UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MANAGING MASSIVE GRAPHS

TESIS PARA OPTAR AL TÍTULO DE GRADO DE DOCTOR EN CIENCIAS,
MENCIÓN COMPUTACIÓN

CECILIA PAOLA HERNÁNDEZ RIVAS

PROFESOR GUÍA:
GONZALO NAVARRO BADINO
MAURICIO MARÍN CAIHUAN

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
JORGE PÉREZ ROJAS
SEBASTIANO VIGNA

Este trabajo ha sido parcialmente financiado por Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Fondecyt Grant 1-110066, por FONDEF IDeA CA12i10314, Beca Conicyt y Beca Universidad de Concepción

SANTIAGO DE CHILE
OCTUBRE 2014

Resumen

Con la popularidad de la Web y, mas recientemente, el amplio uso de las redes sociales, la necesidad de procesar y encontrar información en grafos muy grandes impone varios desafíos: Cómo procesar grafos muy grandes eficientemente, dado que probablemente son muy grandes para la memoria disponible, o incluso si la memoria es suficiente, realizar un paso sobre el grafo es todavía caro computacionalmente? Cómo almacenar esos grafos eficientemente, para ser archivados, o para ejecutar algoritmos de grafos? Cómo descubrir información relevante tal como componentes densos, comunidades, u otras estructuras?

Se han propuesto tres enfoques para manejar grafos grandes. El primero es usar formatos de grafos comprimidos que permiten consultas de navegación básicas directamentee sobre la estructura comprimida, sin la necesidad de descompresión. Esto permite simular cualquier algoritmo de grafo en memoria principal usando mucho menos espacio que la representación plana. Una segunda línea de investigación se focaliza en usar modelos de stream o semi-stream de datos de manera de procesar secuencialmente, idealmente en un paso sobre el disco, usando una cantidad limitada de memoria principal. La tercera línea es el uso de sistemas distribuidos y paralelos donde la memoria es agregada sobre múltiples unidades de procesamiento para procesar el grafo en paralelo.

En esta tesis presentamos varios enfoques para manejar grafos grandes (con arcos sin etiquetas) considerando los tres enfoques. Primero, buscamos por patrones que aparecen en grafos de la Web y redes sociales los que podemos representar en forma compacta, en particular mostramos como generalizar algoritmos para encontrar cliques o bicliques para encontrar sub-estructuras densas que comprimen ambas. Segundo, basado en estos subgrafos densos, proponemos esquemas comprimidos que soportan consultas de vecinos directos y reversos, así como otras consultas mas complejas sobre subgrafos densos. Algunas de las contribuciones combinan técnicas del estado del arte mientras otras incluyen representaciones comprimidas novedosas basadas en estructuras de datos compactas. Encontrar subgrafos densos es una tarea que consume tiempo y espacio, así que proporcionamos algoritmos de streaming and algoritmos de memoria externa para descubrir subgrafos densos, asi como también algoritmos distribuidos para construir las estructuras básicas que usamos para las representaciones comprimidas.

Abstract

With the popularity of the Web and, more recently, the widespread use of social networks, the need to process and find information in very large graphs imposes several challenges: How to process such large graphs efficiently, since they probably do not fit in main memory, or even if they do, performing one pass over the graph is still expensive? How to store those graphs efficiently, be it for archival, or to run graph algorithms? How to discover relevant information such as dense components, communities, or other substructures?

Three main approaches have been proposed to manage large graphs. The first is to use graph compression formats that support basic navigation queries directly over the compressed structure, without the need of decompression. This allows simulating any graph algorithm in main memory using much less space than a plain representation. A second line of research focuses on using data streaming or semi-streaming models, so as to process graphs sequentially, ideally in one pass over the disk, using limited main memory. The third line is the use of parallel and distributed systems, where memory is aggregated over multiple processing units that process the graph in parallel.

In this thesis we present several approaches for managing large graphs (with unlabeled edges) considering the three approaches. First, we look for patterns that arise in Web and social graphs and enable us to represent them compactly, in particular showing how to generalize algorithms that find cliques or bicliques, so as to find dense substructures that comprise both. Second, based on those dense subgraphs, we propose compression schemes that support out/in-neighbor queries, as well as other more complex queries over dense subgraphs. Some of the contributions combine state-of-the-art techniques while others include novel compressed representations based on compact data structures. Finding the dense subgraphs is a time and space consuming task, so we provide streaming and external memory algorithms for discovering dense subgraphs, as well as distributed algorithms for building the basic structures we use in our compressed representations.

Agradecimientos

Quiero agradecer a mi familia que me ha apoyado incondicionalmente durante todo el desarrollo del programa de doctorado. A mi esposo Miguel, e hijos José Miguel y Martín, por sus palabras de aliento y comprensión por el tiempo que mis estudios me impidieron estar presente en sus vidas. A mi mamá y hermanos que siempre manifestaron interés en mis estudios y me brindaron todo su apoyo.

A mis profesores guías Gonzalo Navarro y Mauricio Marín, quienes no sólo me han transmitido conocimientos sino también por la excelente disposición que siempre tuvieron en responder consultas, conversar ideas y corregir el texto de este documento. Además quiero agradecerles por la motivación transmitida por su gran interés en el desarrollo de sus respectivas disciplinas. A los profesores de la comisión Sebastiano Vigna, Benjamín Bustos y Jorge Pérez que han revisado mi tesis proporcionándome observaciones y comentarios valiosos para mejorar mi trabajo.

También quiero agradecer a Susana Ladra, Francisco Claude y Diego Arroyuelo, quienes siempre tuvieron una muy buena disposición a responder consultas. Además quiero agradecer a Ren Cerro, del Departamento de Revisión de Inglés del DCC, quien me ayudó a corregir el inglés de este documento.

A CONICYT por la beca de doctorado y a la beca Universidad de Concepción que me permitieron financiar en gran parte el costo del programa y estadía en Santiago durante los primeros años. Al Departamento de Ingeniería Civil Informática de la Universidad y a la Universidad de Concepción que permitieron que me ausentara de mi trabajo como docente por gran parte del periodo que me ha tomado terminar mis estudios. Al Laboratorio Nacional de Computación de Alto Rendimiento (NLHPC) liderado por el Centro de Modelamiento Matemático de la Universidad de Chile, que me permitió usar recursos computacionales paralelos para una parte del desarrollo de mi tesis.

También quiero agradecer al Departamento de Ciencias de la Computación de la Universidad de Chile, al Departamento de Ingeniería Civil Informática de la Universidad de Concepción, a los proyectos Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Fondecyt Grant 1-110066, y FONDEF IDeA CA12i10314 por el apoyo financiero brindado para asistir a las conferencias internacionales donde se publicaron los resultados obtenidos en la tesis.

Finalmente quiero agradecer a todos mis colegas y amigos por sus palabras de aliento y buenos deseos en el éxito de mis estudios.

Contents

List of Tables

List of Figures

Introduction

Massive graphs appear in a wide range of domains including the Web, social networks, RDF graphs, protein networks and many more. For instance, on a recent estimation the Web graph has more than 7.8 billion pages with more than 200 billion edges[1], and the social network Facebook has more than 950 million active users worldwide.[2] Other large graphs include LinkData with about 31 billion of RDF triples and 504 million RDF links[3]. Managing and mining large graphs imposes several challenges triggered by different aspects, such as the data volume itself, data complexity, how fast the data is being generated, and application needs [109]. In our research, we focus specifically on large Web graphs and social networks, where graphs are modeled with unlabeled edges.

Web graphs represent the link structure of the Web. They are usually modeled as directed graphs where nodes represent pages and edges represent links among pages. On the other hand, social networks represent relationships among social entities. These networks are modeled by undirected or directed graphs depending on the relation they model. For instance, the friendship relation in Facebook is symmetric and then it is modeled by an undirected graph, whereas the "following" relation on Twitter and LiveJournal is not symmetric and therefore it is modeled by a directed graph.

The link structure of Web graphs is often used by ranking algorithms such as PageRank [21] and HITS [77], as well as for spam detection [13, 104], for detecting communities [80, 56], and for understanding the structure and evolution of the network [54, 56]. A social network structure is often used for mining and analysis purposes, such as identifying interest groups or communities, detecting important actors [105], and understanding information propagation [89, 30]. Those algorithms require a graph representation that supports at least forward navigation (i.e., to the out-neighbors of a node, or those pointed from it), and many require backward navigation as well (i.e., to the in-neighbors of a node, or those that point to it).

In the last decade, various algorithms have been proposed to address problems associated with large graphs. However, just from the size point of view, it is unlikely that we can store graphs with a billion nodes and even more edges, in the memory of a single commodity machine. For this reason different approaches have been used to manage large graphs, such as using compression, data streaming and parallel/distributed systems.

[1] `http://www.worldwidewebsize.com`, on August 6, 2012.

[2] `http://newsroom.fb.com/content/default.aspx?NewsAreaId=22` considering June 2012.

[3] http://www.w3.org/

Compressed data structures aim to reduce the amount of memory used by representing graphs in compressed form while being able to answer the queries of interest without decompression. Even though these compressed structures are usually slower than uncompressed representations, they are still much faster than incurring I/O costs: They can be orders of magnitude faster when they can fit completely in main memory graphs that would otherwise require disk storage. Most compression methods are based on exploiting patterns that provide compression opportunities, such as locality and similarity of adjacency lists, sparseness and clustering of the adjacency matrix, node ordering algorithms, and representing dense patterns more compactly.

Streaming and semi-streaming techniques can be applied with the goal of processing the graph sequentially, ideally in one pass, although a few passes are allowed. The idea is to use main memory efficiently, avoiding random access to disk [52]. External memory algorithms define memory layouts that are suitable to run graph algorithms, where the goal is to exploit locality in order to reduce I/O costs, reducing random accesses to disk [115]. Another approach is the use of parallel and distributed systems, where distributed memory is aggregated to process the graph [111].

A brief outline of the thesis is presented here. In addition, we provide a list of the accepted and published papers that derive from it. It is important to note that our sequential implementation for building the graph factoring edges is not refined enough to support the largest Web graph available at the Webgraph project web site (`http://law.di.unimi.it/datasets.php`) on a server with 96GB of main memory. Such graph has almost 1 billion nodes with 43 billion edges. However, we were able to use the second largest Web graph snapshot (with 50 million nodes and 2 billion edges). Based on the performance trend we observe on the graphs we tried, it is reasonable to expect that we would probably achieve similar results with larger graphs using a more refined version of the application.

Basic Concepts

This chapter reviews basic concepts including data compression, encodings, graph compression, and compact data structures for arbitrary sequences and bitmaps. The chapter describes state-of-the-art compression patterns and techniques used for Web and social graph compression with navigation capabilities. In particular, it describes Re-Pair and subsequent improvements [42, 39], VNM [29], the Webgraph framework and improvements [19, 18, 16], using BFS node ordering [9], k2tree [22, 81], and MPk and improvements [85, 39].

Discovering Graph Patterns

In Chapter 2, our goal is to study different patterns found on graphs that might help to represent them in a compact form. Some previous patterns that have been studied include grammar-based such as Re-Pair [42]. Discovering dense subgraphs in large graphs is a challenging task that has many applications, including community mining, spam detection, social analysis, recommendation systems and bioinformatics. In this context, we find that repre-

senting dense structures such as complete bipartite cliques (bicliques) and more general dense subgraphs allows us to compress them.

We improve discovery algorithms for detecting dense substructures that can be used for reducing edges, and algorithms for extracting such structures. We compare our new algorithm with clustering algorithms such as MCL (Markov Clustering Algorithm) [114]. We use synthetic graph generators and real graphs for evaluation in terms of quality and processing times. We show that our algorithm is similar to MCL in terms of the solution quality, but much faster.

Compression

This part of the thesis is divided in two chapters. Chapters 3 and 4 describe how to exploit the different patterns we discover to compress the graphs and at the same time, provide different types of navigation over compressed representations. In particular, we exploit dense structures, such as complete bipartite cliques (bicliques), and less restrictive dense subgraph patterns. We achieve compression using compact data structures as well as other compression techniques.

We first describe a compression scheme that considers, as dense subgraphs, subgraphs that are complete bipartite graphs (biclique) where each biclique is formed by two nonoverlapping sets S and C. Using this dense subgraph definition we describe a compression scheme that transforms the original graph into a graph where the edges between S and C are factored by adding a virtual node between them. The result is a graph that has between four and ten times fewer edges than the original graph. We apply different compression techniques over this graph and achieve good compression and access times for Web graphs. We also present a compact data structure, based on the collection of bicliques extracted, using two sequences and two bitmaps. We achieve competitive space results, but a much slower in/out-neighbor support. We also evaluate other queries of interest over such representations.

Next, we show that discovering dense bipartite graph patterns, with overlapping sets S and C, we can improve compression with in/out-neighbor query support. We show that if we use these dense subgraphs with virtual nodes, apply BFS traversal on the node ordering and then k2tree, we obtain the best state-of-the-art compression for Web graphs. However, in/out-queries are twice as slow. We also describe a compressed structure based on a compact data structure that represents the edges between S and C implicitly, using only one symbol sequence and one bitmap. We achieve competitive compression and in/out-neighbor access for Web graphs. In the case of social graphs it is possible to improve compression and in/out-neighbor access if used in combination with the MPk implementation given by Claude and Ladra [39]. In addition, this structure enables mining queries based on the discovered dense subgraphs.

Streaming and External Memory

Chapter 5 describes algorithms for discovering our dense subgraphs using semi-streaming and external algorithms with the goal of reducing the main memory and achieving good quality results and processing times. We propose an external memory algorithm based on *R-way merge sort* and a semi-streaming algorithm based on Hierarchical Heavy Hitters [112]. We show that using a two level *R-way merge sort* enables us to reduce by a half main memory requeriments, yet doubling processing time, with respect to the original algorithm and keeping the same quality results. We also show that applying HHH does not allow reducing much main memory, but helps identify larger dense subgraphs.

Distributed and Parallel Systems

We also aim to provide algorithms that exploit parallelism and are able to use distributed memory to deal with large amounts of data.

In Chapter 6, we design and implement distributed and parallel algorithms for our dense subgraphs discovery algorithm. We first describe a MapReduce algorithm that extracts dense subgraphs and apply them on Web and social graphs. Second, we provide parallel algorithms based on BSP for building compressed structures using dense subgraphs with virtual nodes. We also provide a BSP algorithm for extracting dense subgraphs using dynamic load balancing in order to build the compressed structure based on implicit representation. Using dynamic load balancing is crucial for achieving good speedups and resource efficiency. We show that data locality of our graphs is an important factor for providing scalable algorithms.

Publications derived from the Thesis

The main contributions of the thesis have appeared in the following papers.

1. Compression of Web and Social Graphs supporting Neighbor and Community Queries. Cecilia Hernández and Gonzalo Navarro. The 5th SNA-KDD Workshop, August 21, 2011, San Diego, CA, USA. Copyright 2011 ACM 978-1-4503-0225-8. Chapter 3 Section 3.2 and 4 Section 4.2 describe the main results of this paper.
2. Compressed Representation of Web and Social Networks via Dense Subgraphs. Cecilia Hernández and Gonzalo Navarro. The 19th International Symposium String Processing and Information Retrieval, SPIRE 2012. Cartagena de Indias, Colombia, October 2012. Chapter 3 Section 3.3.1 and Chapter 4 Section 4.4 describe the main results of this paper.
3. Compressed Representations for Web and Social graphs. Cecilia Hernández and Gonzalo Navarro. Knowledge and Information Systems. Vol. 40, number 2, pages 279-313, 2014. Chapters 2, 3, and 4 describe the results of this paper.
4. Discovering Dense Subgraphs in Parallel for Compressing Web and Social networks. Cecilia Hernández and Mauricio Marín. The 20th International Symposium String

Processing and Information Retrieval, SPIRE 2013. Jerusalem, Israel, October 2013. Chapter 6 describes the results of this paper.

Chapter 1

Basic Concepts

This chapter describes the basic concepts needed to read this thesis. We include basic data compression topics such as encodings, graph compression methods and compact data structures.

1.1 Data Compression

Data compression is usually referred as to coding with the goal of using less space to represent the same data. Compressing data is relevant for data storage and data transmission since using less space improves resource utilization. Using less resources, such as storage, has the important consequence of improving the overall processing time, since it may be possible to store data at a higher level (and then faster) in the memory hierarchy.

Compression methods are usually evaluated in terms of Information theory concepts, such the entropy, which quantifies the amount of information in a given sequence. The information theory of Shannon [107] measures the amount of information in terms of bits, and more precisely provides the means of measuring the compressibility of a given object. For instance, if a given text sequence has redundancy, then it is possible to compress it by transforming it to a shorter sequence without losing information. Thus, the same amount of information can be represented with fewer bits. In other words, the compressed representation has more information per bit, so it is more unpredictable because it has less redundancy, and therefore it has higher entropy than the original text.

Shannon's definition of entropy, is $H(X) = \sum_{x \in X} P(x) \log \frac{1}{P(x)}$, where X is a discrete random variable with possible values $x_1, x_2, ..., x_n$ and $P(x)$ is its probability. When taken from a finite input sequence the entropy can be measured in terms of frequencies instead of probabilities. Furthermore, it has been shown that it is possible to find redundancy depending on the *context* where symbols appear in the input sequence. We refer as *context* of an input symbol x the fixed-length sequence of input symbols that precedes x. When the context has a length of zero we refer to *zero-order* empirical entropy which is defined as $H_0(S) = \sum_{c \in \sum} \frac{n_c}{n} \log \frac{n}{n_c}$, where n_c is the number of occurrences of c in sequence S with an alphabet

Σ. However, if we consider the context being of a length greater than zero, such as k, for some sequences it is possible to obtain greater compression. In this case, we speak of *k-order* entropy and it is defined as $H_k(S) = \sum_{c \in \sum^k} \frac{S^c}{n} H_0(S^c)$, where S^c is the sequence of symbols preceded by the context c in S. It can be proved that $H_k(S) \leq H_{k-1}(S) \leq ... \leq H_0(S)$.

1.2 Encodings

Data Compression uses different kinds of redundancy in the input sequence to encode it into a sequence that uses fewer bits. Some methods focus on finding regularities that are repeated in the input sequence. Then, new symbols are used to represent repeated patterns in the sequence. These methods are called "dictionary-based" such as Re-Pair [82].

Many compression techniques include variable-length encodings such as Huffman [71], which generates prefix-free codes, that is, the bit string representing a symbol is never a prefix of the string representing another symbol. The way that Huffman encoding works is that is based on the frequency of symbols in the input sequence, encoding those that occur more frequently with fewer bits. The compressed representation using these encodings needs both the sequence represented by the new encoding plus a vocabulary of the symbols to retrieve the original sequence. One disadvantage of variable-length encoding is that they do not allow direct access to a specific code.

There are other variable-length encodings that are more suitable for applications encoding numbers where the smaller values are more frequent. In such cases it is better to encode shorter symbols with shorter codes. Some of these encoding methods are **Unary codes**, **Gamma codes**, **Delta codes**, **Rice codes**, and **Golomb codes**. We briefly describe these codes, not because we use them directly, but because they are part of some of the off-the-shelf compression schemes we use.

- **Unary Codes**: The unary encoding represents an integer x by $1^{x-1}0$, where the 0 allows recognizing the end of a code. For instance, if $x = 4$, its unary encoding is 1110. In general, this encoding is used within other encodings.
- **Gamma (γ) Codes**: The γ encoding represents an integer x by representing the binary length of x in unary and concatenating its binary representation without the most significant bit. For instance, if $x = 4$, its γ code is 11000. The representation of an integer x uses $2 \log x + 1$ bits.
- **Delta (δ) Codes**: The δ encoding is an extension of the γ encoding for larger integers. It is basically the same as the γ encoding, but they represent the binary length of an integer x using γ-codes instead of unary codes.
- **Rice Codes**: Rice codes are parameterized codes that receive two values, the integer x and a parameter b. Then x is represented as $q = \lfloor \frac{x-1}{2^b} \rfloor$ in unary concatenated with $r = x - q2^b - 1$ in binary using b bits, for a total of $\lfloor \frac{x-1}{2^b} + b \rfloor$ bits.
- **Golomb Codes**: Golomb codes are basically the same as Rice codes, except that Rice codes define the tunable parameter as a power of 2 (i.e., 2^b), while Golomb codes are more general, avoiding the limitation for the tunable parameter to be a power of 2.

The consequence is that Rice codes produce simpler codes, but possibly suboptimal.

1.3 Graph Compression

This section describes related work on graph compression. They are either part of our compressed representation proposal or we use them to compare our compressed representations. In particular, we focus on massive graphs such as Web and social graphs.

Let us consider a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. For directed graphs an out-neighbor (or succesor) of a vertex $u \in V$ is any vertex $v \in V$, such that there is an edge from vertex u to vertex v. Similarly, an in-neighbor (or predecessor) of $u \in V$ is any vertex $v \in V$, such there is an edge from vertex v to u. In the context of Web graphs, vertices represent web pages and directed edges represent the hyperlinks. Social networks, on the other hand, can be represented by directed or undirected graphs depending on what relationship is modeled in the graph. For instance, when modeling the friendship relationship of Facebook, vertices are users and edges are represented by the friendship relation. In this case, the graph is undirected. On the other hand, the Twitter graph might be modeled by a directed graph where vertices are users and the relation of one user following another is an edge.

Compressing Web graphs has been an active research area for some time. Some of the Web graph properties most exploited for compression include *Power law distribution*, *Similarity of adjacency lists*, and *Locality*. All works measure Web compression using the *bpe* metric, which refers to the number of bits required per edge. The *bpe* corresponds to the total number of bits used to represent the graph divided by the total number of edges.

- Heavy-tail distribution. In/out-neighbors distributions are heavy tail [87], as opposed to some original works claiming power-law distributions [26]. This still can be exploited to obtain reduced-space representations.
- Locality. This property refers to the idea that most of the hyperlinks from a site point within the same site, including the directory hierarchy in the site. Then, it is possible to use node ordering, such URL, that makes most out-neighbors be close to each other. This helps to use simple encodings, like gap encoding, for reducing the amount of bits needed to represent adjacency lists.
- Similarity of adjacency lists. This property is related to the fact that many pages share a high fraction of hyperlinks with others. Then, it is possible to compress them by referring to a similar list and encoding the edits.

The first work that exploits locality and similarity is the Connectivity server [14]. It is important to note that the focus on the Connectivity server was not compression, but in fact it was to provide linkage information for all indexed pages in a search engine.

Suel and Yuan [110] built a tool for Web graph compression distinguishing global links (pages on different hosts) from local ones (pages on the same host) and combining different coding techniques, such as Huffman and Golomb codes. Adler et al. [3] achieved compression

by using similarity of adjacency lists. Their work is based on a Web graph model proposed by Kumar [79], based on the following idea *"A new page adds links by picking an existing page, and copying some links from that page to itself"*. Adler et al. exploit the idea by coding an adjacency list by referring to an already coded adjacency list of another node that points to many of the same pages. They used this property with Huffman coding to achieve compression of global links. The LINK database by Randall et al. [100] proposed lexicographic ordering of URLs as a way to exploit locality and similarity of (nearby) adjacency lists for compressing Web graphs. They achieve a bpe around 6.

