measurement-driven analysis of information propagation in the flickr social network, Madrid, Spain. In *the 18th International Conference on World Wide Web*, pages 721–730, April, 2009.

[11] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks, Paris, France. In *the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 219–228, June, 2009.

[12] David R Clark and J Ian Munro. Efficient suffix trees on secondary storage, Atlanta, Georgia. In *the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, January, 1996.

[13] Francisco Claude and Susana Ladra. Practical representations for web and social graphs, Glasgow, United Kingdom. In *the 20th ACM International Conference on Information and Knowledge Management*, pages 1185–1190, October, 2011.

[14] Francisco Claude and Gonzalo Navarro. Fast and compact web graph representations. *ACM Transactions on the Web*, 4(4):16, 2010.

[15] Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences, Melbourne, Australia. In *the 15th International Symposium on String Processing and Information Retrieval*, pages 176–187, November, 2008.

[16] Francisco Claude, Gonzalo Navarro, and Alberto Ordónez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

[17] Peter Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.

[18] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009.

[19] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries, Santorini Island, Greece. In *the 4th Workshop on Efficient and Experimental Algorithms*, pages 27–38, May, 2005.

[20] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes, Baltimore, Maryland, USA. In *the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, January, 2003.

[21] Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. Wavelet trees: From theory to practice, Palinuro, Cilento Coast, Italy. In *Data Compression, Communications and Processing, the 1st International Conference on*, pages 210–221, June, 2011.

[22] Cecilia Hernández and Gonzalo Navarro. Compressed representations for web and social graphs. *Knowledge and Information Systems*, 40(2):279–313, 2014.

[23] David A Huffman et al. A method for the construction of minimum redundancy codes. *The Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[24] Hongwei Huo, Longgang Chen, Heng Zhao, Jeffrey Scott Vitter, Yakov Nekrich, and Qiang Yu. A data-aware FM-index, San Diego, CA, USA. In *the 17th Workshop on Algorithm Engineering and Experiments*, pages 10–23, January, 2015.

[25] Guy Jacobson. Space-efficient static trees and graphs, North Carolina, USA. In *the 30th Annual Symposium on Foundations of Computer Science*, pages 549–554, October, 1989.

[26] Juha Karkkainen, Dominik Kempa, and Simon J Puglisi. Hybrid compression of bitvectors for the FM-index, Snowbird, UT, USA. In *the 24th Data Compression Conference*, pages 302–311, March, 2014.

[27] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[28] Veli Mäkinen and Gonzalo Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. *Technical report C-2004-20 (April). University of Helsinki, Helsinki, Finland*, 2004.

[29] Hossein Maserrat and Jian Pei. Neighbor query friendly compression of social networks, Washington, DC, USA. In *the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 533–542, July, 2010.

[30] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks, San Diego, California, USA. In *the 7th ACM SIGCOMM Conference on Internet Measurement*, pages 29–42, October, 2007.

[31] Katarzyna Musiał, Przemysław Kazienko, and Piotr Bródka. User position measures in social networks, Paris, France. In *the 3rd Workshop on Social Network Mining and Analysis*, page 6, June, 2009.

[32] Gonzalo Navarro and Alberto Ordónez. Compressing huffman models on large alphabets, Snowbird, UT, USA. In *the 23rd Data Compression Conference*, pages 381–390, March, 2013.

[33] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps, Bordeaux, France. In *Experimental Algorithms: 11th International Symposium*, pages 295–306. June, 2012.

[34] Gonzalo Navarro, Simon J. Puglisi, and Daniel Valenzuela. General document retrieval in compact space. *Journal of Experimental Algorithmics*, 19:2.3:1.1–2.3:1.46, 2015.

[35] Mihai Pătraşcu. Succincter, Philadelphia, PA, USA. In *the 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–313, October, 2008.

[36] 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*, 3(4):43, 2007.

[37] Keith H Randall, Raymie Stata, Rajiv G Wickremesinghe, and Janet L Wiener. The link database: Fast access to graphs of the web, Snowbird, UT, USA. In *the 12th Data Compression Conference*, pages 122–131, April, 2002.

[38] Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds, Miami, Florida, USA. In *the 17th annual ACM-SIAM Symposium on Discrete Algorithm*, pages 1230–1239, January, 2006.

[39] Hiroo Saito, Masashi Toyoda, Masaru Kitsuregawa, and Kazuyuki Aihara. A large-scale study of link spam detection by graph algorithms, Banff, Canada. In *the 3rd International Workshop on Adversarial Information Retrieval on the Web*, pages 45–48, May, 2007.

[40] Kazumi Saito, Masahiro Kimura, Kouzou Ohara, and Hiroshi Motoda. Efficient discovery of influential nodes for sis models in social networks. *Knowledge and Information Systems*, 30(3):613–635, 2012.

[41] Torsten Suel and Jun Yuan. Compressing the graph structure of the web, Snowbird, Utah, USA. In *the 11th Data Compression Conference*, pages 213–222, March, 2001.

[42] Dong Zhou, David G. Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences, Rome, Italy. In *Experimental Algorithms, 12th International Symposium*, pages 151–163, June, 2013.