Raghavan and García Molina [98] achieve about 5 bpe and define a compact structure based on dense supernodes and superedges. Such supernodes consist of a set of connected nodes in the Web graph. They also introduce the idea of *reference encoding*, which is also used later by Boldi and Vigna [19], where they represent an adjacency list based on the similarity found with a prototype adjacency list.

This section describes in a more detailed way some Web and social graph compression schemes because, either we use them as part of our compressed representation, or we use it to compare our results. We only consider those which are the state-of-the-art in compression.

1.3.1 The Webgraph Framework (Boldi and Vigna)

Boldi and Vigna [19] proposed the WebGraph framework. This approach exploits power-law distribution of gaps between adjacent successors, similarity, and locality of adjacency lists using URL node ordering. Then, using such ordering, many edges will probably have small differences, $|x-y|$, where (x, y) is an edge. This feature helps to define an encoding based on gaps. Besides, URLs that are close in lexicographic order are likely to have similar adjacency lists (as they probably belong to the same sites). Therefore, they encode adjacency lists using references to previous adjacency lists, based on their similarity. Let r be an integer (reference), if $r > 0$, the list x is described as a difference from the list of $x - r$. If $r = 0$ the list is not compressed by using a reference. A bit string is used to tell which successors must or must not be copied. A "1" in the bit string means that the node is present en the reference list. These bit strings are called copy lists. Copy lists are encoded by copy blocks, where the first block is "0" if the copy list starts with a 0. The block is represented by the length of "1"s or "0"s in the copy list decremented by 1 except the first block, and the last block is omitted. Another list is used for representing the remaining nodes in the list. The value of r is limited by a window size w. A large w provides better compression, but higher memory usage and more running time for compressing/decompressing.

Finally, Boldi and Vigna also exploit the fact that remaining nodes are often consecutive, so instead of directly using gap encoding they isolate subsequences and represent them with intervals (considering an interval threshold, which is at least a minimum number of integers in the interval, L_{min}). Each interval is represented by the left extreme and the length (number of integers it contains). Left extremes are compressed using differences between each left extreme and previous right extreme minus 2. The rest of the remaining nodes are compressed using differences. Table 1.1 shows an example of using copy lists, Table 1.2 using copy blocks and Table 1.3 includes an interval reference for remaining nodes.

Node	Outdegree	Ref. (r)	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Table 1.1: Representation of plain copy lists.

Node	Outdegree	Ref. (r)	# blocks	Copy block	Extra nodes
...	...	...	...	...	...
15	11	0			13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	7	0,0,2,1,1,0,0	22, 316, 317, 3041
17	0				
18	5	3	1	4	50
...	...	...	...	...	...

Table 1.2: Representation of copy blocks and extra nodes.

The main parameters of this compression technique are w, L_{min}, and m, where w is the window size, L_{min} is the minimum interval length for representing intervals, and m is the maximum reference count. The window size means that the list l_i can only be expressed as a near-copy of l_{i-w} to l_{i-1}, whereas the reference count of list l_i is $r(l_i) = 0$ if it is not expressed as a near-copy of another list, or $r(l_i) = r(l_j) + 1$ if l_i is encoded as a near-copy of list l_j. Increasing w and m improves compression ratio, but also increases access time.

In a later work, Boldi et al. [18] explored existing and novel node ordering methods, such as URL, lexicographic, Gray ordering, etc. Boldi et al. [16], more recently, designed node orderings based on clustering methods, and achieved improvements on compressing Web graphs and social networks with a clustering algorithm called Layered Label Propagation (LLP). They apply the algorithm Absolute Pott Model (APM) [102], which is a variant of a label propagation algorithm, to avoid having large clusters that tend to capture a large number of nodes. They proposed the LLP algorithm which also considers how to label nodes belonging to the same and different clusters.

1.3.2 BFS Ordering (Apostolico and Drovandi)

A different and very competitive compression technique was proposed by Apostolico and Drovandi [9]. Their approach uses Breath-First Traversal (BFS) node ordering to improve compression. Although this is the default node ordering used, it is possible to apply the compression scheme using the node ordering given by the input graph (option -s).

Node	Outdegree	Ref. (r)	# blocks	Copy list	# intervals	Left extremes	Length	Residuals
...	...	...	...	...	...	...	...	...
15	11	0			2	0,2	3,0	5,189,111,718
16	10	1	7	0,0,2,1,1,0,0	1	600	9	12, 3018
17	0							
18	5	3	1	4	0			50
...	...	...	...	...	...	...	...	...

Table 1.3: Representation of copy blocks, intervals and residuals.

Node	Outdegree	Adjacency lists
...	...	...
i	8	13, 15, 16, 17, 20, 21, 23, 24
$i+1$	9	13, 15, 16, 17, 19, 20, 25, 31, 32
$i+2$	0	
$i+3$	2	15, 16
...	...	...

Table 1.4: Adjacency lists. It is assumed that v_i is the first node of the chunk.

Node	Outdegree	Adjacency lists
...	...	...
i	8	$\phi 13$ $\phi 1$ $\phi 0$ $\phi 0$ $\phi 2$ $\phi 0$ $\phi 1$ $\phi 0$
$i+1$	9	$\beta 0$ $\beta 0$ $\beta 0$ $\chi 0$ $\alpha 0$ $\beta 2$ $\phi 5$ $\phi 0$
$i+2$	0	
$i+3$	2	$\beta 2$ $\alpha 0$
...	...	...

Table 1.5: Encoding of adjacency list.

They propose a scheme that uses consecutive integer numbers for indexing each node in the traversal tree and build a traversal list containing the out-degree of each of the nodes of the tree. Then, they compress the graph using the traversal list plus all the edges not present in the BFS tree using the indices defined by the BFS traversal.

They encode the adjacency lists of nodes in increasing order by chunks of length l. Parameter l (called the level) provides a trade-off between compression performance and time to retrieve the adjacency list of a node. Each encoding consists of the integer gap between adjacent elements and a type indicator (ϕ, β, α, χ) based on the following cases:

1. $A_{i-1}^j \leq A_i^j < A_i^j$: the code is the string $\phi \cdot (A_i^j - A_i^{j-1} - 1)$
2. $A_i^{j-1} < A_i^j \leq A_i^j$: the code is the string $\beta \cdot (A_i^j - A_{i-1}^j)$
3. $A_i^{j-1} < A_i^j < A_{i-1}^j$: distinguishing two subcases:
   (a) if $A_i^j - A_i^{j-1} - 1 \leq A_{i-1}^j - A_i^j - 1$ then the code is the string $\alpha \cdot (A_i^j - A_i^{j-1} - 1)$
   (b) otherwise the code is the string $\chi \cdot (A_{i-1}^j - A_i^j - 1)$

The types α and ϕ encode the gap of the consecutive elements in the adjacency list (A_i^{j-1}) whereas the types β and χ encode the gaps given with an adjacency element in the same position of the adjacency list of the previous node (A_{i-1}^j). Table 1.4 shows an example with adjacency lists and Table 1.5 shows its encoding using these types of special characters.

Using BFS node ordering enables to encode two nodes connected by a link with close index values, which helps improve compression ratio. In fact, Chierichetti et al. [33] showed that finding an optimal node index assignment that minimizes $\sum_{(v_i,v_j)\in E} \log|i-j|$ is NP-hard. In addition, since Web graphs usually share a high fraction of neighbors, the adjacency lists consist of consecutive integer lists. These features allows them to use different types of encodings to exploit distinct types of redundancies (shown in Table 1.6). The redundancies

Outdegree	Adjacency lists
...	...
9	$\beta 7$ $\phi 1$ $\phi 1$ $\phi 1$ $\phi 0$ $\phi 1$ $\phi 1$ $\phi 1$ $\phi 1$
9	$\beta 0$ $\beta 1$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 2$
10	$\beta 0$ $\beta 1$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 1$ $\phi 903$
10	$\beta 0$ $\beta 1$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 223$ $\phi 900$
10	$\beta 0$ $\beta 1$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 1$ $\alpha 0$
10	$\beta 0$ $\beta 1$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 1$ $\beta 0$
10	$\beta 0$ $\beta 1$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 1$ $\beta 0$
10	$\beta 0$ $\beta 1$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 0$ $\beta 1$ $\beta 0$
...	...

Table 1.6: Exploiting redundancies in adjacency lists.

include intervals of constant node degrees, identical elements in a sequence of at least L_{min} (as the block of ϕ 1's in the Table 1.6), and identical rows. They exploit these redundancies using run-length encoding for representing identical adjacency lists, or adjacency lists with a common sequence of identical elements; and gap-encoding for intervals of constant node degrees. They add additional encodings for identifying these types of redundancies, adding two special characters, Σ, and Σ_F, where Σ_F identifies the type of redundancy. Depending on the redundancy the encoding is expressed as "*type* $\Sigma\Sigma_F$ *gap* l", or "*type* $\Sigma\Sigma_F$ *gap* l w h", where *type* is any of the special characters α, β, χ, or ϕ, Σ_F is a integer value to identify the redundancy, *gap* is the integer gap, l is the number of identical elements on the same line, w and h are the width and height of the identical columns and rows.

Finally, they use Huffman codes to encode α,β,χ, and Σ_F and propose a new encoding, π-code, for representing gaps, Σ (an integer that does not appear in a gap), and other integers.

1.3.3 Re-Pair (Claude and Navarro)

Another approach, proposed by Claude and Navarro [42], provides Web compression by applying Re-Pair [82], a grammar-based compression scheme. The algorithm is based on finding the most frequent pairs of symbols in a sequence. Then, they replace such pairs by a new symbol which is added to a dictionary with the pair that represents, as a new rule. The algorithm iterates until no more replacements are convenient. Since exact Re-Pair requires too much memory over large sequences, Claude and Navarro provide an approximate algorithm. The approximate algorithm consists of applying Re-Pair using limited memory in addition to the sequence $T(G)$, where $T(G)$ is the concatenation of all adjacency lists of a Web graph. In addition, since the algorithm uses sequential access patterns, it is well adapted to secondary memory. Each adjacency list of vertex v_i in $T(G)$ is defined as $T(v_i) = -v_i v_{i_1} v_{i_2} ... v_{i_r}$, where v_i is the vertex id and $v_{i_1} v_{i_2} ... v_{i_r}$ are the out-neighbors of vertex v_i. The id of the vertex is never considered in any possible candidate pair for replacement, and it is only included in $T(G)$ as a mark for separating adjacency lists.

The compression achieved by Claude and Navarro [42] provides relevant space/time trade-offs compared with WebGraph [19]. They provide faster navigation using the same space, but

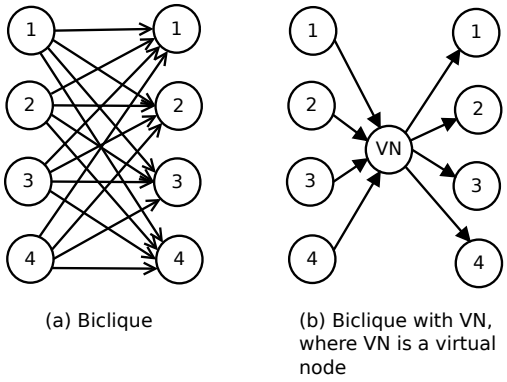


Figure 1.1: Biclique example

WebGraph can use less space if slower navigation is allowed. However, later improvements of WebGraph, which are based on using other node orderings, such as Gray-ordering [18], and clustering ordering based in LLP [16] improve compression without affecting navigation speeds, then overcoming Re-Pair [42].

Another representation based on Re-Pair is able to solve out/in-neighbors [41]. This scheme was obtained by combining the Re-Pair-based representation [42] with compact sequence representations [40] of the resulting adjacency lists.

1.3.4 Virtual Node Mining (Buehrer and Chellapilla)

Buehrer and Chellapilla [29] exploited the existence of many groups consisting of sets of pages that share the same out-neighbors, which defines complete bipartite graphs (bicliques). A complete bipartite graph is defined as $G(S \cup C, E)$, such that for any two vertices, $s \in S$ and $c \in C$, (s, c) is an edge in G. In other words, a biclique is a graph consisting of two sets (S and C), where all vertices in set S are connected to all vertices in set C. Their approach is based on reducing the number of edges by defining virtual nodes that are artificially added in the graph to connect the two sets in a biclique. The goal of the approach is to reduce the number of edges represented in the whole graph, since each of the bicliques can reduce the number of edges from $|S \times C|$ to $|S + C|$. In order to discover bicliques they use a clustering and a frequent itemset mining algorithms. They apply this process iteratively on the graph until the edge reduction gain is no longer significant. Then, they apply delta codes on the edge reduced graph. This compression scheme allowed Buehrer and Chellapilla to achieve sizes between 1.5 and 3 bpe on Web graphs. However, they did not report navigation times. An example of a biclique and its representation using a virtual node is seen in Figure 1.1.

This compression scheme is relevant not only for compressing Web graphs, but also in the context of community discovery. Moreover, the algorithm proposed does not depend on the node ordering used, it is scalable and allows for incremental updates. Anh and Moffat [8] also exploit similarity and locality of adjacency lists, but they divide the lists into groups

of h consecutive lists. A *model* for a group is built as a union of the group lists. They reduced lists by replacing consecutive sequences in all h lists by a new symbol. The process can be made recursive by applying it to the n/h representative lists. They finally applied codes such as ς-codes [20] over all lists. This approach is somehow similar to that of Buehrer and Chellapilla [29], but Anh and Moffat [8] do not specify how they actually detect similar consecutive lists.

1.3.5 Grabowski and Bieniecki

Grabowski and Bieniecki [66] recently provide a very compact and fast technique for Web graphs. Their algorithms are based on blocks consisting of multiple adjacency lists in a similar way to the scheme of Anh and Moffat [8], reducing edge redundancy. Their approach consists of having blocks of h adjacency lists. Each block is converted into two streams: the first stream stores a *long list* of all integers on the h input lists, without duplicates, and the second stream stores flags that allow reconstructing the original lists. The *long list* is compacted using differential encoding, zero-terminated and encoded using a byte code. They use a byte code with 1, 2 and b bytes per codeword. The byte coder consists of using 2 bits in the first codeword byte, which tell the length of the current codeword.

The flag stream describes to which input lists a given integer on the output list belongs; where the number of bits per item on the output list is h, which is defined to be a multiple of 8. The length of the flag stream is defined by the length of the *long list*. The flag stream can be left raw (with no encoding, which they called *LM-bitmap*) or it can be encoded using gaps between the successive 1s in the flag sequence, which are written on individual bytes (*LM-diff*). Finally, the two streams are concatenated and compressed with the Deflate algorithm. The Deflate algorithm consists of a series of blocks, each preceded by 3-bit header. The first bit indicates if it is the last block in the stream (1) or not (0). The other 2 bits identify a literal section (00), a static Huffman compressed block, using a pre-agreed Huffman tree (01), a compressed block with the Huffman table supplied (10) and not used (11).

The compression parameter of the approach is the block size, h. Using a larger h exploits a wider range of similar lists, but the flag stream gets sparser and the Deflate algorithm does not behave well on that kind of data. In addition, decoding larger blocks takes longer. Overall they achieve bpes between 1 and 2.

1.3.6 K2-tree (Brisaboa, Ladra and Navarro)

Most of the Web graph compression schemes (as the ones described above) support out-neighbor queries, that is, the list of nodes pointed from a given node, just as an adjacency list. Being able to solve in-neighbor queries (i.e., the list of nodes pointing to a given node) is interesting for many applications from random sampling of graphs to various types of mining and structure discovery activities, as mentioned in the Introduction. It is also interesting in order to represent undirected graphs without having to store each edge twice.

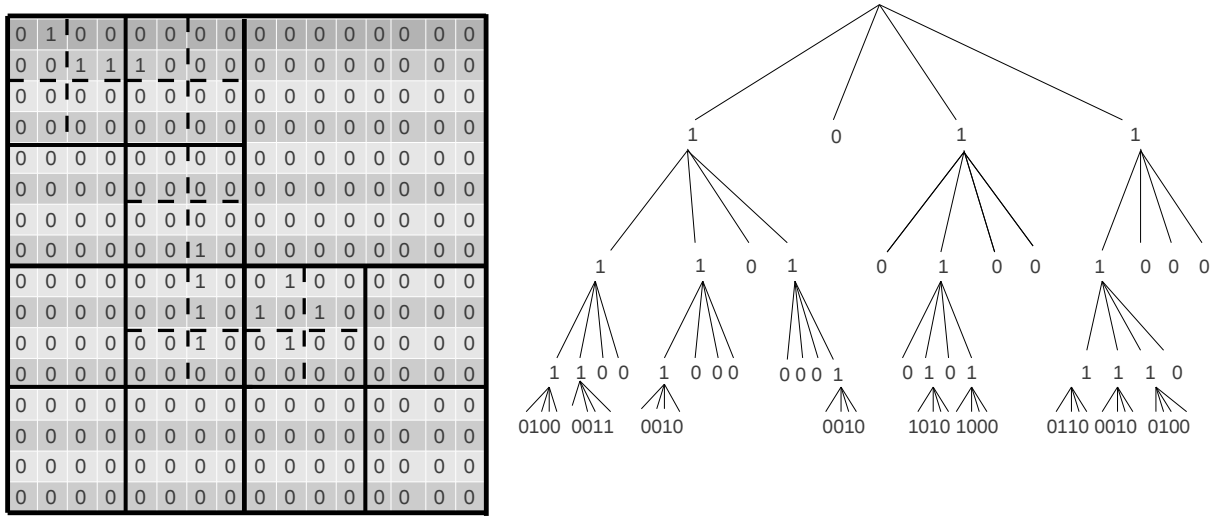


Figure 1.2: K2tree representation

Brisaboa et al. [22] exploited the sparseness and clustering of the adjacency matrix to reduce space while providing out/in-neighbor navigation in a natural symmetric form, using a structure called k2tree. They have recently improved their results by applying BFS node ordering on the graph before building the k2tree [23] This achieves the best known space/time tradeoffs supporting out/in-neighbor access for Web graphs. The k2tree scheme represents the adjacency matrix by a k^2-ary tree of height $h = \lceil \log_k n \rceil$ (where n is the number of vertices). It divides the adjacency matrix into k^2 submatrices of size n^2/k^2. Completely empty subzones are represented just with a 0-bit, whereas nonempty subzones are marked with a 1-bit and recursively subdivided. The leaf nodes contain the actual bits of the adjacency matrix, in compressed form. Supporting out- and in-neighbor queries in k2tree is symmetric, as it involves finding the points in a row or column of the matrix.

Figure 1.2 shows an example of k2tree with $k = 2$. As observed, each node contains a bit 1 in the internal nodes and a 0 in a leaf node, with the exception on the last level of the tree, where all nodes are leaves that represent a 0 or a 1 in the adjacency matrix of the graph. Level 0 of the tree represents the root, who has k^2 children at level 1. As a child each node in level 1 has a 1 or a 0. All internal nodes with a 1 have k^2 children, and nodes with a 0 represent leaves and then have no children. The tree keeps growing up to the last level, where the tree represents individual cells in the adjacency matrix.

Once the k2tree structure is determined, the compression is achieved by representing the tree structure compactly, by using two bit arrays: A bitmap T for the tree structure and a bitmap L for representing leaves at the last level, which represent actual cells in the adjacency matrix. The authors use an additional bitmap that helps them compute queries faster. In the end, the final representation of a Web graph based on its adjacency matrix consists of the concatenation of the two bit arrays, $T : L$, and the extra structure to support rank

operations over T efficiently.

K2trees provide Web graph compression between 1 and 3 bpe and support in/out-neighbor queries. Navigation times are slower than those provided by compression techniques that support only out-neighbor navigation like the lastest WebGraph [16] and the proposal by Apostolico and Drovandi [9].

Recently, Claude and Ladra [39] improved the compression performance on Web graphs by combining the k2tree with the Re-Pair-based representation [42]. They called the approach *k2-Partitioned*. The structure exploits the similarity of adjacency lists by splitting the graph into subgraphs formed by groups of intra-domains, which happen to occur around the adjacency matrix diagonal. Extra-domain links are captured by all the edges where some subgraphs point to others. The scheme provides good compression because k2tree compresses the subgraphs (which are dense) very well and the extra-domain part of the graph is much sparser. The access speed is also high since the height of the trees are smaller than when applying k2tree on the complete graph.

1.4 Compressing Social Networks

Some recent works on compressing social networks [33, 85] have unveiled compression opportunities as well, although to a lesser degree than on Web graphs. The approach by Chierichetti et al. [33] is based on the Webgraph framework [19], using shingling ordering (based on Jaccard coefficient) [26, 62] and exploiting link reciprocity. Shingles were introduced by Broder et al. [24] and have been used to estimate the similarity of Web pages using a technique based on overlapping windows of terms. The shingling technique has been applied to generate a number of constant-size fingerprints for two subsets A and B from a set S of a universe U of elements, such that the similarity of A and B can be computed easily by comparing fingerprints of A and B. In the context of graphs, the idea is to obtain a fingerprint of the out-neighbors of a node and ordering the nodes according to this fingerprint. Broder et al. [24] show that the probability that the shingles of A and B are identical is the same as the *Jaccard coefficient* $J(A, B) = |A \cap B|/A \cup B|$, which captures the notion of similarity of sets A and B. They called a *shingle* the smallest element in A according to a random permutation.

In addition, they show that, instead of using random permutations, it is enough to use min-wise independent hash families. Shingles have been used in graph algorithms for detecting similar out-neighbors, using a set of hash functions. The idea is that if nodes share out-neighbors then with high probability they will have the same shingle and hence be close to each other in a shingle-based ordering. Even though they achieve interesting compression for social networks, their approach requires decompressing the graph in order to retrieve the out-neighbors.

Chierichetti et al. [33] state that there is no obvious node ordering for compressing social networks. They showed that using "Shingles ordering" provides some compression. Boldi et al. [17] studied the effect of useful permutation strategies to compress social networks,

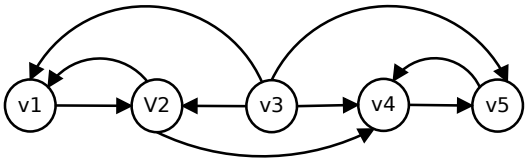
where they proposed two node orderings based on the Gray ordering that provide better compression results. One node orderings is loose host-by-host Gray ordering, which is based on keeping URLs from the same host adjacent, and order them using Gray ordering. The other one is strict host-by-host Gray ordering and it is is like the loose ordering, but Gray ordering is applied considering only local links, i.e., links to URLs of the same host. Later, Boldi et al. [16] showed that using LLP node ordering on Web and social graphs improve further compression efficiency. However, these schemes achieve compression results that are not nearly as good as on Web graphs.

Maserrat and Pei [85] achieve compression by defining an Eulerian data structure using multi-position linearization of directed graphs. This scheme is based on decomposing the graph into small dense subgraphs and supports out/in-neighbor queries in sublinear time. Masserrat and Pei work on the idea that social networks are locally dense and globally sparse. For doing that, they propose a linearization scheme that allows them to represent the graph by using a sequence. Basically, they define an MPk linearization of a graph $G(V, E)$, with $n = |V|$ and $m = |E|$. An MPk linearization consists of a sequence S (where each element consists of a vertex identifier and a pointer) that satisfies the following property: For any edge $(u, v) \in E$, there exists a subsequence W of length k in S, such that u and v appear in W. In this sequence vertex identifiers can appear more than once.

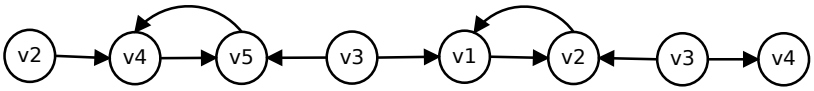
They show that, if they use $2K$ bits per each element in the sequence S, it is possible to represent the graph G. The $2k$ bits are used to mark which ones of the $2k$ neighbors (from a position in S) correspond to an edge. Pointers are used to find the next appearance of the vertex identifier in the sequence. They show that finding the minimum MPk linearization is NP-hard and then provide a heuristic for computing an MPk linearization. Figure 1.3 shows an example with a linearization MP1, where Figure 1.3-(a) shows the traditional representation, Figure 1.3-(b) shows MP1 linearization, Figure 1.3-(c) shows the compressed representation by Maserrat and Pei [85], with MP1, where each node is represented with 2 bits and a pointer to the next position where a node appears in the line.

The heuristic can be summarized as follows: First, add a random vertex to the sequence. Then, find the vertex that has more edges to the last k vertexes in the sequence, add it to the sequence and remove the edges. Repeat the process until there is no edge left in the graph.

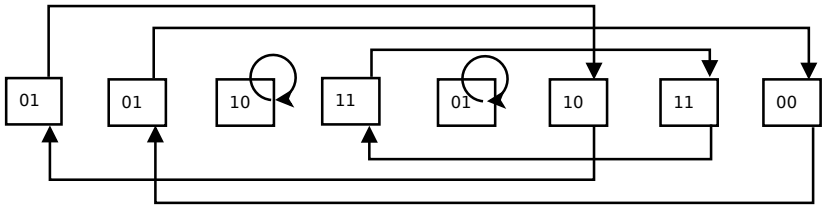
Later, Claude and Ladra [39] proposed a new indexing MPk for linearization that allows them to improve the access time for in/out-neighbors. Their basic idea is to use the same representation used by Maserrat and Pei, but they replace pointers to the next occurrences by an index for sequences supporting rank, select and access operations. They define two sequences N and S. Each element in sequence N has $2k$ bits representing if the current vertex index (index in the sequence) has an edge to the $2k$ neighbors. Sequence S contains the vertex identifiers in the linearization, which is used instead of having pointers to find the next appearance of a vertex identifier. Figure 1.3-(d) shows the sequences N and S. Retrieving out/in-neighbors requires the operations rank/select/access over these sequences. For instance, to be able to recover all occurrences of a vertex identifier in the sequence they compute $rank_S(v, |S|)$ and then for each occurrence j they obtain the position in S of the vertex identifier by computing $select_S(v, j)$.



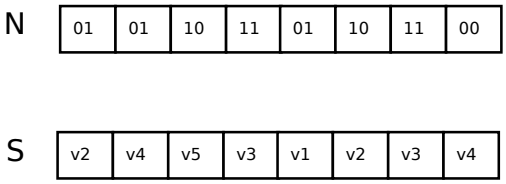
(a) Original Graph



(b) Mp1



(c) Compressed representation (Maserrat and Pei, 2010)



(d) Compressed representation (Claude and Ladra, 2011)

Figure 1.3: An example of MP1 (k=1).

1.5 Compact Data Structures: Sequences

This section describes compact data structures based on sequences. In this case, we distinguish binary sequences from symbol sequences. The first ones are also known as bitmaps, since they only contain symbols that are "0"s and "1"s, while the others can store arbitrary symbols from an alphabet Σ of size σ. The basic operations, sometimes called queries, over sequences are *Rank*, *Select*, and *Access*. $Rank_S(a, i)$ counts the number of occurrences of symbol a up to the position i in the sequence S. $Select_S(a, i)$ finds the position of the i-th occurrence of a in the sequence S. Finally, $Access_S(i)$ returns the symbol at position i in the sequence S. This problem has been addressed in theory and practice, and there exist several implementations.

1.5.1 Binary Sequences

Consider a binary sequence or bitmap $B[1, n]$ with m ones. Several schemes have been proposed to efficiently compute *rank* and *select* operations on bitmaps. Jacobson [72] developed succinct data structures using *rank* and *select* operations. He proposed an implementation to compute *rank* in $O(1)$ time and use it to implement binary trees. However, computing *select* takes $O(\log \log n)$ time, since the scheme is based on binary search lookups.

The idea for computing *rank* in constant time consists of building a structure based on a two-level directory. The first level divides the binary sequence into *superblocks* of size $s = (\log n)^2/2$ bits and defines an array $S[1, n/s]$, where $S[i]$ stores the number of 1s observed up to the beginning of the superblock. Therefore, S contains $\frac{n}{(\log n)^2/2}$ superblocks, where each $S[i]$ requires $\log n$ bits, therefore S requires $2n/\log n$ bits, which is $o(n)$. The second level divides the binary sequence on *blocks* of size $b = (\log n)/2$ and defines an array $R[1, n/b]$, where $R[i]$ stores the *rank* values up to the beginning of each corresponding superblock, that is $R[j] = rank(B, jb) - S[j/\log n]$. Since the values in R are relative to a corresponding superblock, the maximum value that can be stored in R is $O(\log n)^2$, therefore they only require $2 \log \log n$ bits. Overall, the total space required by R is $\frac{n}{(\log n)/2} 2 \log \log n = \frac{4n \log \log n}{\log n} = o(n)$.

In addition, the structure builds a *table* (T) for all possible sequences of $(\log n)/2$ bits. This table has $2^{\frac{\log n}{2}}$ rows and $(\log n)/2$ columns, and each cell (r, c) contains the number of 1s observed in the binary representation of row id r up to the c-th bit. The maximum value that can be stored in the table can be encoded in $\log \log n - 1$ bits. Then the table requires a space of $2^{\frac{\log n}{2}} \frac{\log n}{2} (\log \log n - 1) \leq \frac{1}{2} \sqrt{n} \log n \log \log n = o(n)$ bits. The final result consists of solving $rank(B, i) = S[j] + R[k] + T[r, c]$, for corresponding superblock j, block k, row r, and column c. The overall space is $n + o(n)$, computing *rank* in $O(1)$ time.

Later, Clark [37] and Munro [38] improved these results computing *rank* and *select* operations in $O(1)$ time using $n + o(n)$ bits, where n bits are used for storing the bitmap of size n and $o(n)$ bits are used for the data structures needed to compute *rank* and *select* operations. The main idea for improving *select* time operation consists of dividing the bitmap

into superblocks with equal amount of ones ($s = \log^2 n$), therefore the length of superblocks are variable as opposed to superblocks of equal sizes. Superblocks can be dense or sparse; it is said that a superblock is sparse if its length in B is at least $s \log n \log \log n$, otherwise is dense. If a superblock is sparse, all the answers are stored with an overhead of $O(\frac{n}{\log \log n})$. The whole process for computing *select* is more complex than *rank*, but the idea is to identify dense and sparse superblocks, storing all answers for sparse superblocks and apply a second level of indirection on dense *superblocks* using *blocks*. In the second level, dense and sparse blocks are identified, and dense blocks also use tables as the ones described above for ranks.

The solutions proposed by Clark and Munro [37, 38] were improved by Pagh [95], and Raman et al. [99] achieving $nH_0(B) + o(n)$ bits, and keeping query time constant, by representing compressed binary sequences. Pagh [95] proposed using compressed blocks of the same size, representing the number of 1s in each block and the number that identifies each block. Raman et al. [99] provide the same solution for *rank* than Pagh [95], but they achieve *select* in constant time by using perfect hashing.

Therefore, each block is represented as a tuple (c_i, o_i), where c_i represents the class of the block and o_i represents the offset of the block in a list of all possible blocks in class c_i. If the block size is b then the number of bits required to represent c_i is given by $\lceil \log(b+1) \rceil$ and each o_i is represented by using $\lceil \log \binom{b}{c_i} \rceil$ bits. The c_i are of fixed length and require $O(\frac{n \log \log n / \log n}{\log n})$ bits of space, while the o_i are of variable length. The implementation uses three tables E, R and S. Table E stores all possible combinations of b bits sorted by class and all answers for rank at every position of each combination. Table R stores the concatenation of all the c_i, using $\lceil \log(b+1) \rceil$ bits per field, and table S stores the concatenation of all o_i using $\lceil \log \binom{b}{c_i} \rceil$ bits per field. It can be shown that the total space for S is $nH_0(B)+O(n/\log n)$ bits. Overall, the structure requires $nH_0(B)+o(n)$ bits. Additional data structures used for answering *rank* and *select* do not increase the $o(n)$ component. Patrascu [97] also requires $nH_0(B) + o(n)$ bits and answers *rank* and *select* operations in constant time. This approach works well when the binary sequence has many or a few 1s, and the $o(n)$ term can be $O(n/\log^c n)$ bits for any desired constant c.

Okanohara and Sadakane proposed an alternative scheme that gets rid of the $o(n)$ extra bits and achieves close results for sparse bitmaps (small m). They present four rank/select directories: *esp*, *rerank*, *vcode*, and *sdarray*. Each of them is based on different ideas and has different trade-offs in terms of speed and space. In particular, most of them are very good for *select* operations, but *ranks* are slower. The best alternative is the *sdarray* directory. In this work, we have used practical implementations developed by González and Navarro [64] following the ideas of Jacobson and Munro [37, 38], and the implementation for compressed bitmaps developed by Claude and Navarro [40], based on the proposal of Raman et al. [99].

1.5.2 Arbitrary Sequences

As mentioned earlier, arbitrary sequences store symbols from an alphabet Σ of size σ, where $\sigma > 2$. This section describes the implementations we use in this work.

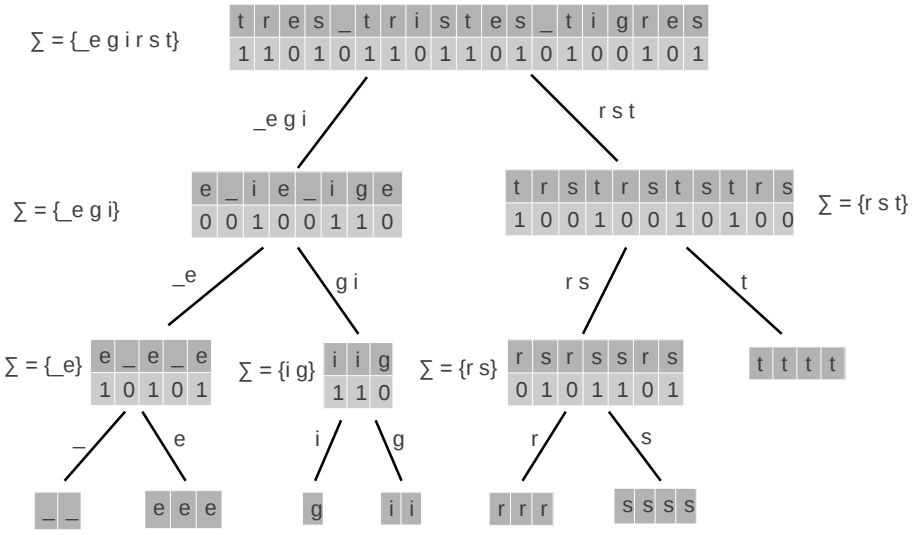


Figure 1.4: A wavelet tree example.

Wavelet Trees

A wavelet tree [67] consists of a binary tree, where the alphabet is recursively subdivided. The root and internal nodes store bitmaps. At each node of the tree, the bitmaps represent the symbols in the sequence S that belong to the corresponding subalphabet. If the symbol of the sequence belongs to the first half of the subalphabet, it is represented with a "0" in the bitmap and handled at the left child of the node. Otherwise it is represented with a "1" and handled at the right child.

Given a sequence S of length $n = |S|$, where $S[i] \in \Sigma$, and $\sigma = |\Sigma|$, the construction of the wavelet tree consists of the following steps: The root of the tree consists of a bitmap B, where $B[i] = 0$ if $S[i] \in [0, \frac{\sigma}{2}]$ and $B[i] = 1$, if $S[i] \in [\frac{\sigma}{2} + 1, \sigma]$. The next level of the tree is built based on the symbols associated to the node by parent bitmap B. In this case, if some $B[i] = 0$, the left child bitmap is created, and the alphabet corresponding to such symbols are again divided by two, defining a bit "0" for the first half and a "1" for the second half. Similarly, the right child in the tree considers the symbols in $S[i]$ for which the bitmap at the root stores a "1", creating the corresponding bitmap associated to the right child. The same process continues dividing the alphabet in half at each node in the tree. Finally, the leaf nodes represent individual symbols. The complete representation consists of all the bitmaps of the tree, with the exception of the leaf nodes, which are not represented. Note that the tree requires the pointers to left and right children, which provide the navigation of the tree. The complete sequence can be recovered by only accessing the bitmaps. Figure 1.4 shows an example, including text for clarity although it is not really represented in the structure.

As mentioned earlier, the three basic query operations over sequences are rank, select and access. The wavelet tree enables these operations by traversing the tree. Access and rank queries involves performing a top-bottom traversal of the tree and performing ranks on the

bitmaps. Select, on the other hand is done by a bottom-up traversal over the tree performing selects on the bitmaps.

An access operation, $S[i] = Access_S(i)$, is done as follows. First, see if $B[i]$ at the root is zero, if so, go to the left child and compute a new position j with $j = rank_B(0, i)$ (computing the number of occurrences of a 0 in B up to position i). In the case $B[i]$ is a "1", go to the right child and compute a new position j with $j = rank_B(1, i)$ (computing the number of occurrences of a 1 in B up to position i). Using the new position j keep following down the tree up to the leaf, where it is possible to know the symbol in $S[i]$. The time is bound to $\log \sigma$ evaluations of bitmap ranks.

A rank operation, $rank_S(a, i)$, computes the number of occurrences of symbol a in $S[1, i]$. First, by examining the alphabet, decide if symbol a belongs to the first half of the alphabet, $a \in [0, \frac{\sigma}{2}]$, or the second, $a \in [\frac{\sigma}{2} + 1, \sigma]$. If it is in the first, go to the left child and compute the new position j as $j = rank_B(0, i)$. Otherwise, go to the right child and compute the new position j as $j = rank_B(1, i)$. Repeat the process going down on the tree up to the leaf, where the answer of the query is j. In this case, the time is also bound to $\log \sigma$ evaluations of bitmap ranks.

Finally, a select operation, $select_S(a, i)$, computes the position of the ith-occurrence of symbol a in S. We start at the leaf node that represents symbol a, which is found by the position of a in the alphabet. Determine whether the leaf is a left or a right child, if it is a left child, compute a new position j doing a select on the parent bitmap with $j = select_B(0, i)$. If is a right child, compute a new position j with $j = select_B(1, i)$. Keep going up in the tree up to the root bitmap, based on applying select on the parent for 0 or for 1 depending on whether the node is a left or a right child of its parent. In this case, the time is bound to $\log \sigma$ evaluations of bitmap selects.

Overall, the Wavelet tree supports rank/select/access queries in $O(\log \sigma)$ time. The space is bounded depending on the bitmap representation. When using plain bitmaps [64] the space is bounded to $n \log \sigma + o(n) \log \sigma$ bits. However, when using compressed bitmaps [99] the space is bounded to $nH_0(n) + o(n) \log \sigma$ bits. As our alphabets are large, in this work, we use the version *without pointers* [40], which saves an extra space of the form $O(\sigma \log n)$.

The wavelet tree without pointers is an alternative representation of wavelet trees that eliminates the pointers of the tree [40]. The representation stores $\log \sigma$ bitmaps of length n, where bitmaps at the same level are concatenated. The first bitmap corresponds to the root of the tree, the second bitmap corresponds to the concatenation of the left and right bitmaps, and so on. This representation requires calculating the interval $[s, e]$ corresponding to the bitmap of a node, and the interval $[s', e']$ upon a child or parent operation. They show how to compute the children intervals using *rank* operations on the parent interval. The extra space is reduced from $O(\sigma \log n)$ to $O(\log \sigma \log n)$ bits.

Golynski, Munro and Rao Sequences

Another representation for general sequences was proposed by Golynski, Munro and Rao [63]. Such representation uses $n \log \sigma + no(\log \sigma)$ bits and can solve access and rank queries in $O(\log \log \sigma)$ time, and select in $O(1)$; or select in $O(\log \log \sigma)$, rank in $O(\log \log \sigma \log \log \sigma)$

and access in $O(1)$ time. The representation of the sequence is done using a matrix T of size $\sigma \times n$, where rows are indexed by σ, and columns by positions in the sequence. Each cell in the matrix (i, j), indicates whether symbol i occurs in position j in the sequence. Using one bitmap A (given by rows) to represent the whole matrix T, requires $|A| = \sigma n$ bits, allowing to answer *rank* and *select* operations at the block level. Since the space required by A is too high, instead of representing bitmap A, A is divided into blocks of size σ defining a bitmap $B = 1^{k_1}01^{k_2}01^{k_3}...0$, where k_i is the number of 1s in each block. Using bitmap B instead of A, the space is reduced to $2n + o(n)$ bits. In order to allow *rank*, *select*, and *access* operations in a block, the authors use an additional structure called a *chunk*. Each *chunk* stores σ symbols of the text using a sequence and a bitmap. The sequence is a permutation π obtained by sorting alphabetically the sequence represented by the *chunk* and uses a data structure that allows constant-time computation of π^{-1} [92]. Bitmap X stores the number of times a symbol i of the alphabet appears in the *chunk* using a representation similar to bitmap B, then $X = 1^{l_1}01^{l_2}01^{l_3}...0$, where l_i corresponds to the number of times symbol i appears in the *chunk*. Overall, the total space is given by $4n + n \log \sigma + no(\log \sigma)$ bits.

Chapter 2

Discovering Graph Patterns

There is a wide variety of real systems that are modeled by graphs, such as communication, Web, social, and biological networks. Graph analysis has become crucial to understand the features of these systems. Finding groups, clusters or communities in large graphs has been a topic of study for different applications, for instance, in the context of web search, online retailers, or protein-protein interactions in a biological network. Clustering web clients to identify groups with similar interest and geographically close to each other may improve the performance of web services, where mirror servers may serve similar groups [78]. Online retailers might want to identify customers with similar interests to improve recommender systems [101]. Identifying communities is also interesting for classifying the members of groups, for example some members might have a central position in the clusters, that is, share a large number of edges with the other members. These might mean that those members are important for the stability of the group.

In this chapter we describe scalable techniques for discovering graph patterns in Web and social graphs that are of interest for data mining and analysis, as well as for representing graphs compactly. All graphs we consider are modeled with unlabeled edges. The graph patterns presented here are based on dense subgraphs such as cliques, complete bipartite graphs (bicliques) and some combinations of both. These graph patterns are the basis for the compressed representations we present in Chapter 3.

Our discovery algorithm uses clustering and mining for identifying patterns. We show that our clustering approach is much more scalable than MCL [55, 114] (a state-of-the-art clustering algorithm) even when comparing the sequential algorithm against the parallel version of MCL. We evaluate the quality and running time of the algorithm using synthetic and real graphs.

2.1 Problem Definition

We represent a Web graph as a directed graph $G = (V, E)$ where V is a set of vertices (pages) and $E \subseteq V \times V$ is a set of edges (hyperlinks). For an edge $e = (u, v)$, we call u the *source*

and v the *center* of e. In social networks, nodes are individuals (or other types of agents) and edges represent some relationship between the two nodes. These graphs can be directed or undirected. In case they are undirected, we make them directed by representing both reciprocal directed edges. Thus from now on we consider only directed graphs.

We follow the idea of "dense communities" in the Web of Kumar et al. [80] and Dourisboure et al. [56], where a community is defined as a group of pages related by a common interest. Such Web communities are characterized by dense directed bipartite subgraphs. In fact, Kumar et al. [80] summarize that a "random large enough and dense bipartite subgraph of the Web almost surely has a core (a complete bipartite subgraph)", which they aim to detect. Left sets of dense subgraphs are called *Fans* and right sets are called *Centers*. In this work, we call the sets *Sources* (S) and *Centers* (C) respectively, which are the same names given by Buehrer and Chellapilla [29]. One important difference of our work from previous work [80, 56] is that we do not remove edges before applying the discovery algorithm. Such works remove all *nepotistic links*, that is, links between two pages that belong to the same domain, while we work based on Web pages and hyperlinks among pages. Furthermore, using domains does not make sense in terms of social networks.

For technical reasons that will be clear next, we will add all the edges (u, u) to our directed graphs. We indicate in a small bitmap of $|V|$ bits which nodes u actually had a self-loop, so that later we can remove from the edges output by our structures only the spurious self-loops.

We also note that the discovery algorithms are applied over Web graphs with *natural node ordering* [16], which is basically URL ordering, because they provide better results than using other node orderings. We retain the name *natural node ordering* used in the Web site (*http://law.di.unimi.it/webdata*) [15].

We will find patterns of the following kind.

Definition 2.1.1 A *dense subgraph* $H(S, C)$ of $G = (V, E)$ is a graph $G'(S \cup C, S \times C)$, where $S, C \subseteq V$.

Note that Definition 2.1.1 includes cliques ($S = C$) and bicliques ($S \cap C = \emptyset$), but also more general subgraphs. Our goal is to represent the $|S| \cdot |C|$ edges of a dense subgraph using $O(|S| + |C|)$ space. In Chapter 3 we explore different techniques that exploit this definition, considering bicliques and more general dense subgraphs.

2.2 Related Work

In the context of the Web, Donato et al. [54] show that several web mining techniques used to discover the structure and evolution of the Web graph, such as weakly and strongly connected components, depth-first search and breath-first search, are based on classical graph algorithms. In a later work Donato et al. [53] present an experimental study of the statistical and topological properties of the Web graph. Other proposals use graph algorithms to detect spam farms [104, 62]. Saito et al. [104] present a method for spam detection based

on classical graph algorithms such as identification of weakly and strongly connected components, maximal clique enumeration and minimum cuts. On the other hand, Gibson et al. [62] propose large dense subgraph extraction as a primitive for spam detection. Their algorithm used efficient heuristics based on the idea of shingles [25] (mostly used to estimate similarity among web pages) together with frequency itemset mining approximation. There are other techniques, based on graph algorithms, aiming to extract small subgraphs or small communities for mining purposes [80, 59, 96]. Fortunato et al. [60] provides a comparison among several community detection algorithms.

There is no consensus about the definition of a community, being there various possible definitions [5]. However, as described by Fortunato et al. [60], finding communities is somehow similar to clustering, in the sense that the members of a community are groups of vertices which probably share common properties like groups of vertices with high concentration of edges, but low concentration between different groups. These groups of vertices can be called communities or clusters.

In the context of Web graphs, communities are group of pages having topical similarities. Detecting communities in these graphs may help to identify the artificial clusters created by link farms in order to deceive ranking algorithms such as PageRank [21], with the goal of discouraging spam practices. Detecting communities on directed graphs is more challenging than on undirected graphs, since the adjacency matrix is asymmetric and then it is not possible to avoid representing all edges. Another challenge with community detection algorithms is that in many networks vertices may belong to more than one group and then it is likely that communities are overlapped. For instance, in social networks, an individual belongs to different circles at the same time.

Biological applications that use clustering or community detection include Protein-Protein interaction (PPI), as the interactions between proteins are fundamental for each process in a cell [36]. Communities correspond to functional groups, that is, to proteins having the same or similar functions. For instance, in PPI (protein-protein interaction) networks, the vertices represent proteins and edges represent interactions between proteins. Most communities are associated with cancer and metastasis, which indirectly shows how important it is to detect similar groups in PPI networks. One of the most used clustering algorithms in bioinformatic applications is MCL [55] (later mathematically analyzed [114]). MCL is an unsupervised algorithm based on simulation of stochastic flow in graphs. The algorithm consists of alternation of matrix expansion and matrix inflation, where expansion means taking the power of a matrix using the usual matrix product, and inflation is a particular way of rescaling the entries of a stochastic matrix such that it remains stochastic. MCL has been mostly applied in bioinformatic applications [27], but also in social network analysis [86]. MCL deals with both labeled and unlabeled graphs, while the clustering we use deals only with unlabeled graphs.

2.3 Discovering Dense Subgraphs

In this section we describe how we discover dense subgraphs. Even finding a clique of a certain size is NP-complete, and the existing algorithms require time exponential on that size (e.g., Algorithm 457 of Bron and Kerbosch [28]). Thus, we need to resort to fast heuristics for our huge graphs of interest. Besides, we want to capture other types of dense subgraphs, not just cliques. We first use a scalable clustering algorithm [29], which uses the idea of "shingles" [62]. Once the clustering has identified nodes whose adjacency lists are sufficiently similar, we run a heavier frequent itemset mining algorithm [29] inside each cluster. This mining algorithm is the one that finds sets of nodes S that point to all the elements of another set of nodes C (they can also point to other nodes). This algorithm was designed to find bicliques: a node u cannot be in S and C unless (u, u) is an edge. As those edges are rare in Web graphs and social networks, this algorithm misses the opportunity to detect dense subgraphs and is restricted to find bicliques. To make the algorithm sensitive to dense subgraphs, we insert all the edges $\{(u, u), u \in V\}$ in E, as anticipated. This is sufficient to make the frequent itemset mining algorithm find the more general dense subgraphs. The spurious edges added are removed at query time, as explained.

The clustering algorithm represents each adjacency list with P fingerprints (hash values), generating a matrix of fingerprints of $|V|$ rows and P columns. Then it traverses the matrix column-wise. At stage i the matrix rows are sorted lexicographically by their first i column values, and the algorithm groups the rows with the same fingerprints in columns 1 to i. When the number of rows in a group falls below a small number, it is converted into a cluster formed by the nodes corresponding to the rows. Groups that remain after the last column is processed are also converted into clusters.

On each cluster we apply the frequent itemset mining algorithm, which discovers dense subgraphs in the cluster. This algorithm first computes frequencies of the nodes mentioned in the adjacency lists, and sorts the list by decreasing frequency of the nodes. Then the nodes are sorted lexicographically according to their lists. Now each list is inserted into a prefix tree, discarding nodes of frequency 1. This prefix tree has a structure similar to the tree obtained by the hierarchical termset clustering defined by Morik et al. [90]. Each node p in the prefix tree has a label (consisting of the node id), and it represents the sequence $l(p)$ of labels from the root to the node. Such node p stores also the range of graph nodes whose list start with $l(p)$.

Note that a tree node p at depth $c = |l(p)|$ representing a range of s graph nodes identifies a dense subgraph $H(S, C)$, where S are the graph nodes in the range stored at the tree node, and C are the graph nodes listed in $l(p)$. Thus $|S| = s$ and $|C| = c$. We can thus point out all the tree nodes p where $s \cdot c$ is over a size threshold, and choose them from largest to lowest saving (which must be recalculated each time we choose the largest).

The main algorithm is given in Figure 2.1.

Figure 2.2(a) shows a dense subgraph pattern with the traditional representation and (b) shows the way we represent them using the discovery algorithm described.

Input: G: input graph; P: number of fingerprints, $Iters$: number of iterations, scs: dense subgraph size ($|S| * |C|$), $threshold$: threshold for clustering.
Output: $dscoll$: Collection of dense subgraphs
$ds \leftarrow \emptyset$
$dscoll \leftarrow \emptyset$
for $i \leftarrow 1$ **to** $Iters$ **do**
  $MatrixM \leftarrow computeFingerprints(G, P)$
  $ClustersC \leftarrow getClusters(M, G, threshold)$
  for $c \in C$ **do**
    $ds \leftarrow mine(G, c, scs)$
    $dscoll.add(bc)$
  end for
end for

Figure 2.1: Main dense subgraph discovery algorithm

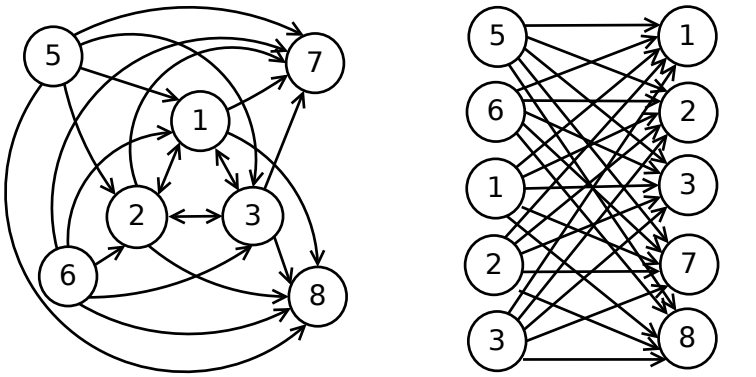


Figure 2.2: Dense subgraph representation

The whole algorithm can be summarized in the following steps. Figure 2.3 shows an example.

Step 1 Clustering-1 (build hashed matrix representing G). We traverse the graph specified as a set of adjacency lists, adding edges (u, u). Then, we compute a hash value H associated with each edge of the adjacency list P times, and choose the P smallest hashes associated with each adjacency list. Therefore, for each adjacency list, we obtain P hash values. This step requires $O(P|E|)$ time.

Step 2 Clustering-2 (build clusters). We build clusters consisting of groups of similar hashes, by sorting the hash matrix by columns, and select adjacency lists associated with clusters based on hashes. This requires $O(P|V| \log |V|)$ time.

Step 3 Mining-1 (reorder cluster edges). We compute edge frequencies on each cluster, sort-

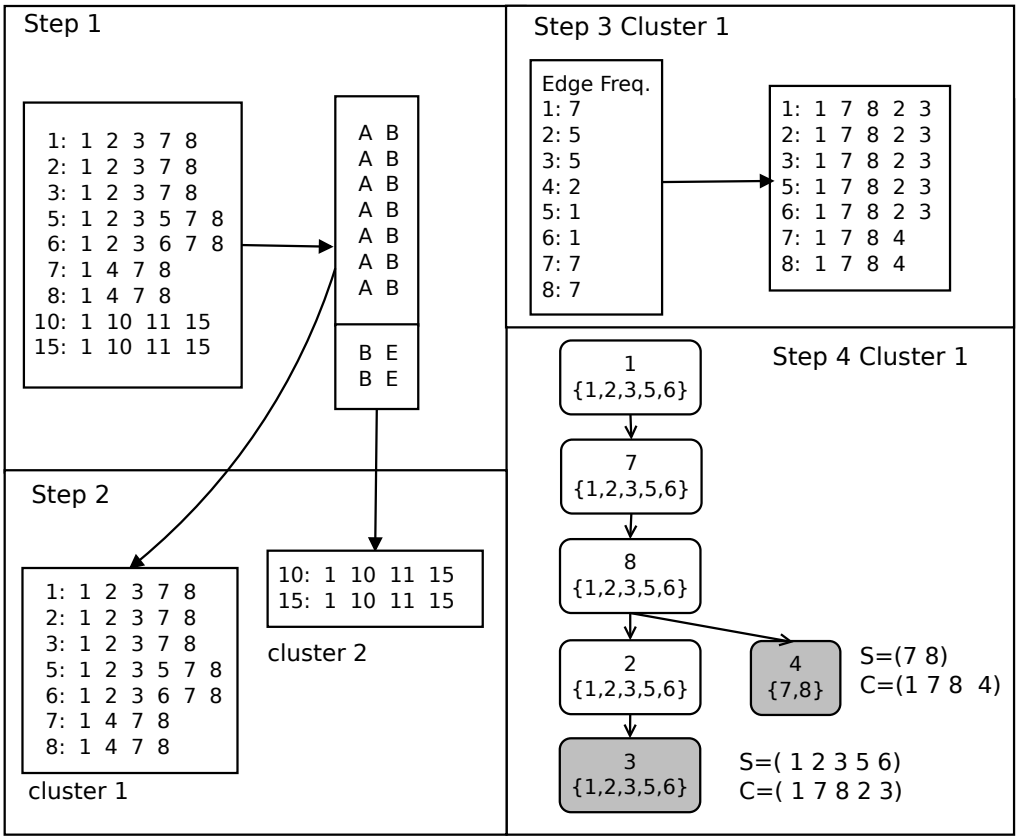


Figure 2.3: Example of the dense subgraph discovery process.

ing them from largest to smallest (discarding edges with frequency of 1), and reorder them based on that order. This step takes $O(|E| \log |E|)$ time.

Step 4 Mining-2 (discover dense subgraphs and replacing). We compute a prefix tree for each cluster, with tree nodes labeled with the node id of edges. Dense subgraphs ($G'(S \cup C, S \times C)$) with higher edge saving ($|S| \cdot |C|$) are identified in the tree. The overall step takes $O(|E| \log |E|)$ time.

Therefore the overall algorithm time complexity, taking P as a constant, is bounded by $O(|E| \log |E|)$.

In Chapter 3, the dense subgraphs found $H(S, C)$ will be replaced by a new virtual node whose in-neighbors are S and whose out-neighbors are C. As the result is still a graph, the dense subgraph discovery process can be repeated on the resulting graph, since the discovery

Table 2.1: Main statistics of the Web graphs and social networks we used in our experiments.

Dataset	$\|V\|$	$\|E\|$
eu-2005	862,664	19,235,140
indochina-2004	7,414,866	194,109,311
uk-2002	18,520,486	298,113,762
arabic-2005	22,744,080	639,999,458
sk-2005	50,636,154	1,949,412,601
enron	69,244	276,143
dblp-2011	986,324	6,707,236
LiveJournal-SNAP	4,847,571	68,993,773
LiveJournal-2008	5,363,260	79,023,142
enwiki-2013	4,206,785	101,355,853

algorithm is a heuristic and it is able to find dense subgraphs in a number of iterations. Chapter 3 also describes another representation where subgraphs of the form $H(S, C)$ will be extracted from the original graph and represented using compact data structures.

2.4 Evaluation of the Discovery Algorithm

We evaluate our discovery algorithm using synthetic and real Web graph snapshots. We use generators *GTgraph* and *RMAT* for synthetic graphs[1] [10, 32] (as explained later in this section). We also use real Web graph snapshots available from the WebGraph project[2]. In all the experiments we describe in this chapter, we use a Linux PC with 16 processors Intel Xeon at 2.4 GHz, with 72 GB of RAM, and 12 MB of cache. We used g++ compiler with full optimization.

First, we evaluate the sensitivity of the number of hashes (parameter P) used in the first step of our clustering. For doing so, we use a real Web graph (eu-2005, see Table 2.1). We measure the impact of P in various metrics that predict compression effectiveness. Table 2.2 shows the number of discovered cliques (# Cliques), total number of edges in those cliques (|Cliques|), number of bicliques (# Bicliques), total number of edges in cliques and bicliques (Edges), total number of nodes participating in cliques and bicliques (Nodes), and the ratio between both (Ratio, which gives the reduction factor using our technique that represents edges implicitly, described in Section 4.1).

All these metrics show that using $P = 2$ is slightly better than using other values. When increasing P, the algorithm discovers more and smaller cliques and bicliques, but the overall compression in terms of representing more edges with fewer vertices is better with $P = 2$.

Second, we evaluate our subgraph discovery algorithm. For doing so, we use the *GTgraph* suite of synthetic graph simulators. From this suite, we use the `SSCA#2` generator to create

[1] Available at `www.cse.psu.edu/~madduri/software/GTgraph`

[2] `http://law.dsi.unimi.it` [15]

Table 2.2: Compression metrics using different P values with eu-2005

P	# Cliques	|Cliques|	# Bicliques	$Edges$	$Nodes$	$\frac{Edges}{Nodes}$
2	33,482	248,964	58,467	17,208,908	2,357,455	7.30
4	34,237	246,022	60,226	17,199,357	2,426,753	7.08
8	34,863	245,848	60,934	17,205,357	2,524,240	6.81

Table 2.3: Synthetic clique graphs with different number of nodes (Nodes), edges (Edges), maximum clique size (MC), and total number of vertices participating in cliques (R). Column d gives the average number of edges per node, and the last column is the average clique size

Name	Nodes	Edges	d	MC	R	avg size
PL	999,993	9,994,044	9.99	0	0	-
V16	65,536	610,500	9.31	15	6,548	9.50
V16	65,536	1,276,810	19.48	30	3,785	17.09
V16	65,536	2,161,482	32.98	50	2,398	27.21
V16	65,536	4,329,790	66.06	100	1,263	51.83
V17	131,072	1,214,986	9.26	15	13,130	9.48
V17	131,072	2,542,586	19.39	30	7,589	17.05
V17	131,072	4,309,368	32.87	50	4,790	27.23
V17	131,072	8,739,056	66.67	100	2,495	52.95
V20	1,048,576	9,730,142	9.76	15	104,861	9.50
V20	1,048,576	20,293,364	19.60	30	60,822	17.02
V20	1,048,576	34,344,134	32.90	50	38,544	27.07
V20	1,048,576	69,324,658	66.18	100	20,102	52.10

random-sized clique graphs [10, 32]. We use the parameter *MaxCliqueSize* to set the maximum size of cliques (MC), set the *Scale* parameter to 16, 17 or 20, so as to define 2^{16}, 2^{17} or 2^{20} vertices on the graph, and set the parameter $ProbIntercliqueEdges = 0.0$ (probability of edges among cliques), which tells the generator to create a clique graph, that is, a graph consisting of isolated cliques. Therefore, with this generator we can precisely control the actual cliques present in the graph, and their corresponding sizes. We call those *real cliques*.

We also use the generator `R-MAT` of the suite to create a power-law graph without any cliques. The properties of the synthetic clique graphs and the power-law graph used are described in Table 2.3. The first graph, PL, is the power-law graph, whereas the others are clique graphs (V16,V17,V20). Finally, we define new graphs (PL-V16, PL-V17, and PL-V20), which are the result of merging graphs PL with V16, PL with V17, and PL with V20. The merging process is done by computing the union of the edge sets belonging to the PL graph and one of the clique graphs. That is, both PL and Vxx share the same set of nodes (called 1 to $|V|$) and we take the union of the edges in both graphs. We apply our dense graph discovery algorithm on those merged graphs, whose features are displayed in Table 2.4. Figure 2.4 (left) shows the out-degree histogram for PL, V17 (with $MC = 100$) and PL-V17 graphs. We evaluate the ability of our discovery algorithm to extract all the real cliques from these graphs.

Table 2.4: Synthetic merged power-law and clique graphs.

Name	Nodes	Edges	MC	d
PL-V16	999,993	10,604,408	15	10.60
PL-V16	999,993	11,270,660	30	11.27
PL-V16	999,993	12,155,249	50	12.15
PL-V16	999,993	14,323,320	100	14.32
PL-V17	999,993	11,208,968	15	11.20
PL-V17	999,993	12,536,277	30	12.53
PL-V17	999,993	14,303,175	50	14.30
PL-V17	999,993	18,732,584	100	18.73
PL-V20	1,048,576	19,724,071	15	18.81
PL-V20	1,048,576	30,287,168	30	28.88
PL-V20	1,048,576	44,337,825	50	42.28
PL-V20	1,048,576	79,317,960	100	75.64

For evaluation purposes we compare our clustering against MCL clustering,[3] by changing the first steps (finding clusters) in our discovery algorithm.

To measure how similar the discovered and the real clique sets are, we compute the Average Relative Error (ARE), which is the average of the absolute difference between true and discovered cliques:

$$ARE = \frac{1}{|R|} \sum_{i \in R} \frac{|r_i - \hat{r}_i|}{r_i} \ , \tag{2.1}$$

where r_i and $\hat{r}_i$ are the real and discovered clique sizes, and $|R|$ is the number of real cliques. We consider a real clique to be "discovered" if we find more than half of its vertices.

We also evaluate the discovery algorithm based on precision and recall:

$$precision = \frac{\sum_{i \in R} |RCE \cap DCE|}{\sum_{i \in R} |DCE|} \ , \tag{2.2}$$

$$recall = \frac{\sum_{i \in R} |RCE \cap DCE|}{\sum_{i \in R} |RCE|} \ , \tag{2.3}$$

where RCE is the node set of a real clique and DCE is the node set of the corresponding discovered clique.

In addition, we compare the number of discovered cliques ($|A|$) with respect to real cliques:

$$recallNumCliques = \frac{|A|}{|R|} \ . \tag{2.4}$$

[3] Available at `http://micans.org/mcl/`

Table 2.5: Time required per retrieved clique of different sizes. *avg* refers to the clique average size, *tms* refers to the average time in milliseconds to discover all cliques ($|A|$) using our dense subgraph algorithm. $|A|$M denotes all cliques discovered with MCL with corresponding *avg*M clique average size, and *tms*M and *ptms*M denote the sequential and parallel execution time for MCL.

Name	*MC*	$\|A\|$	*avg*	*tms*	$\|A\|$M	*avg*M	*tms*M	*ptms*M
PL-V16	15	6,501	9.00	236.1	5,810	7.96	4,359.2	1,938.5
PL-V16	30	3,766	16.53	336.4	3,596	15.18	7,877.3	3,129.1
PL-V16	50	2,389	26.58	305.1	2,331	25.40	11,190.4	5,089.2
PL-V16	100	1,261	51.08	590.0	1,242	50.80	19,839.7	9,363.1
PL-V17	15	13,071	9.00	120.5	12,032	8.30	2,048.4	977.9
PL-V17	30	7,565	16.53	129.8	7,321	15.83	3,226.3	1,612.3
PL-V17	50	4,776	26.70	203.1	4,706	26.21	4,886.3	2,394.1
PL-V17	100	2,492	51.85	318.2	2,481	51.89	10,153.5	4,446.1
PL-V20	15	104,771	9.06	103.1	103,437	9.31	580.2	103.6
PL-V20	30	60,773	16.56	150.3	60,614	16.97	614.6	152.4
PL-V20	50	38,524	26.62	155.4	38,473	27.09	639.7	248.2
PL-V20	100	20,095	51.62	178.6	20,097	52.11	1,371.1	505.7

In order to compare the clustering algorithms, we first measure execution times. We execute the version of the discovery algorithm that uses MCL only with one iteration with $I = 2.0$ (default setting for *Inflation* parameter). We also execute our clustering, where we use 40 to 100 iterations in order to reach similar clustering quality (yet our iterations are much faster than that of MCL). Table 2.5 shows the number of discovered cliques ($|A|$), average sizes (*avg*), and the average time in milliseconds (*tms*) to discover all the cliques when using our dense subgraph algorithm. We also add the corresponding values obtained using MCL clustering ($|A|$M, *avg*M). The MCL execution time (*tms*M) considers sequential time, whereas *ptms*M considers parallel execution time with 16 threads. We report here our sequential execution times. Still, already our sequential algorithm is an order of magnitude faster than sequential MCL. Our approach works better than MCL for graphs that have fewer cliques, as in PL-V16 and PL-V17. In such cases, even our sequential time with multiple iterations is much faster than one iteration of the the parallel MCL with 16 threads. For graphs that contain more cliques and small MC values, the time of our sequential algorithm is comparable to parallel MCL using 16 threads, yet, as the cliques grow, MCL does not scale well and even its parallel version becomes slower than ours.

Figure 2.4 (right) shows that *ARE* (Eq. (2.1)) values are very low in our strategy (less than 0.06, i.e., 6%) and the error grows slightly when the number of cliques increases in graphs. However, changing our clustering algorithm to MCL, the average relative error increases when the graph contains smaller or fewer cliques hidden in the graph. On the other hand, in all cases we have a precision of 1.0, which means that we only recover existing cliques. Figure 2.5 (left) shows recall (Eq. (2.3)), and again we observe that our discovery algorithm behaves very well (more than 0.93, i.e., 93%) for different number and size of cliques hidden in the graphs. In contrast, MCL is very sensitive to the number and size of cliques, being less effective for fewer or smaller cliques. We see a similar behavior in Figure 2.5 (right), where

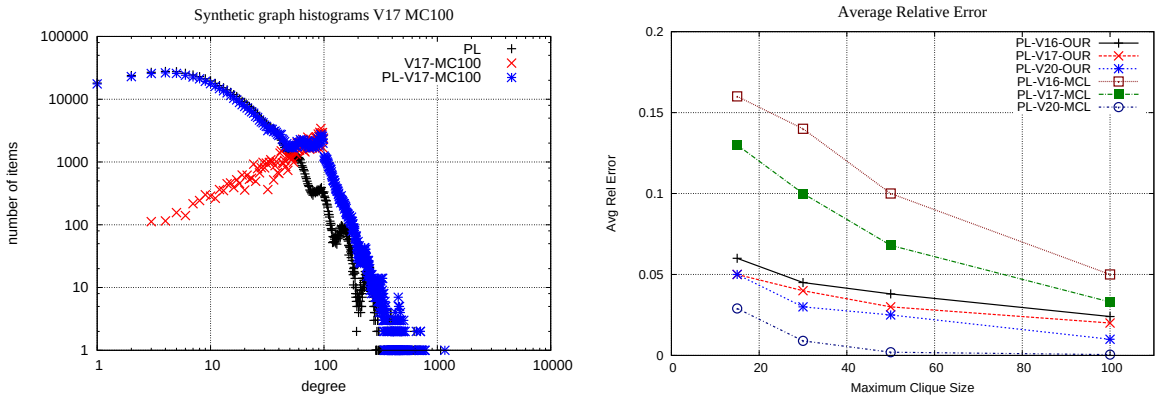


Figure 2.4: Outdegree histograms (left) and Average Relative Error (right) in synthetic graphs

we measure $recallNumCliques$ (Eq. (2.4)).

To summarize, with our discovery strategy we discover 98%–99% of the cliques (Figure 2.5 (right)), and find their correct vertices with average relative errors between 1% and 6% (Figure 2.4 (right)). The performance is better for larger cliques. One possible reason is that the clustering algorithm we use tends to find greater similarity on those adjacency lists that have more vertices in common.

We also evaluate the impact in scalability and compression (described in Section 4.1) using MCL over a real undirected social graph (dblp-2011, see Table 2.1). We execute MCL with different values for the inflation parameter (I). Table 2.6 shows the compression (bpe) and sequential execution time (tms) and parallel execution with 16 threads (ptms). It also shows that our clustering approach outperforms MCL, achieving less space than its slowest construction within the time of its fastest construction.

Table 2.6: Compression (bpe) and time using MCL with different inflation I values for dblp-2011

Metric	Inflation (I)					Ours
	1.2	1.4	2.0	3.0	4.0	
bpe	8.76	9.43	10.17	10.44	10.51	8.41
tms	116,093	36,258	11,643	5,736	5,671	5,449
ptms	17,313	5,509	2,072	1,526	1,710	

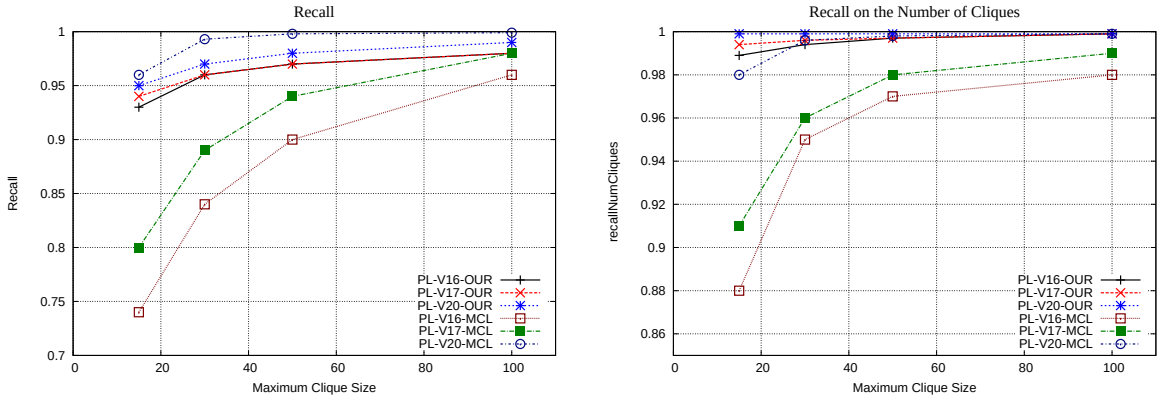


Figure 2.5: Recall on the number of vertices (left) and on the number of cliques (right) discovered in synthetic graphs

Table 2.7: Compression (bpe) and time using MCL with different inflation values I for eu-2005

Metric	Inflation (I)					Ours
	1.2	1.4	2.0	3.0	4.0	
bpe	3.46	3.13	3.18	3.21	3.25	2.67
tms	-	-	-	-	-	2,874
ptms	65,359	62,297	59,535	59,285	89,066	-

To confirm the scalability problems of MCL, we also execute it over a larger graph, namely eu-2005 (which is the smallest Web graph we use, see Table 2.1). We use different I values, from $I = 1.2$ to $I = 4.0$ (using $I = 6.0$ takes more than 2 days). We use parallel MCL with 16 threads; sequential MCL was disregarded since the parallel execution is already several orders of magnitude slower than our sequential algorithm. Table 2.7 shows the results, where we also give the achieved compression in *bpe* using our compact data structures (Section 4.4.2). This confirms that the clustering we use in our discovery algorithm is much more scalable than MCL.

The MCL scalability issue has been reported in several works [86, 88, 83, 68]. In fact, Mishra [88] reports that MCL performs poorly on sparse graphs. Additionally, the time complexity of our algorithm is $O(E \log E)$, while a straightforward implementation of MCL is $O(V^3)$ time, as mentioned in the MCL web site FAQ section[4]. Another issue with MCL is that it does not guarantee good effectiveness on directed graphs[5].

[4] `http://micans.org/mcl/man/mclfaq.html#howfast`
[5] `http://micans.org/mcl/man/mclfaq.html#goodinput`

2.5 Conclusions

This chapter describes a heuristic-based algorithm for discovering dense subgraphs that include cliques, bicliques and some combinations of both. Since, finding a clique of a given size is NP-complete, we adapt an idea that finds bicliques using shingles [29]. By inserting the edges (v, v) for each node v, the algorithm finds more general dense subgraphs that include cliques and bicliques. We compare this new algorithm with MCL, a state-of-the-art clustering algorithm. Our results confirm, in the scenario of Web graphs, previous findings about lack of scalability of MCL.

The algorithm is based in four steps. The first step consists of transforming each adjacency list of the graph into P hash values using the idea of "shingles". The second step consists of building clusters formed by groups of similar hashes. The third step consists of taking the adjacency lists of each cluster and sorting them by edge frequency. The last phase consists of building a prefix tree for each cluster with tree nodes labeled with the node id of edges. Dense subgraphs with higher edge saving ($|S| \cdot |C|$) are identified in the tree.

In terms of the evaluation of the algorithm, we first determine the best P in the clustering algorithm. Then, we compare the clustering algorithm against MCL using synthetic and real datasets in terms of quality of the solution and execution time. We measure quality, based on precision, recall, and average relative error. Using synthetic graphs, we found that the quality of our solution provides a very low average relative error (less than 0.06, i.e., 6%) and the error grows slightly when the number of cliques increases in graphs. However, changing the clustering algorithm to MCL, the average relative error increases when the graph contains smaller or fewer cliques hidden in the graph. In terms of recall, we observe that our discovery algorithm behaves very well (more than 0.93, i.e., 93%) for different number and size of cliques hidden in the graphs. In contrast, MCL is very sensitive to the number and size of cliques, being less effective for fewer or smaller cliques. In terms of execution time, we found that our strategy is much more efficient than using MCL since the sequential algorithm is much faster (about 20 times faster) than the parallel version (with 16 threads) of MCL.

Chapter 3

Web Graph Compression by Factoring Edges

This chapter presents compression approaches for Web graphs based on factoring edges. We evaluate different strategies for reducing the number of edges of the input graph and then apply compression techniques that have been successful for compressing Web graphs. Such techniques exploit other properties such as similarity and locality of adjacency lists, BFS and Layered Label Propagation (LLP) node orderings, and sparseness of the graph represented by an adjacency matrix. Our schemes provide compressed representations for retrieving out-neighbors, that is, the outlinks of a node; out/in-neighbors, that is both outlinks (nodes to which a node points) and inlinks (nodes that point to a given node).

3.1 Composing Methods

In this section we evaluate the impact of combining edge-reduction with other methods. We proceed in two stages: an edge-reduction stage yields a new graph, containing fewer edges and more nodes (including virtual nodes); and a compression stage that applies existing compression techniques on the edge-reduced graph.

We first study the impact of reducing edges on Web graphs using Re-Pair [42] and VNM [29] (only for factoring edges), and combine them to improve the edge reduction. We refer to VNM as the version that captures only bicliques (i.e., where $S \cap C = \emptyset$) and adds virtual nodes to connect sets S and C. Our study shows that reducing edges with VNM and then applying the compression scheme of Apostolico and Drovandi [9] provides competitive compression for out-neighbor queries. We also show that using the edge-reduced graph and then using k2tree provides good compression ratios for out-in/neighbor navigation over Web graphs. We show that applying this scheme over social networks does not work well.

The second approach considers Dense Subgraph Mining with virtual nodes (DSM) with set overlaps (i.e., where $S \cap C \neq \emptyset$), instead of just bicliques for factoring edges. We show that using DSM with virtual nodes (in a similar way as done in the previous scheme), and

then applying the compression scheme of Apostolico and Drovandi [9] provides the best compression for out-neighbor navigation support. We also apply Layered Label Propagation node ordering (LLP) proposed by Boldi et al. [16] over the edge-reduced graph before applying their compression technique. We found that this ordering is also attractive in terms of compression and random access time performance. On the other hand, using the same edge-reduced graph with BFS node ordering, and then k2tree [23] provides the best compression for out/in-neighbor support for Web graphs.

3.2 Bicliques with Virtual Nodes (VNM)

First, we study the edge-reduction stage using three alternatives. The first is VNM, which we implemented as described in its article [29], using C++ and STL, just for finding bicliques. However, we did not implement their compression of the resulting graph. In this section, we use the bpe (bits per edge) reported by the authors for comparison (VNM_b). In our implementation, we have three parameters: ES specifies the minimum size $|S|\cdot|C|$ to consider for edge factoring, T is the numbers of iterations we carry out, and P is the number of hashes used in the clustering stage of the discovery algorithm (which is described in detail in Chapter 2).

Second, we implemented Re-Pair as just an edge-reducing method (i.e., rule $r \rightarrow ab$ is seen as the creation of a virtual node r with edges to a and b), which we call RP_o. Third, we consider VNMRP, which applies VNM and then RP_o.

For the compression stage we also considered three alternatives when using VNM with bicliques. First, we used the full Re-Pair compression scheme obtained from its authors [42] (RP_c). Second, we used WebGraph (BV), version 3.4.0 taken from `http://webgraph.dsi.unimi.it`. Third, we used AD version 0.2.1, without dependencies[1].

3.2.1 Performance Effectiveness

We used datasets of Web graphs and social networks of Table 2.1. We executed our experiments on a Linux PC with 8 processors Intel Xeon at 2.4GHz, with 32 GB of RAM and 12 MB of cache, using sequential algorithms.

The first experiment evaluates the edge-reduction ratio defined as the total number of edges of the original graph divided by the edges remaining after edge-reduction (including from/to virtual nodes). Figure 3.1 shows the edge-reduction ratio achieved by VNM, RP_o, and by VNMRP for different numbers of iterations. The results suggest that combining both techniques provides better compression rates than applying either of them individually.

Figure 3.2 shows the node degree distribution using a size-rank plot (to each abscissa x we associate the sum of the frequencies of all data points with abscissa greater than or equal to

[1] Available at http://www.dia.uniroma3.it/~drovandi/software.php

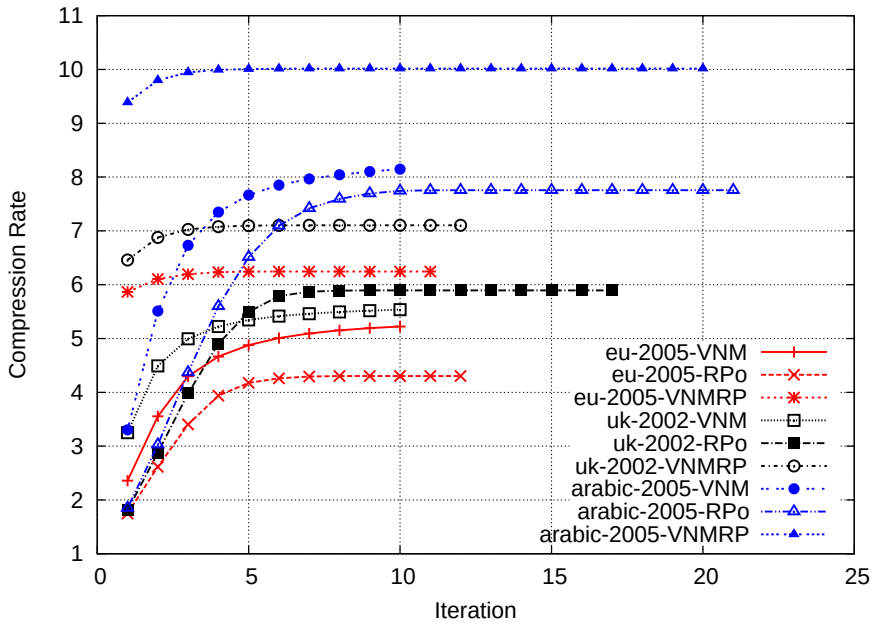


Figure 3.1: Compression rate based on number of edges using VNM for factoring edges only.

x) for the indochina-2004 original graph and after applying VNM, RP_o, and VNMRP. Very similar figures are obtained for the other graphs. We observe that reducing edges produces very clean power law distributions in the node degrees, regardless of the edge-reduction method used. However, this new shape does not seem to have, in general, an impact on the compression achieved with AD, beyond the mere reduction in the data size.

The second experiment measures the compression in terms of *bpe* (bits per edge) aiming at compression with random access. In this experiment, we combine edge-reduction using VNM with compression using BV and AD. We tuned VNM using parameter ES. VNM* ($ES = 4$) allows the discovery of any virtual node pattern that reduces edges, thus minimizing the edges as much as possible in the edge-reducing stage. VNM* does not achieve the lowest possible bpe values when combined with BV, but these are achieved with $ES = 30$ on `eu-2005`, and $ES = 100$ on the other Web graphs (those are used in the row labeled VNM). The best compression is achieved with loose host-by-host Gray ordering of BV with $w = 7$ and $m = -1$ (for maximum compression). This suggests that there are sufficient regularities that BV can exploit after removing redundant edges, even if there are more nodes in the graph. On the other hand, using VNM* with AD achieves the best compression.

Table 3.1 shows the compression results to support direct access. Bpe values using BV schemes include offset spaces for supporting random access. We include bare compression methods (VNM_b, RP_c, BV, and AD) and our best performing combinations. To enable direct access we use BV with $w = 7$ and $m = 3$, and AD with $l = 4$ and 8. Since both BV and AD exploit locality and similarity, the results suggest that the BFS ordering used in AD works better than the LLP ordering used in BV in terms of compression. We denote VNM+BV to the case when we apply BV over the edge-reduced graph without reordering with LLP the

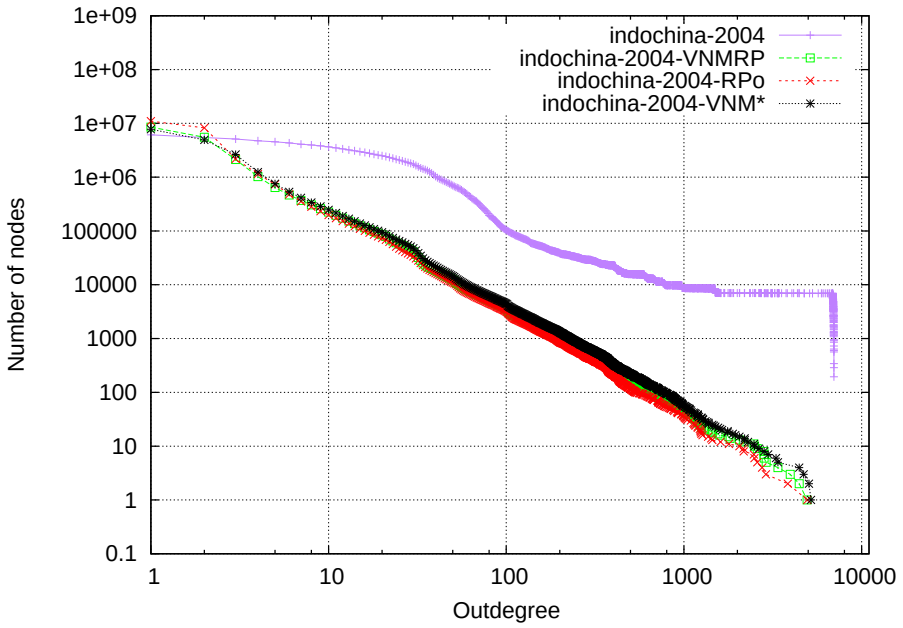


Figure 3.2: Node degree distribution using size-rank plot of indochina-2004.

reduced graph, and VNM+BVLLP to when we apply LLP ordering over the edge-reduced graph before applying the compression technique. We achieve the best compression results using VNM*+AD$_8$.

3.2.2 Bidirectional Navigation

We additionally consider combining edge-reduction methods with techniques that support out/in-neighbor queries. We evaluate the following variants:

T1: Re-Pair GMR [41] on the original graph.
T2: k2treeNAT [22] on the original graph with URL-based ordering (denominated *Natural* by Boldi et al. [16]).
T3: VNM on the original graph and then k2tree over the reduced graph, VNM-ESx-Ty+k2tree, where x represents the size of bicliques to capture and y the number of iterations.

The space/time requirements of these techniques on Web graphs are displayed in Figure 3.3. The combination of VNM and k2tree (T3) achieves the best space efficiency on Web graphs when supporting bidirectional neighbors. However, this comes at a significant price in access time. The combination, on the other hand, does not work well on social networks. Table 3.2 shows that, while the edge reduction on Web graphs is between 5 and 8.25, the reduction on social network graphs is not attractive, being less than two, which suggests that this scheme does not work well on social networks.

Table 3.1: Compression in bpe when allowing random access, for various methods. VNM_b are the bpe values reported by Buehrer and Chellapilla [29]. Best compression values are shown in bold.

Name	eu-2005	indochina-2004	uk-2002	arabic-2005
VNM_b	2.90	-	1.95	1.81
RP_c	4.98	2.62	4.28	3.22
VNM^*+RP_c	3.57	2.30	3.65	2.78
$BVLLP_{m3w7}$	4.20	1.78	2.81	2.17
VNM+BV_{m3w7}	2.87	1.51	2.42	1.87
VNM+$BVLLP_{m3w7}$	2.71	1.32	2.22	1.59
AD_{l8}	3.64	1.49	2.64	2.07
RP_o+AD_{l8}	3.15	1.71	2.67	2.01
VNMRP+AD_{l8}	2.28	1.17	1.92	1.50
VNM^*+AD_{l4}	2.39	1.24	2.05	1.57
VNM^*+AD_{l8}	**2.26**	**1.12**	**1.87**	**1.47**

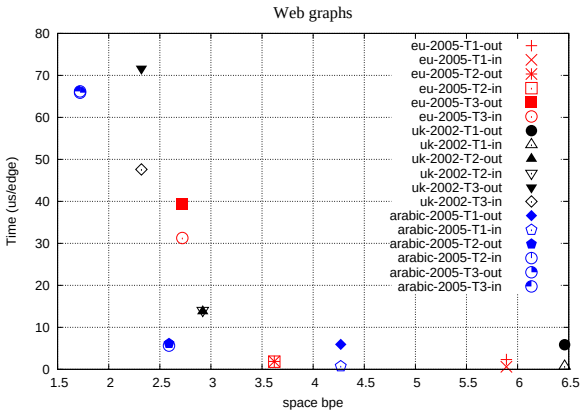


Figure 3.3: Space/time efficiency with out/in-neighbor queries with T1,T2, and T3 on Web graphs.

3.3 Dense Subgraphs with Virtual Nodes (DSM)

In this section we describe the second scheme, where instead of using bicliques with virtual nodes, we use dense subgraphs (with $S \cap C \neq \emptyset$) with virtual nodes. Depending on the representation of the final graph we obtain various structures supporting out-neighbor and out/in-neighbor navigation. We show that using DSM improves the results obtained with VNM for Web graphs, described in previous section.

In a first phase we apply the discovery of dense subgraphs explained in Chapter 2. Then we apply the idea of virtual nodes [29] (VNM) over the original graph, to factor out the edges of the dense subgraphs found. In this case, we add self-loops nodes to adjacency lists to include cliques in the discovery not just bicliques (as explained in Chapter 2).

As the result is still a graph, we iterate on the process. On each iteration we discover

Table 3.2: Edge reduction given by $|E|/|E2|$, where $E2$ is the number of edges on the edge reduced graph (after applying VNM over the original graph in 10 iterations).

Dataset	$\|E\|/\|E2\|$
eu-2005	5.26
uk-2002	5.58
indochina-2004	9.21
arabic-2005	8.25
sk-2005	8.24
enron	1.32
dblp-2011	1.25
LiveJournal-SNAP	1.24
LiveJournal-2008	1.27
enwiki-2013	1.18

dense subgraphs in the current graph, and replace their edges using virtual nodes. We refer to this approach as DSM (Dense Subgraph Mining).

The outcome of this phase is a graph equivalent to the original one, in the sense that we must expand paths that go through virtual nodes to find all the direct neighbors of a node. The new graph has far fewer edges and a small amount of virtual nodes in addition to the original graph nodes. On a second phase, we apply different state-of-the-art compression techniques and node orderings over this graph to achieve compression and fast out- and out/in-neighbor queries. Likewise, using VNM, this scheme also used the parameters ES, T, and P.

As explained, we input the graph in natural ordering to the DSM algorithm. If we retain this order on the output and give virtual nodes identifiers larger than those of the original nodes, we can easily distinguish which nodes are virtual and which are original. If, we instead, use a different ordering on the output, such as BFS, we need an additional bitmap to mark which nodes are virtual.

3.3.1 Dense Subgraph Mining Effectiveness

In the experiments of this section we use Web graph snapshots available from the *WebGraph* project [15], described in Table 2.1. Table 3.3 gives the main statistics of the Web graphs used. We define $G1(V1, E1)$ as the original Web graph and $G2(V2, E2)$ as the result of removing the (u, u) edges from $G1$ (as explained, we will store a bitmap marking which of those edges were originally present). Algorithm DSM will operate on $G2$ (where it will start by adding (u, u) for every node). We call $G3(V3, E3)$ the outcome of the DSM algorithm, where $V3 = V1 \cup VN$, VN are the virtual nodes added, and $E3$ are the resulting edges in $G3$. We always use $P = 2$ for DSM.

In order to compare the use of *bicliques* (VNM) with *dense subgraphs* (DSM) we show some statistics using virtual nodes and using dense subgraphs. Table 3.4 shows the main features

Table 3.3: Main statistics of the Web graphs we used in our experiments, where $V1$ and $E1$ are the number of nodes and edges of the original graph, $E2$ is the number of edges of the graph discarding self-loops, $d1 = \frac{|E1|}{|V1|}$ and $d2 = \frac{|E2|}{|V1|}$.

Dataset	$\|V1\|$	$\|E1\|$	$d1$	$\|E2\|$	$d2$
eu-2005	862,664	19,235,140	22.30	18,733,713	21.72
indochina-2004	7,414,866	194,109,311	26.18	191,606,827	25.84
uk-2002	18,520,486	298,113,762	16.10	292,243,663	15.78
arabic-2005	22,744,080	639,999,458	28.14	631,153,669	27.75
sk-2005	50,636,154	1,949,412,601	38.50	1,930,292,948	38.12

Table 3.4: Statistics using bicliques (VNM) versus dense subgraphs (DSM) with virtual nodes for 10 iterations, where $V3$ and $E3$ are the number of nodes and edges in the edge-reduced graph, $d3 = \frac{|E3|}{|V3|}$ and $|VN|$ is the number of virtual nodes in the edge reduced graph.

Dataset	Subgraph	$\|V3\|$	$\|E3\|$	$d3$	$\|E2\|/\|E3\|$	$\|VN\|$
eu-2005	Biclique	1,040,453	3,560,528	3.42	5.26	177,789
	Dense	1,042,260	3,516,473	3.37	5.32	179,596
indochina-2004	Biclique	8,302,612	21,885,657	2.63	8.75	887,745
	Dense	8,281,465	20,784,639	2.50	9.21	866,599
uk-2002	Biclique	20,543,057	52,298,694	2.54	5.58	2,022,571
	Dense	20,663,762	51,247,927	2.48	5.70	2,143,276
arabic-2005	Biclique	26,242,000	76,447,369	2.91	8.25	3,497,920
	Dense	26,193,220	74,071,714	2.82	8.52	3,449,140
sk-2005	Biclique	57,457,788	237,222,117	4.12	8.14	6,821,637
	Dense	57,609,639	234,141,918	4.06	8.24	6,973,485

of $G3$, using $ES = 10$ and carrying out 10 iterations. The table also shows the number of virtual nodes ($|VN|$), the resulting average arity ($d3$), and the size gain estimation based on the edge reduction, given by $|E2|/|E3|$. The edge reduction is significant, from 5X to 9X, whereas the increase in nodes is moderate, 7%–20%. As observed in Table 3.4, using DSM improves edge-reduction ratio ($|E2|/|E3|$) compared with VNM, using a similar number of virtual nodes ($|VN|$). Table 3.4 only shows results on Web graphs since the edge reduction ratio on social networks is always less than 2.

3.3.2 Performance Evaluation with Out-neighbor Support

In this section we evaluate the space and time performance when supporting out-neighbor queries, by applying DSM and then state-of-the-art compression on the resulting graph. For the second phase we use BVLLP (version 3.4.0 from *WebGraph*, which uses LLP ordering [16]) and AD (version 0.2.1 of their software, giving it the input in natural order [9]). We compare our results with the best alternatives, including BV [16], AD [9], and GB [66]. Combining DSM with GB was slightly worse than GB standalone, so we omit that combination. We also omit the other representations [42] that are less competitive as we showed in section 3.2.

Table 3.5 shows the compression achieved with the combinations. We add the best result from the previous section which used VNM (see Table 3.1) instead of DSM. The parameters for each of the techniques are tuned to provide the best performance. We refer to BVLLP as applying BV using LLP node ordering with parameters $m = 100$ and $w = 7$, where m is the maximum reference chain and w is the window size (those parameter values improve compression, but increase access times a little, as observed in Figure 3.4 (left)); AD_l as using AD with parameter l; and GB_h as using GB with parameter h. For our representations we add a bitmap of length $|V|$ marking which nodes have a self-loop (as our technique otherwise loses this information). We use RRR for compressing the self-loop bitmap. We compute bits per edge (bpe) as the total amount of bits of the compressed graph plus the self-loop bitmap, divided by $E1$.

We refer to DSM-ESx-Ty as using $ES = x$ and iterating DSM for $T = y$ times. We tuned our combinations using DSM with BV_{m3w7} (DSM-ESx-Ty+BV), BVLLP_{m3w7} (DSM-ESx-Ty+BVLLP, reordering nodes with LLP after edge factorization), and DSM with AD_8 (DSM-ESx-Ty+AD_8). Using DSM with BV (without reordering nodes), we found that the best ES values were 30 for eu-2005 and 100 for indochina-2004, uk-2002 and arabic-2005; while the best T value was 10. On the other hand, the best ES value when combining DSM with AD were 10 for eu-2005 and arabic-2005; and 15 for indochina-2004 and uk-2002. Those are the x values that correspond to ESx in the table.

Table 3.5 shows GB outperforms BV and AD by a wide margin. Among our representations, the one using $T = 10$ combined with AD_8 gives the best results, improving the previous result when we used VNM with AD. Overall, in most datasets, the best compression ratio for accessing out-neighbors is achieved by GB_{128}, but our technique is very close for datasets *uk-2002* and *arabic-2005*, and we slightly outperform it for *indochina-2004*. Only for the smallest graph, *eu-2005*, is GB_{128} better by far. Nevertheless, as observed in Figure 3.4 (right), over transposed graphs our technique achieves better compression and access time

Table 3.5: Compression performance in bpe, with support for out-neighbor queries. The best performing one per graph is in bold and the second best in italics.

Dataset	eu-2005	indochina-2004	uk-2002	arabic-2005	sk-2005
BVLLP$_{m100w7}$	3.74	1.47	2.37	1.78	1.92
AD$_{l100}$	2.92	1.10	1.93	1.66	1.86
GB$_{h128}$	**1.70**	*1.07*	**1.71**	**1.29**	**1.43**
VNM*+AD$_{l8}$	2.26	1.12	1.87	1.47	1.58
DSM-ESx-T10+BV$_{m100w7}$	2.92	1.35	2.43	1.89	1.82
DSM-ESx-T10+BVLLP$_{m100w7}$	2.60	1.22	2.06	1.48	1.65
DSM-ESx-T10+AD$_{l8}$	*2.19*	**1.04**	*1.81*	*1.39*	*1.58*

than GB$_h$, and the sum favors our techniques when supporting in- and out-neighbors (i.e., when storing both the direct and reverse graphs).

Table 3.6 displays the space and time performance for out-neighbor (Direct), in-neighbor (Reverse) and in/out-neighbor (Direct/Reverse) queries. We show the results for different techniques for the Web graph eu-2005 using the best compression parameters for each technique. Tables 3.7, 3.8, 3.9, 3.10 show the performance for indochina-2004, uk-2002, arabic-2005 and sk-2005 respectively.

Figure 3.4 (left) shows the space/time tradeoffs achieved using BV, AD, and GB (using parameter value $h = 8$, 32, 64, 128), compared to using DSM before applying BV or AD. When combining DSM with BV we used the optimum ES values mentioned above, and used BV with parameters $w = 7$, and $m = 3$, 100, and 1000. When combining with AD we also use the optimum ES value and test different values of l for AD in the second phase. We did not use a greater T because the edge reduction obtained did not compensate the extra virtual nodes added. We compute the time per edge by measuring the total time, t, needed to extract the out-neighbors of all vertices in $G1$ in a random order, and then dividing t by the total number of recovered edges (i.e., $|E1|$).

We observe that both BV and AD improve when combined with DSM. In particular, the combination of DSM with AD dominates BV, AD, and DSM plus BV. It achieves almost the same space/time performance as GB, which dominates all the others, and surpasses it in graph indochina-2004. Only in the smallest graph, eu-2005, does GB clearly dominate our combination.

Figure 3.4 (right) shows the same results on the transposed graphs. Note that the DSM preprocessing is the same for the original and the transposed graphs, so we preprocess the graph once and then represent the reduced original and transposed graphs. On the transposed graphs, we observe that the alternative that combines DSM with BV actually performs worse than plain BV on large graphs. GB does not perform as well as on the original graphs, but on *eu-2005* it is the best alternative. AD behaves very well on *uk-2002*, but our best combination outperforms it over the other datasets. In fact, our best combination is one of the two best alternatives in all datasets.

Table 3.6: Compression performance in bpe and random access time in microseconds for neighbor queries, for eu-2005. The best performing alternatives considering space and access time are in bold. We only include the best alternative for Direct/Reverse queries of DSM + BV which is DSM + BVLLP.

Technique	Direct		Reverse		Direct/Reverse	
	bpe	time (us)	bpe	time (us)	bpe	time (us)
$BVLLP_{m3w7}$	**4.20**	**0.124**	**3.30**	**0.087**	**7.50**	**0.105**
$BVLLP_{m50w7}$	3.77	0.737	3.15	0.193	6.92	0.465
$BVLLP_{m100w7}$	3.74	1.368	3.15	0.227	6.89	0.796
AD_{l4}	4.40	0.124	3.69	0.096	8.09	0.110
AD_{l8}	**3.64**	**0.184**	**3.31**	**0.148**	**6.95**	**0.166**
AD_{l25}	3.10	0.439	3.03	0.371	6.13	0.405
AD_{l100}	2.92	1.493	2.89	1.337	5.81	1.414
GB_{h16}	**3.02**	**0.301**	**2.61**	**0.461**	**5.63**	**0.382**
GB_{h32}	2.34	0.496	2.16	0.684	4.50	0.592
GB_{h64}	1.93	0.841	1.90	1.093	3.83	0.968
GB_{h128}	1.70	1.448	1.77	1.795	3.47	1.162
DSM-ES30-T10+BV_{m3w7}	3.07	0.196	2.86	0.115	-	-
DSM-ES30-T10+BV_{m50w7}	2.93	0.387	2.77	0.175	-	-
DSM-ES30-T10+BV_{m100w7}	2.92	0.478	2.77	0.198	-	-
DSM-ES30-T10+$BVLLP_{m3w7}$	2.69	0.233	2.54	0.155	5.23	0.193
DSM-ES30-T10+$BVLLP_{m50w7}$	**2.61**	**0.376**	**2.49**	**0.201**	**5.10**	**0.288**
DSM-ES30-T10+$BVLLP_{m100w7}$	2.60	0.443	2.48	0.219	5.08	0.331
DSM-ES10-T10+AD_{l4}	**2.31**	**0.647**	**2.30**	**0.451**	**4.61**	**0.549**
DSM-ES10-T10+AD_{l8}	2.19	1.105	2.16	0.769	4.35	0.936
DSM-ES10-T10+AD_{l25}	2.10	2.709	2.12	1.966	4.22	2.339
DSM-ES10-T10+AD_{l100}	2.06	9.725	2.09	6.725	4.15	8.233
k2part	-	-	-	-	**4.01**	**0.900**

Table 3.7: Compression performance in bpe and random access time in microseconds for neighbor queries, for indochina-2004. The best performing alternatives considering space and access time are in bold. We only include the best alternative for Direct/Reverse queries of DSM + BV which is DSM + BVLLP.

Technique	Direct		Reverse		Direct/Reverse	
	bpe	time (us)	bpe	time (us)	bpe	time (us)
$BVLLP_{m3w7}$	**1.78**	**0.098**	**1.38**	**0.077**	**3.16**	**0.087**
$BVLLP_{m50w7}$	1.49	0.543	1.28	0.225	2.77	0.384
$BVLLP_{m100w7}$	1.47	1.013	1.28	0.299	2.75	0.658
AD_{l4}	2.13	0.080	1.27	0.056	3.40	0.068
AD_{l8}	**1.49**	**0.107**	**1.11**	**0.082**	**2.60**	**0.094**
AD_{l25}	1.23	0.245	1.00	0.195	2.23	0.220
AD_{l100}	1.10	0.821	0.95	0.680	2.05	0.751
GB_{h16}	**1.62**	**0.218**	**1.45**	**0.353**	**3.07**	**0.286**
GB_{h32}	1.36	0.510	1.24	0.509	2.60	0.510
GB_{h64}	1.18	0.807	1.12	0.810	2.30	0.809
GB_{h128}	1.07	1.312	1.04	1.345	2.11	1.329
DSM-ES100-T10+BV_{m3w7}	1.48	0.124	1.35	0.078	-	-
DSM-ES100-T10+BV_{m50w7}	1.35	0.236	1.27	0.128	-	-
DSM-ES100-T10+BV_{m100w7}	1.35	0.317	1.26	0.134	-	-
DSM-ES100-T10+$BVLLP_{m3w7}$	1.31	0.150	1.25	0.107	2.56	0.128
DSM-ES100-T10+$BVLLP_{m50w7}$	**1.22**	**0.272**	**1.18**	**0.157**	**2.40**	**0.214**
DSM-ES100-T10+$BVLLP_{m100w7}$	1.22	0.341	1.18	0.171	2.40	0.256
DSM-ES15-T10+AD_{l4}	**1.16**	**0.303**	**1.04**	**0.232**	**2.20**	**0.277**
DSM-ES15-T10+AD_{l8}	1.04	0.475	0.93	0.378	1.97	0.440
DSM-ES15-T10+AD_{l25}	0.96	1.101	0.86	0.870	1.82	1.013
DSM-ES15-T10+AD_{l100}	0.92	3.685	0.83	2.901	1.75	3.296
k2part	-	-	-	-	**2.09**	**0.307**

Table 3.8: Compression performance in bpe and random access time in microseconds for neighbor queries, for uk-2002. The best performing alternatives considering space and access time are in bold. We only include the best alternative for Direct/Reverse queries of DSM + BV which is DSM + BVLLP.

Technique	Direct		Reverse		Direct/Reverse	
	bpe	time (us)	bpe	time (us)	bpe	time (us)
$BVLLP_{m3w7}$	**2.81**	**0.131**	**2.21**	**0.110**	**5.02**	**0.121**
$BVLLP_{m50w7}$	2.40	0.623	2.07	0.233	4.47	0.429
$BVLLP_{m100w7}$	2.37	1.120	2.06	0.276	4.43	0.699
AD_{l4}	3.37	0.120	2.20	0.085	5.57	0.102
AD_{l8}	**2.64**	**0.175**	**1.96**	**0.125**	**4.60**	**0.150**
AD_{l25}	2.11	0.369	1.79	0.303	3.90	0.336
AD_{l100}	1.93	1.250	1.73	1.036	3.66	1.142
GB_{h16}	**2.86**	**0.278**	**2.48**	**0.505**	**5.34**	**0.392**
GB_{h32}	2.27	0.450	2.10	0.724	4.37	0.589
GB_{h64}	1.92	0.793	1.88	1.107	3.80	0.951
GB_{h128}	1.71	1.405	1.77	1.798	3.48	1.603
DSM-ES100-T10+BV_{m3w7}	2.68	0.183	2.45	0.116	-	-
DSM-ES100-T10+BV_{m50w7}	2.44	0.396	2.32	0.183	-	-
DSM-ES100-T10+BV_{m100w7}	2.43	0.517	2.32	0.238	-	-
DSM-ES100-T10+$BVLLP_{m3w7}$	2.21	0.228	2.10	0.157	4.31	0.192
DSM-ES100-T10+$BVLLP_{m50w7}$	**2.07**	**0.372**	**1.99**	**0.223**	**4.06**	**0.298**
DSM-ES100-T10+$BVLLP_{m100w7}$	2.06	0.443	1.99	0.262	4.05	0.352
DSM-ES15-T10+AD_{l4}	**2.00**	**0.446**	**1.89**	**0.327**	**3.89**	**0.386**
DSM-ES15-T10+AD_{l8}	1.81	0.713	1.73	0.522	3.54	0.618
DSM-ES15-T10+AD_{l25}	1.67	1.551	1.60	1.224	3.27	1.387
DSM-ES15-T10+AD_{l100}	1.61	4.760	1.55	4.021	3.16	4.390
k2part	-	-	-	-	**3.40**	**0.542**

Table 3.9: Compression performance in bpe and random access time in microseconds for neighbor queries, for arabic-2005. The best performing alternatives considering space and access time are in bold. We only include the best alternative for Direct/Reverse queries of DSM + BV which is DSM + BVLLP.

Technique	Direct		Reverse		Direct/Reverse	
	bpe	time (us)	bpe	time (us)	bpe	time (us)
$BVLLP_{m3w7}$	**2.17**	**0.102**	**1.55**	**0.078**	**3.72**	**0.091**
$BVLLP_{m50w7}$	1.78	0.615	1.45	0.222	3.23	0.419
$BVLLP_{m100w7}$	1.76	1.667	1.44	0.268	3.20	0.721
AD_{l4}	2.86	0.096	1.80	0.063	4.66	0.079
AD_{l8}	**2.07**	**0.123**	**1.64**	**0.100**	**3.71**	**0.111**
AD_{l25}	1.82	0.318	1.53	0.243	3.35	0.280
AD_{l100}	1.66	1.008	1.51	0.844	3.17	0.924
GB_{h16}	**2.38**	**0.232**	**1.90**	**0.354**	**4.28**	**0.293**
GB_{h32}	1.83	0.400	1.60	0.544	3.43	0.472
GB_{h64}	1.49	0.813	1.42	0.892	2.91	0.853
GB_{h128}	1.29	1.356	1.31	1.534	2.60	1.446
DSM-ES100-T10+BV_{m3w7}	2.06	0.152	1.77	0.082	-	-
DSM-ES100-T10+BV_{m50w7}	1.90	0.310	1.70	0.123	-	-
DSM-ES100-T10+BV_{m100w7}	1.89	0.423	1.70	0.139	-	-
DSM-ES100-T10+$BVLLP_{m3w7}$	1.58	0.192	1.43	0.118	3.01	0.155
DSM-ES100-T10+$BVLLP_{m50w7}$	**1.49**	**0.328**	**1.37**	**0.152**	**2.86**	**0.240**
DSM-ES100-T10+$BVLLP_{m100w7}$	1.48	0.368	1.37	0.160	2.85	0.264
DSM-ES10-T10+AD_{l4}	**1.50**	**0.469**	**1.35**	**0.312**	**2.85**	**0.391**
DSM-ES10-T10+AD_{l8}	1.39	0.781	1.26	0.561	2.65	0.672
DSM-ES10-T10+AD_{l25}	1.31	1.822	1.20	1.399	2.51	1.609
DSM-ES10-T10+AD_{l100}	1.28	5.904	1.17	5.136	2.45	5.519
k2part	-	-	-	-	2.90	0.562

Table 3.10: Compression performance in bpe and random access time in microseconds for neighbor queries, for sk-2005. The best performing alternatives considering space and access time are in bold. We only include the best alternative for Direct/Reverse queries of DSM + BV which is DSM + BVLLP. Performance for Reverse and Direct/Reverse queries using AD and k2part do not appear because the corresponding software crashed.

Technique	Direct		Reverse		Direct/Reverse	
	bpe	time (us)	bpe	time (us)	bpe	time (us)
BVLLP$_{m3w7}$	**2.39**	**0.097**	**1.63**	**0.069**	**4.02**	**0.083**
BVLLP$_{m50w7}$	1.95	0.618	1.53	0.150	3.48	0.383
BVLLP$_{m100w7}$	1.92	1.141	1.53	0.229	3.45	0.692
AD$_{l4}$	2.91	0.084	x	x	x	x
AD$_{l8}$	**2.39**	**0.130**	x	x	x	x
AD$_{l25}$	1.99	0.294	x	x	x	x
AD$_{l100}$	1.86	1.050	x	x	x	x
GB$_{h16}$	**2.14**	**0.223**	**1.72**	**0.442**	**3.86**	**0.333**
GB$_{h32}$	1.82	0.475	1.54	0.579	3.36	0.528
GB$_{h64}$	1.59	0.712	1.43	0.905	3.02	0.805
GB$_{h128}$	1.43	1.242	1.36	1.465	2.79	1.355
DSM-ES100-T10+BV$_{m3w7}$	1.94	0.161	1.60	0.083	-	-
DSM-ES100-T10+BV$_{m50w7}$	1.83	0.280	1.53	0.106	-	-
DSM-ES100-T10+BV$_{m100w7}$	1.82	0.327	1.53	0.123	-	-
DSM-ES100-T10+BVLLP$_{m3w7}$	1.74	0.210	1.51	0.115	3.25	0.163
DSM-ES100-T10+BVLLP$_{m50w7}$	**1.66**	**0.365**	**1.47**	**0.147**	**3.13**	**0.256**
DSM-ES100-T10+BVLLP$_{m100w7}$	1.65	0.387	1.47	0.152	3.12	0.269
DSM-ES15-T10+AD$_{l4}$	**1.66**	**0.536**	**1.47**	**0.312**	**3.13**	**0.425**
DSM-ES15-T10+AD$_{l8}$	1.58	0.813	1.40	0.496	2.98	0.653
DSM-ES15-T10+AD$_{l25}$	1.52	2.056	1.34	1.311	2.86	1.691
DSM-ES15-T10+AD$_{l100}$	1.49	6.967	1.32	4.636	2.81	5.844
k2part	-	-	-	-	x	x

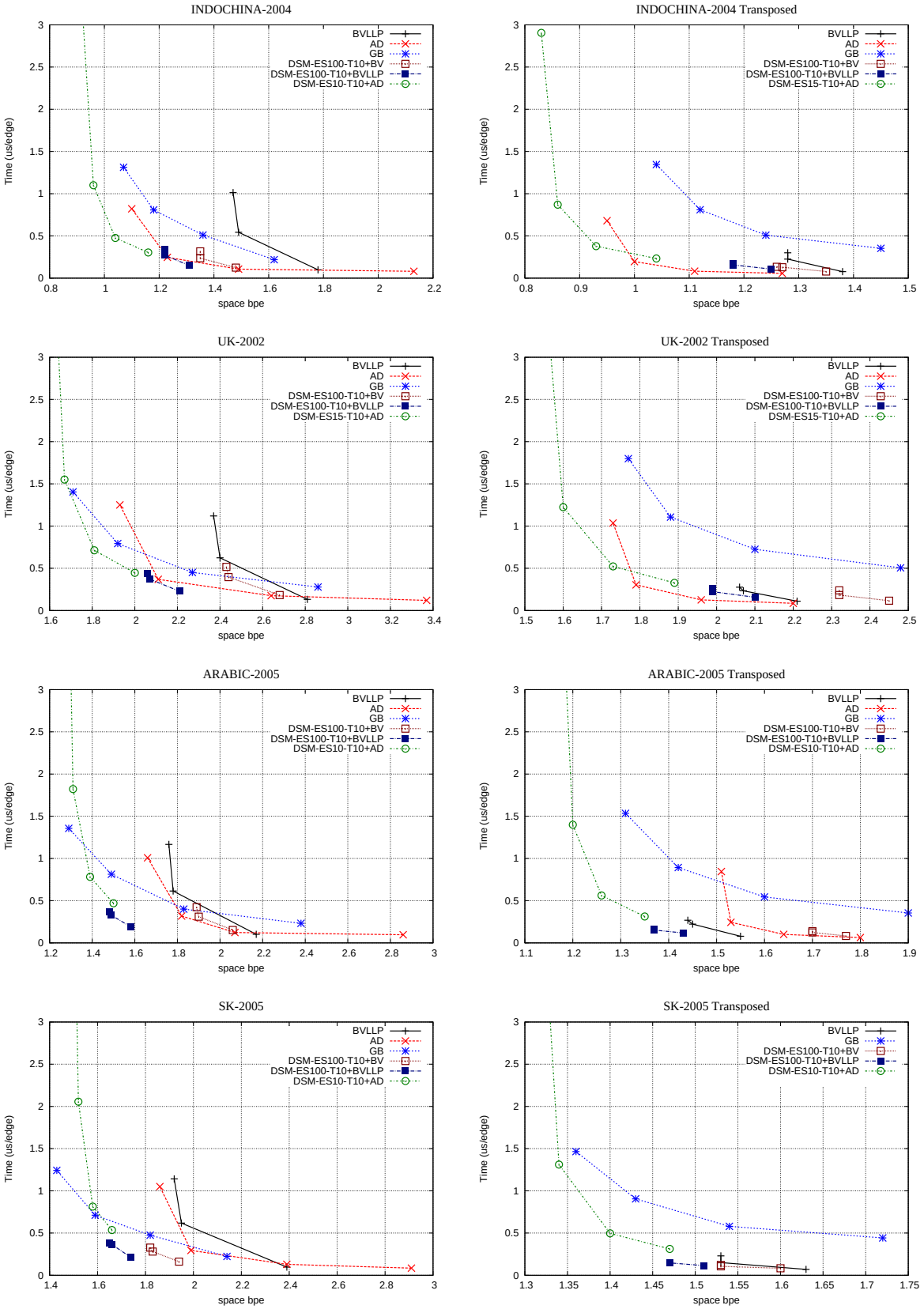


Figure 3.4: Space/time efficiency with out-neighbor queries for random access.

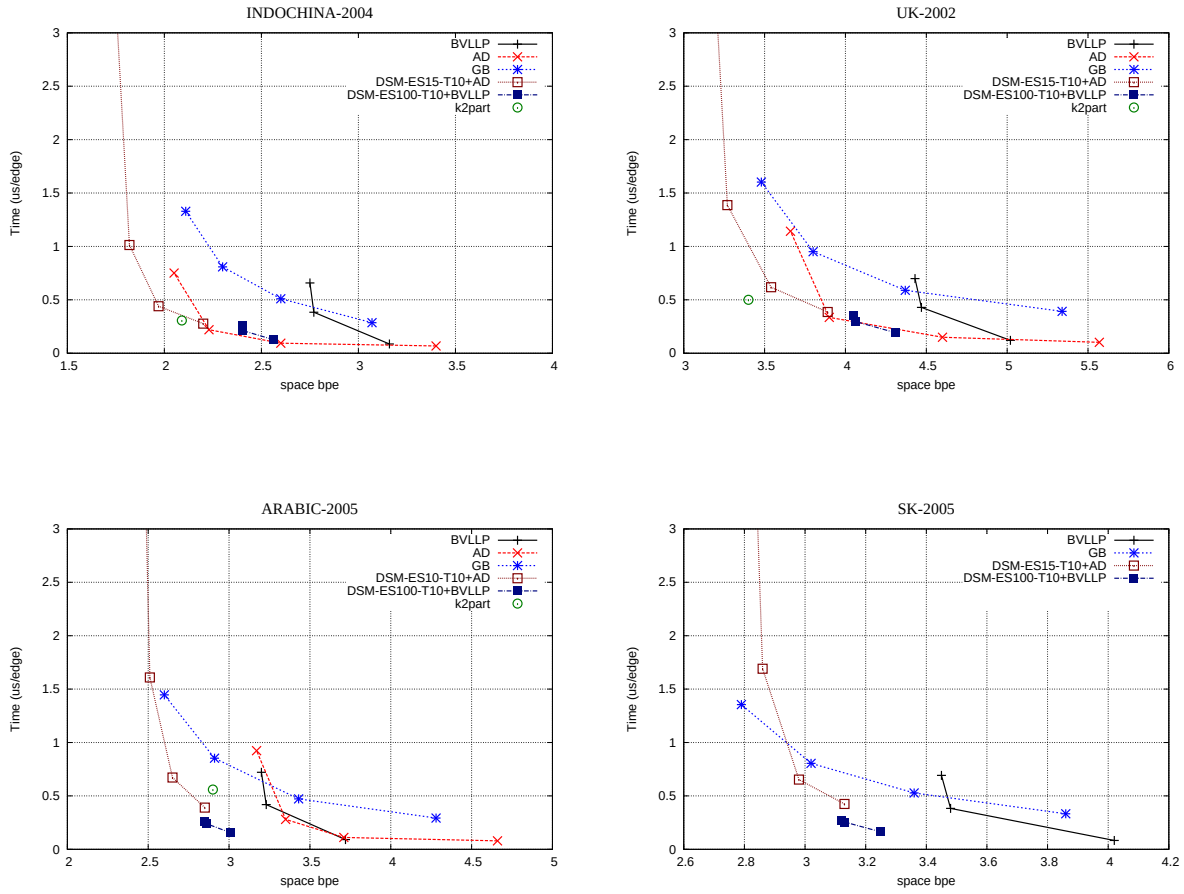


Figure 3.5: Space/time efficiency with out/in-neighbor queries using BV (WebGraph version 3.4.0), BVLLP (BV with LLP node ordering), AD, GB with direct and transposed graphs.

Figure 3.5 shows the space required to store the original plus the transposed graphs, combined with the time for out-neighbor queries (which is very similar to that for in-neighbor queries; these are run on the transposed graph). It can be seen that our new combinations of DSM plus AD dominate most of the space/time tradeoff, except on eu-2005. However, a data structure specific for out/in-neighbor queries (k2part [39]) offers comparable (and in some graphs much better) time performance, but we outperform it in space, considerably on some graphs.

Next we will consider a truly bidirectional representation for the reduced graph, obtaining much less space with higher query time.

Table 3.11: Compression performance when combining with k2tree. VNM-ESx-Ty + k2tree and DSM-ESx-Ty+k2tree refer to applying VNM or DSM and then k2tree over the edge reduced graph without reordering the nodes of such graph, whereas DSM-ESx-Ty + k2treeBFS refers to applying DSM and then reordering the nodes of the edge reduced graph using BFS before applying k2tree. Best compression results are in bold.

Dataset	eu-2005	indochina-2004	uk-2002	arabic-2005	sk-2005
k2treeNAT	3.45	1.35	2.77	2.47	2.82
k2treeBFS	3.22	1.23	2.04	1.67	1.91
VNM-ES10-T5 + k2tree	2.76	1.38	2.45	1.78	1.80
VNM-ES10-T10 + k2tree	2.71	1.34	2.35	1.70	1.75
DSM-ES10-T5 + k2tree	2.76	1.36	2.40	1.76	1.80
DSM-ES10-T10 + k2tree	2.71	1.34	2.40	1.76	1.78
DSM-ES15-T5 + k2tree	2.65	1.27	2.28	1.67	1.77
DSM-ES15-T10 + k2tree	2.59	1.27	2.27	1.66	1.76
DSM-ES100-T5 + k2tree	2.56	1.16	2.13	1.52	1.60
DSM-ES100-T10 + k2tree	2.48	1.14	2.08	1.47	1.58
DSM-ES10-T5 + k2treeBFS	2.21	0.90	1.56	1.12	1.35
DSM-ES10-T10 + k2treeBFS	**2.11**	**0.87**	**1.53**	**1.08**	1.30
DSM-ES15-T5 + k2treeBFS	**2.11**	**0.87**	1.54	1.14	1.32
DSM-ES15-T10 + k2treeBFS	2.21	0.89	1.57	**1.08**	**1.29**
DSM-ES100-T5 + k2treeBFS	2.54	0.95	1.67	1.21	1.39
DSM-ES100-T10 + k2treeBFS	2.45	0.93	1.64	1.18	1.37

3.3.3 Performance Evaluation with Out/In-neighbor Support

In this section we combine the output of DSM and virtual nodes with a compression technique that supports out/in-neighbor queries: the k2tree [23]. We use the best current implementation [23]. We apply dense subgraph discovery with parameters $ES = 10, 15, 100$ and $T = 5, 10$. In all cases process DSM is run over the graph in natural order. We denote k2treeBFS the variant that switches to BFS order on $G3$ when applying the k2tree representation, and k2tree the variant that retains natural (URL-based) order.

Table 3.11 shows the compression achieved. We observe that the compression ratio is markedly better when using BFS ordering. In particular the setting $ES = 10$, $T = 10$ and k2treeBFS is always the best. The space is also much better than that achieved by representing the original plus transposed graphs in Section 3.3.2. We do not show results for social networks because this scheme does not work well on them. The next chapter describes a successful approach for social networks that uses bicliques and dense subgraphs represented with compact data structures.

Figure 3.6 shows the space/time tradeoff when solving out-neighbor queries (in-neighbor times are very similar). We include k2treeNAT [22], k2treeBFS [23], k2part [39], and disregard other structures that have been superseded by the last k2tree improvements [41]. We also include in the plots one choice DSM-ESx-Ty+AD from Section 3.3.2, which represents

the direct and transposed graphs using DSM and $T = 10$ combined with AD using various values of l. Finally, we also include the alternative DSM-ESx-Ty+BVLLP from Section 3.3.2, for accessing direct and reverse access because this option provides the best access times and compression is only slightly worse than the alternative with AD for the largest graphs.

All those structures are clearly superseded in space by our new combinations of DSM and k2treeBFS or k2tree. Again, the combination with BFS gives much better results, and using different ES values yields various space/time tradeoffs. On the other hand, these smaller representations reaching 0.9–1.6 bpe on the larger graphs are also significantly slower, requiring 5–20 μsec per retrieved neighbor.

3.4 Conclusions

This chapter presents two schemes for compressing Web graphs and supporting out-neighbor and out/in-neighbor navigation. Both schemes are based on factoring edges of the original graph using virtual nodes and applying other compression techniques over the reduced graph. The first scheme uses bicliques ($S \cap C = \emptyset$) and the second uses dense subgraphs ($S \cap C \neq \emptyset$). We show that both schemes provide competitive state-of-the-art compression efficiency for out-neighbor navigation when using BVLLP [16], and AD [9] over the reduced graph. We also show that using dense subgraphs is slightly better than using bicliques. We observe that applying BVLLP over the graph after factoring edges, that is, applying LLP node ordering over the edge reduced graph and then the compression technique, improves compression and provides very competitive random access time. On the other hand, applying AD [9] over the reduced graph provides the best compression performance, but random access time is higher than other alternatives.

In the context of out/in-neighbor navigation, we show that applying BFS node ordering and k2tree [22] (DSM-ESx-Ty-k2treeBFS) over the reduced graph (using dense subgraphs) provides the best compression efficiency, although about twice as slow as the second best compression approach (k2treeBFS).

We observe that the main characteristics that made Web graph compression methods succeed, are much less pronounced in social networks. In particular, we found that our algorithm for factoring edges based on dense subgraphs do not work well on social networks, where the compression rate based on the number of edges after of the factoring is much lower (between 1.18 and 1.32 in enron) than on Web graphs (between 5.26 and 9.21).

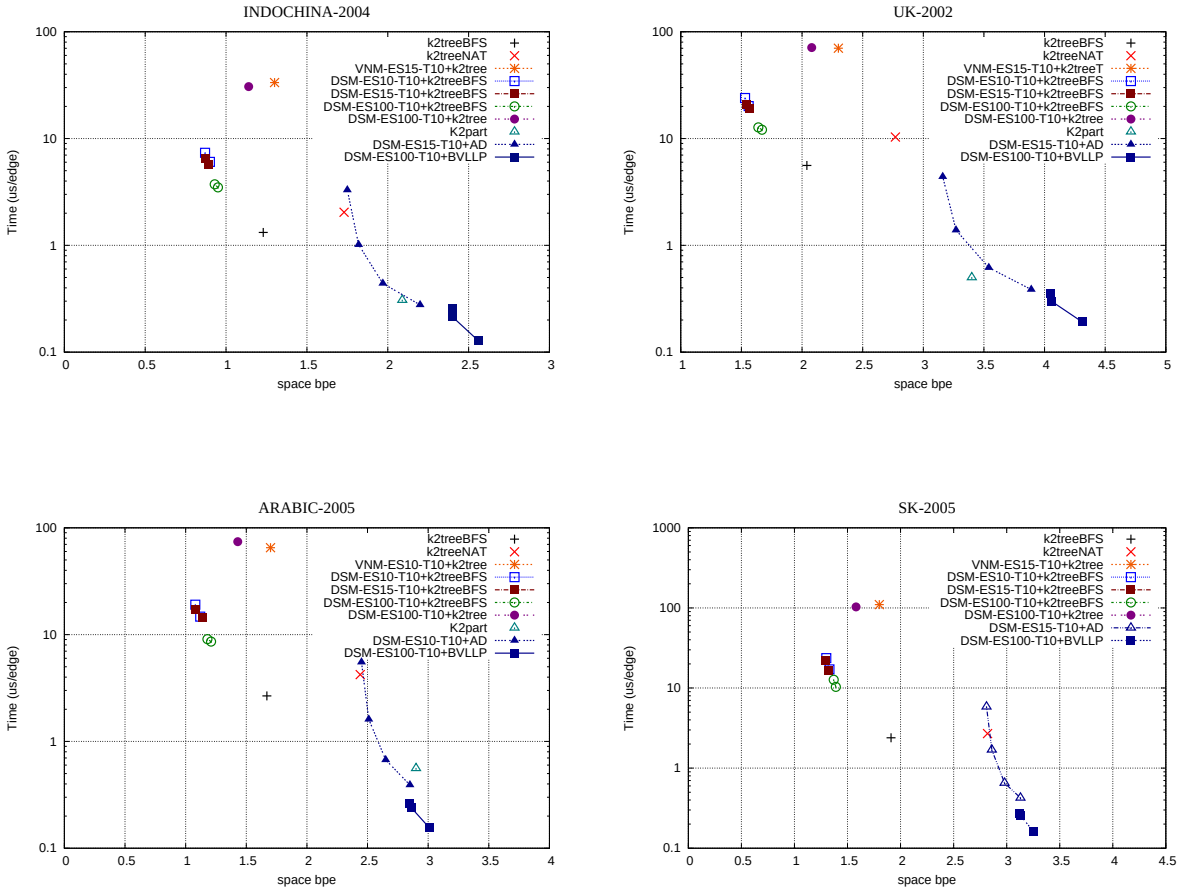


Figure 3.6: Space/time efficiency with out/in-neighbor queries using k2tree with natural (k2treeNAT) and BFS (k2treeBFS) node ordering. DSM-ESx-Ty+k2tree and VNM-ESx-Ty+k2tree refer to applying k2tree over the reduced graph without reordering nodes, whereas in DSM-ESx-Ty+k2treeBFS and VNM-ESx-Ty+k2treeBFS alternatives nodes of the reduced graph are reordered with BFS before applying k2tree. DSM-ESx-Ty+AD and DSM-ESx-Ty+BVLLP are the corresponding alternatives for in/out-neighbor access from Section 3.3.2.

Chapter 4

Representing Edges Implicitly

This chapter describes two compressed graph schemes representing edges implicitly using compact data structures based on bitmaps and symbol sequences (described in Chapter 1). Both structures support out/in-neighbor navigation as well as various mining queries.

We first present a compact representation using bicliques (i.e. $S \cap C = \emptyset$). In this case, our goal is to represent the $|S| \cdot |C|$ edges of a biclique $H(S, C)$ in space proportional to $|S| + |C|$.

We present a second representation that improves previous structures using dense subgraphs where the overlapping between sets S and C is allowed (i.e., $S \cap C \neq \emptyset$). Our goal here is to represent the $|S| \cdot |C|$ edges of a dense subgraph $H(S, C)$ in space proportional to $|S| + |C| - |S \cap C|$.

These representations will not use virtual nodes, and their output is no longer a graph. As a result, we cannot iterate on the discovery algorithm in order to find dense subgraphs involving virtual nodes. Instead, the subgraphs are extracted iteratively from the graph, forming a collection that is represented with bitmaps and sequences. Both structures obtain better results by extracting bigger subgraphs first.

4.1 Extracting Bicliques

We extract bicliques using the algorithms described in Chapter 2, but we do not add self-loops to each adjacency list as described in that chapter. We use three parameters: P, the number of hashes in the clustering stage of the discovery, a list of ES values, where ES is the minimum $|S| \cdot |C|$ size of bicliques found, and *threshold*. Parameters P and ES are the same as before, yet now we use a decreasing list of ES values. The discovery algorithm continues extracting subgraphs of a size ES_i until the number of subgraphs drops below the *threshold* on a single iteration; then ES is set to the next value in the list for the next iteration. Here the number of iterations will depend on the number of extracted subgraphs on each iteration and the *threshold* value. The goal of having the ES list in decreasing order is to avoid that

extracting a small biclique precludes the identification of a larger one, which gives a higher benefit.

4.2 Representing the Graph Using Bicliques

After we have extracted all the interesting bicliques from $G(V, E)$, we represent G as the set of bicliques plus a *remaining* graph. The compact representation is based on the following definitions:

Definition 4.2.1 *Directed (Undirected) Bipartite Partition, DBP (UBP).* Let $G(V, E)$ be directed (undirected). A bipartite partition of G consists of a class $H = \bigcup H_r$ of bipartite graphs $H_r = H(S_r, C_r)$, and a *remaining* graph $R(V_R, E_R)$, so that all H_r and R are edge-disjoint and $G = H \cup R$.

Definition 4.2.2 *Undirected plus Directed Bipartite Partition, UDBP.* Let $G(V, E)$ be a directed graph. We derive from G an undirected graph $G_u(V_u, E_u)$, containing an undirected edge per pair of reciprocal edges in G. Now consider the *UBP* of G_u into H_u and $R_u(V_{R_u}, E_{R_u})$. Define $dup(E)$ as the set of directed edges formed by a pair of reciprocal edges per undirected edge in E. Then we call $G_d(V_d, E - dup(E_u - E_{R_u}))$ the remaining directed graph of G. Now consider the *DBP* of G_d into H_d and R_d. The *UDBP* of G is formed by H_u, H_d, and R_d.

Definition 4.2.3 We define the *density* of a dense subgraph considering the connections inside a group [5]. In this context, $H(W, E)$ is defined to be γ-dense if $\frac{|E|}{|W|(|W|-1)/2} \geq \gamma$ where $W = S \cup C$

Note that *DBP* and *UBP* aim at representing a graph as a set of bicliques, regarded as directed or undirected. A large biclique $H(S, C)$ will allow us to replace $|S| \cdot |C|$ edges by just $|S| + |C|$ edges and a new node. *UDBP* is more sophisticated, and aims to exploit *reciprocity* in directed graphs, that is, reciprocal edges. It first looks for reciprocal bicliques, and only then for directed bicliques. Whether *UDBP* is better or worse than plain *DBP* on a directed graph will depend on its degree of reciprocity.

4.2.1 Compact Representation of *H*

Let $H = \{H_1, \ldots, H_N\}$ be the bicliques found in either of the previous definitions. We represent H as two sequences of integers with corresponding bitmaps. Sequence X_s with bitmap B_s represent the sequence of sources of the communities and sequence X_c with bitmap B_c represent the respective centers. More precisely, we have $X_s = x_s(1)x_s(2)...x_s(r)...x_s(N)$, where $x_s(r) = s_1...s_k$ represents the set S_r of $H_r = (S_r, C_r)$, $s_i \in S_r$, $s_i < s_{i+1}$ for $1 \leq i < k$, and $B_s = 10^{|S_1|-1}...10^{|S_N|-1}1$. In a similar way, we have $X_c = x_c(1)x_c(2)...x_c(r)...x_c(N)$, where $x_c(r) = c_1...c_m$ represents the set C_r, $c_j \in C_r$, $c_j < c_{j+1}$ for $1 \leq i < m$, and $B_c =$

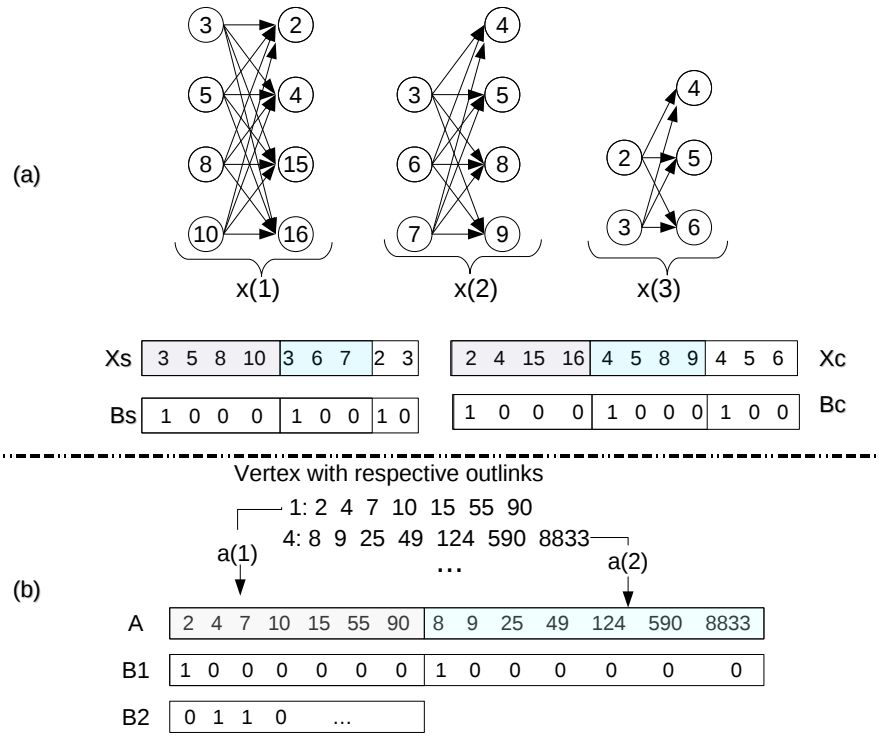


Figure 4.1: Graph and compact representation.

$10^{|C_1|-1}...10^{|C_N|-1}1$. Figure 4.1 shows an example, where (a) shows the compact representation of the set of bicliques (H).

We represent integer sequences and bitmaps with compact data structures that support rank/select/access operations: we use Wavelet Trees (WTs) [67], using the implementation without pointers, for sequences and an uncompressed representation [37] for bitmaps, for a total space of $|X|(H_0(X)+1)+o(|X|\log\sigma)$, where σ is the number of vertices in H. Note that $|X|$ is the sum of the sizes of the communities in H, which can be much less than the number of edges in the subgraph it represents.

We answer out/in-neighbor queries with the algorithms defined in Figure 4.2. Their complexity is $O((|output|+1)\log\sigma)$, which is essentially optimal up to a factor of $O(\log\sigma)$. Figure 4.3 presents the algorithms for getting the *centers* and *sources* of a given vertex x. Figure 4.4 provides the algorithms for knowing if a vertex x participates as a *center* or a *source*, Figure 4.5 gives the algorithm for getting just the number of bicliques where vertex x participates as a *source* or *center*, Figure 4.6 gives the algorithms for listing the members of community x and the bicliques at a $distance = 1$, and Figure 4.7 gives the algorithms for enumerating all biliques with corresponding sizes and their densities.

Table 4.1 gives the time complexities achieved. Most of them are, again, optimal up to factor $O(\log\sigma)$. The exception is Q7, which can be costlier due to repeated results.

out-neighbors (u)
$occur \leftarrow rank_{X_s}(u, |X_s|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_{X_s}(u, i)$
  $p \leftarrow rank_{B_s}(1, y)$
  $s \leftarrow select_{B_c}(1, p)$
  $e \leftarrow select_{B_c}(1, p+1) - 1$
  for $j \leftarrow s$ **to** e **do**
    $out.add(access_{X_c}(j))$
  end for
end for

in-neighbors (u)
$occur \leftarrow rank_{X_c}(u, |X_c|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_{X_c}(u, i)$
  $p \leftarrow rank_{B_c}(1, y)$
  $s \leftarrow select_{B_s}(1, p)$
  $e \leftarrow select_{B_s}(1, p+1) - 1$
  for $j \leftarrow s$ **to** e **do**
    $in.add(access_{X_s}(j))$
  end for
end for

Figure 4.2: Algorithms for out/in-neighbors.

Q1: Get the *centers* of x.
$start \leftarrow select_{B_c}(1, x)$
$end \leftarrow select_{B_c}(1, x+1) - 1$
for $i \leftarrow start$ **to** end **do**
  $centers.add(access_{X_c}(i))$
end for

Q2: Get the *sources* of x.
$start \leftarrow select_{B_s}(1, x)$
$end \leftarrow select_{B_s}(1, x+1) - 1$
for $i \leftarrow start$ **to** end **do**
  $sources.add(access_{X_s}(i))$
end for

Figure 4.3: Algorithms for getting *Centers* and *Sources* of x.

Q3: Get bicliques where u participates as a source.
$occur \leftarrow rank_{X_s}(u, |X_s|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_{X_s}(u, i)$
  $p \leftarrow rank_{B_s}(1, y)$
  $comms.add(p)$
end for

Q4: Get bicliques where u participates as a center.
$occur \leftarrow rank_{X_c}(u, |X_c|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_{X_c}(u, i)$
  $p \leftarrow rank_{B_c}(1, y)$
  $comms.add(p)$
end for

Figure 4.4: Algorithms for getting bicliques where x participates as a source or center.

Q5: Get the number of bicliques where u participates as a source and center.
$ncs \leftarrow rank_{X_s}(u, |X_s|)$
$ncc \leftarrow rank_{X_c}(u, |X_c|)$

Figure 4.5: Algorithm for getting number of bicliques where x participates.

Q6: Enumerate the members of x.
$ss \leftarrow select_{B_s}(1, x)$
$es \leftarrow select_{B_s}(1, x+1) - 1$
for $i \leftarrow ss$ **to** es **do**
  $members.add(access_{X_s}(i))$
end for
$sc \leftarrow select_{B_c}(1, x)$
$ec \leftarrow select_{B_c}(1, x+1) - 1$
for $i \leftarrow sc$ **to** ec **do**
  $members.add(access_{X_c}(i))$
end for

Q7: Enumerate the bicliques at distance 1 of x.
$centers \leftarrow Q1(x)$
for c in $centers$ **do**
  $comms.add(Q3(c))$
end for

Figure 4.6: Algorithms for listing the members of x, and the bicliques at distance 1.

Q8: Enumerate all bicliques with their sizes.
$nc \leftarrow rank_{B_s}(1, |B_s|)$
for $i \leftarrow 1$ **to** nc **do**
  $ss \leftarrow select_{B_s}(1, i+1) - select_{B_s}(1, i)$
  $sc \leftarrow select_{B_c}(1, i+1) - select_{B_c}(1, i)$
  $size\_list.add(ss + sc)$
end for

Q9: Enumerate all bicliques with their densities.
$nc \leftarrow rank_{B_s}(1, |B_s|)$
for $i = 1$ **to** nc **do**
  $sources \leftarrow select_{B_s}(1, i+1) - select_{B_s}(1, i)$
  $centers \leftarrow select_{B_c}(1, i+1) - select_{B_c}(1, i)$
  $edges \leftarrow sources \cdot centers$
  $nodes \leftarrow sources + centers$
  $densities.add(\frac{edges}{nodes \cdot (nodes-1)/2})$
end for

Figure 4.7: Algorithms for enumerating bicliques with sizes and densities.

out-neighbors (u)
if $access_{B2}(u) = 0$ **then**
  $x \leftarrow rank_{B2}(0, u)$
  $start \leftarrow select_{B1}(1, x)$
  $end \leftarrow select_{B1}(1, x + 1) - 1$
  for $i \leftarrow start$ **to** end **do**
    $out.add(access_A(i))$
  end for
end if

in-neighbors (u)
$occur \leftarrow rank_A(u, |A|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_A(u, i)$
  $p \leftarrow rank_{B1}(1, y)$
  $in.add(select_{B2}(0, p)$
end for

Figure 4.8: Algorithms getting out/in-neighbors of x in R.

Query	Time complexity
Q1/Q2	$O(\|output\| \cdot \log \sigma)$
Q3/Q4	$O((\|output\| + 1) \log \sigma)$
Q5	$O(\log \sigma)$
Q6	$O(\|output\| \log \sigma))$
Q7	$O(\|output\| \log \sigma) \ldots O(\|Q1\| \cdot \|output\| \log \sigma)$
Q8/Q9	$O(\|output\|)$

Table 4.1: Time complexity for community queries.

Compact Representation of R

We define a sequence of integers A and two bitmaps $B1$ and $B2$ for representing $R(V_R, E_R)$. Sequence A is defined as $A = a(1)...a(i)...a(N)$, where $|A| = |E_R|$, $a(i)$ is the i-th nonempty direct adjacency list of R, and N is the total number of vertices with at least one edge in R. Bitmap $B1$ is $10^{|a(1)|-1}...10^{|a(N)|-1}$, so $|B1| = |E_R|$. $B2$ is a bitmap such that $B2[i] = 1$ iff vertex i does not have out-neighbors and $|B2| = |V_R|$. Figure 4.1 (b) shows an example. The space using WTs is $|A|(H_0(A) + 1) + |\sigma| + o(|A| \log \sigma)$, where $\sigma = |V_R|$. We answer neighbor queries on R as described in Figure 4.8.

On directed graphs, in-neighbors and out-neighbors are found in time $O((|output| + 1) \log \sigma)$. On undirected graphs we choose arbitrarily to represent each edge $\{u, v\}$ as (u, v) or (v, u). Consequently, finding the neighbors of a node requires carrying out both algorithms.

To carry out out/in-queries on the whole graph, we must query H and R (for *UBP* or *DBP* partitions) or H_u, H_d and R_d (for *UDBP* partitions), and merge the results. Biclique queries are carried out only on H (or H_u and H_d for *UDBP*, then merging the results). Our pseudocodes on X and B addressed the directed case. Those for communities representing undirected graphs are very easy to derive.

4.2.2 Space/time Evaluation

We evaluate our compact data structures supporting out/in-neighbor and community queries. We are interested in space/time requirements in terms of the community graph H (or H_u+H_d), the remaining graph R, and the complete graph $G = H \cup R$.

We compare space/time efficiency using the representations below. We refer as WT-N-b to representing sequence X with wavelet trees and bitmaps with RG [64], and as WT-N-r to using wavelet trees for X and bitmaps compressed with RRR [99]. N is the sampling parameter used for bitmap implementations (if left as a variable, it gives a space/time tradeoff). We will not give the results for using GMR [63] on H because the space achieved is not competitive.

T4 WT-N-b (H) + Re-Pair GMR (R)
T5 WT-N-r (H) + Re-Pair GMR (R)
T6 WT-N-b (H) + k2tree (R)
T7 WT-N-r (H) + k2tree (R)
T8 WT-N-b (H) + WT-64-b (R)
T9 WT-N-b (H) + WT-64-r (R)
T10 WT-N-r (H) + WT-64-b (R)
T11 WT-N-r (H) + WT-64-r (R)

We use the definitions given on this Section over Web graphs and social networks of Table 2.1. We use *UBP* with reciprocal edges to represent the (undirected) dblp-2011 graph, *DBP* on Web graphs and Enron, and *UDBP* for LiveJournal graphs.

Techniques T4–T11 support biclique queries on H and out/in-neighbor queries on $H + R$. We measure compression on G by computing $bpe = \frac{bits(H)+bits(R)}{edges(H)+edges(R)}$ and access time $query\_time = query\_time(H) + query\_time(R)$.

Table 4.2 shows the compression efficiency using the best schemes for Web graphs and Table 4.3 shows the efficiency in social networks.

We compute $bpe(H) = \frac{bits(H)}{edges(H)}$. When using *UDBP* we show $bpe(H_u)$ and $bpe(H_d)$ separately. We observe higher compression on Web graphs than on social networks and better results using compressed bitmaps (WT-64-r). We also show time efficiency for neighbor and different biclique queries in Table 4.4 (for *UDBP* the times for H_u and H_d must be added together). As it can be seen, *all biclique queries are supported within a few microseconds.*

Figure 4.9 shows the space and time on Web graphs and social networks, considering both partitions H (or $H_u + H_d$) and R, and out/in-neighbor queries. On Web graphs, we include the results using k2treeNAT (T2) and VNM + k2tree (T3), recall Figure 3.3. While in general T2 and T3 dominate the space/time tradeoff, variant T7 (WTs on H and k2tree on R) is competitive in some cases. Nevertheless, we remind that T4–T11 additionally support biclique queries.

On social networks, on the other hand, we achieve better results using techniques T4–T11

Table 4.2: Compression performance for Web graphs in bpe, compared to other techniques. WT-64-r(H) refers to using sequences represented by Wavelet Trees (without pointers) using compressed bitmaps (with sampling 64). Best compression results are in bold.

Compression	eu-2005	uk-2002	arabic-2005
k2treeNAT	3.45	2.77	2.47
VNM-ES10-T10+k2tree	**2.71**	**2.35**	**1.70**
WT-64-r(H) + k2tree(R)	3.28	3.10	2.30

Table 4.3: Compression performance for social networks in bpe, compared to other techniques. WT-64-r(H) refers to using sequences represented by Wavelet Trees (without pointers) using compressed bitmaps (with sampling 64). Best compression results are in bold.

Compression	enron	dblp-2011	LiveJournal-SNAP
k2treeLLP	10.31	9.83	17.35
MPk	17.02	8.48	**13.25**
WT-64-r(H)+k2tree(R)	**10.15**	9.90	16.81
WT-64-r(H) + MPk(R)	15.51	**8.45**	13.41
WT-64-r(H)+WT-64-r(R)	10.42	12.10	15.20

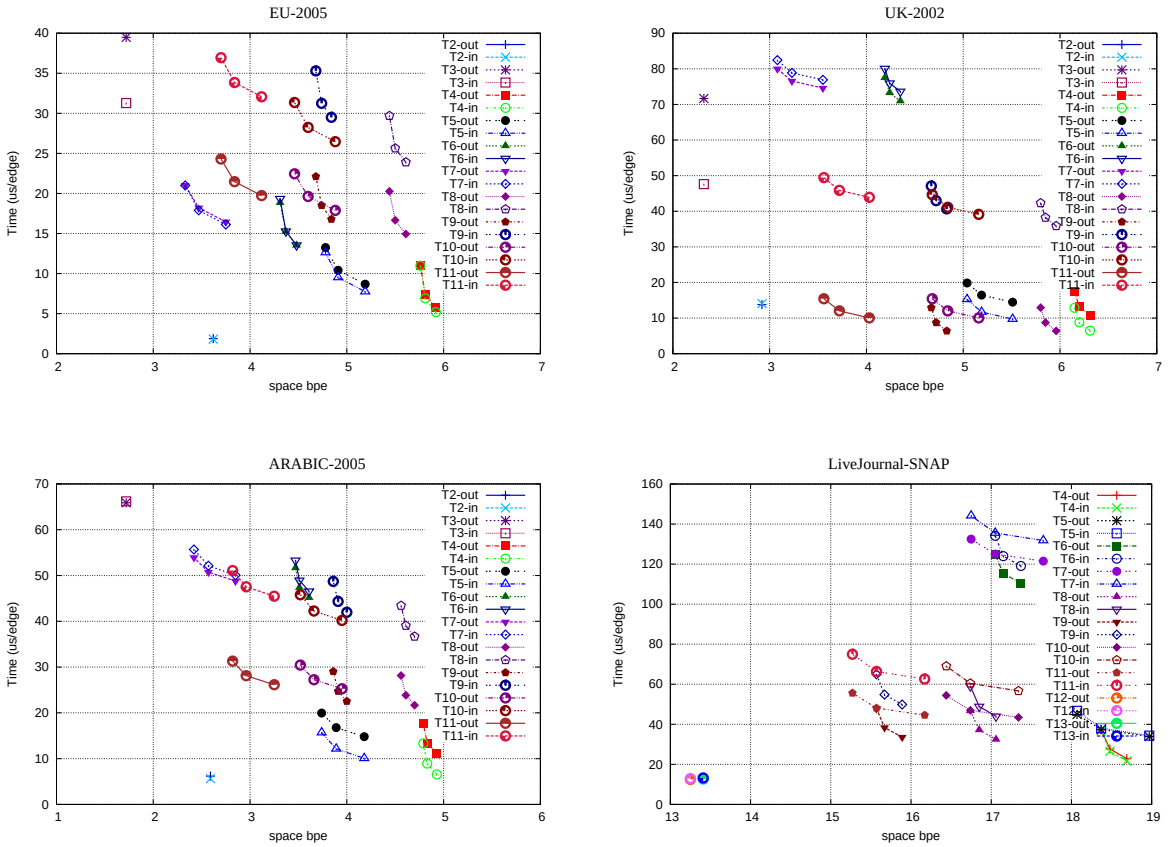


Figure 4.9: Space/time efficiency with out/in-neighbors queries on H+R (using only bicliques) for representations T2–T11.

Table 4.4: Times for biclique queries (*usecs*) on Web graphs and social networks representing H with compact data structures.

Queries	eu-2005	uk-2002	arabic-2005	LiveJournal Directed	LiveJournal Undirected
Out WT-32-b	6.70	8.71	8.62	12.69	10.16
Out WT-32-r	9.66	12.02	12.03	18.22	14.39
In WT-32-b	6.84	8.62	8.68	13.01	13.25
In WT-32-r	9.44	11.51	11.89	18.88	18.93
Q3 WT-32-b	3.15	6.07	5.12	2.56	2.47
Q3 WT-32-r	3.53	6.87	6.05	2.77	2.59
Q4 WT-32-b	1.99	3.69	2.42	2.12	2.25
Q4 WT-32-r	2.39	4.40	3.04	2.59	2.71
Q5 WT-32-b	2.22	4.10	2.78	3.21	3.27
Q5 WT-32-r	2.61	4.86	3.43	3.68	3.80
Q6 WT-32-b	7.17	8.69	10.14	8.81	7.85
Q6 WT-32-r	8.83	11.79	13.18	11.75	10.46
Q7 WT-32-b	119.21	147.88	233.56	74.71	58.24
Q7 WT-32-r	163.02	199.47	303.11	90.70	78.95

than using techniques T1–T3 (we include T2, the best performing technique of Figure 3.3, in Figure 4.9). Variant T11 provides the least space, whereas variants T9 and T4 provide other relevant space/time tradeoffs. However, the best space compression performance is not clear for social networks, since as seen in Table 4.3, it depends on the graph. MPk (we used the improved implementation of Claude and Ladra [39]) is the best for LiveJournal-SNAP, WT-64-r(H)+MPk(R) is the best for dblp-2011, and WT-64-r(H)+k2tree(R) for enron. Figure 4.9 also shows the space and time on LiveJournal-SNAP, where MPk and WT-64-r(H)+MPk(R) provide the best performance.

4.3 Extracting Dense Subgraphs

In this section we extract dense subgraphs using the algorithms described in Section 2 (i.e. dense subgraphs, which are described by bicliques with set overlaps, $S \cap C \neq \emptyset$). In this case, we add self-loops edges (i.e. edges of type (u, u)) on each adjacency list before applying the discovery algorithm to be able to capture cliques. We use the same parameters as the ones we use when we extract bicliques (ES, P, and *threshold*).

4.4 Representing the Graph using Dense Subgraphs

After we have extracted all the interesting dense subgraphs from $G(V, E)$, we represent G as the set of dense subgraphs plus a *remaining* graph.

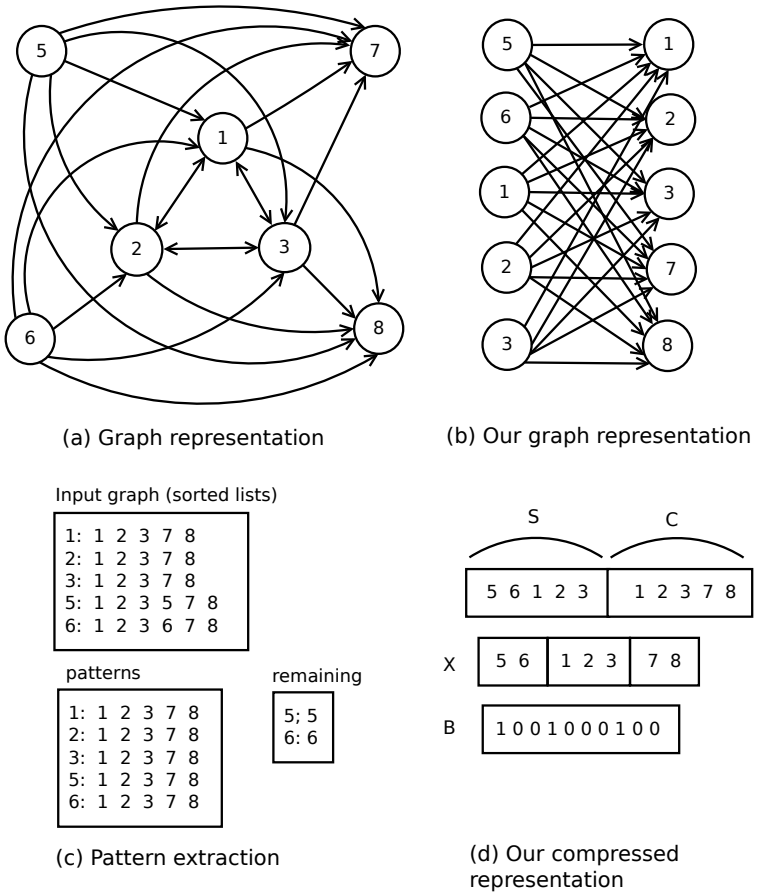


Figure 4.10: Dense subgraph representation

Definition 4.4.1 Let $G(V, E)$ be a directed graph, and let $H(S_r, C_r)$ be edge-disjoint dense subgraphs of G. Then the corresponding *dense subgraph* representation of G is (H, R), where $H = \{H(S_1, C_1), \ldots, H(S_N, C_N)\}$ and $R = G - \bigcup H(S_r, C_r)$ is the remaining graph.

Figure 4.10(a) shows the adjacency list representation for the graph presented in Figure 2.4, where we have already added the self-loops. We also show a dense subgraph, and a remaining subgraph. Figure 4.10(d) shows our compact representation.

Compact Representation of *H*

Let $H = \{H_1, \ldots, H_N\}$ be the dense subgraph collection found in the graph, based on Definition 4.4.1. We represent H as a sequence of integers X with a corresponding bitmap B. Sequence $X = X_1 : X_2 : \ldots : X_N$ represents the sequence of dense subgraphs and bitmap $B = B_1 : B_2 : \ldots B_N$ is used to mark separations in each subgraph. We now describe how a given X_r and B_r represent the dense subgraph $H_r = H(S_r, C_r)$.

Input: Subsets $S_1 \dots S_N$ and $C_1 \dots C_N$
Output: Sequence X and Bitmap B
$X \leftarrow \varepsilon$
$B \leftarrow \varepsilon$
for $i \leftarrow 0$ **to** N **do**
  $L \leftarrow S_i - C_i$
  $M \leftarrow S_i \cap C_i$
  $R \leftarrow C_i - S_i$
  $X \leftarrow X : L : M : R$
  $B \leftarrow B : 10^{|L|}10^{|M|}10^{|R|}$
end for
return X, B

Figure 4.11: Algorithm for building sequence X and bitmap B

We define X_r and B_r based on the overlapping between the sets S and C. Sequence X_r will have three components: L, M, and R, written one after the other in this order. Component L lists the elements of $S-C$. Component M lists the elements of $S \cap C$. Finally, component R lists the elements of $C - S$. Bitmap $B_r = 10^{|L|}10^{|M|}10^{|R|}$ gives alignment information to determine the limits of the components. In this way, we avoid repeating nodes in the intersection, and have sufficient information to determine all the edges of the dense subgraph. Figure 4.10(d) shows this representation for our example, which has just one dense subgraph. Algorithm in Figure 4.11 describes how X and B are built.

We compress the graph $G = H \cup R$, using sequence X and bitmap B for H. For R we use some bidirectional compressed graph representation.

To support our query algorithms, X and B are represented with compact data structures for sequences that implement *rank/select/access* operations. We use WTs [67] for sequence X and compressed bitmap representation RRR [99] for bitmap B. The total space is $|X|H_0(X) + o(|X| \log \sigma) + |X|H_0(B)$ bits, where $\sigma \leq |V|$ is the number of vertices in subgraph H. The $|X|H_0(X) + o(|X| \lg \sigma)$ owes to the wavelet tree representation, whereas $|X|H_0(B) + o(|X|)$ owes to the bitmap B. Note that $|X|$ is the sum of the number of nodes of the dense subgraphs in H, which can be much less than the number of edges in the subgraph it represents.

4.4.1 Neighbor Queries

We answer out/in-neighbor queries as described by algorithms in Figures 4.12 and 4.13. Their complexity is $O((|output| + 1) \log \sigma)$, which is away from optimal by a factor $O(\log \sigma)$. To exemplify the treatment of (u, u) edges, these algorithms always remove them before delivering the query results (as explained, more complex management is necessary if the graph actually contains some of those edges). Note this finds only the edges represented in component H; those in R must be also extracted, using the out/in-neighbor algorithm provided by the representation we have chosen for it.

Input: Sequence X, Bitmap B and vertex u
Output: List of out-neighbors of u
$out \leftarrow \varepsilon$
$occur \leftarrow rank_X(u, |X|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_X(u, i)$
  $p \leftarrow select_B(0, y + 1)$
  $o \leftarrow p - y$ { $= rank_B(1, p)$ }
  $m \leftarrow o \bmod 3$
  if $m = 1$ **then**
    $s \leftarrow select_B(1, o + 1) - (o + 1) + 1$
    $e \leftarrow select_B(1, o + 3) - (o + 3)$
  else if $m = 2$ **then**
    $s \leftarrow select_B(1, o) - o + 1$
    $e \leftarrow select_B(1, o + 2) - (o + 2)$
  else
    $s \leftarrow 1$
    $e \leftarrow 0$
  end if
  for $j \leftarrow s$ **to** e **do**
    $d \leftarrow access_X(j)$
    if ($d \neq u$) **then**
      $out \leftarrow out : d$
    end if
  end for
end for
return out

Figure 4.12: Algorithm for getting out-neighbors of u

We explain how the out-neighbors algorithm works; the case of in-neighbors is analogous. Using $select_X(u, i)$ we find all the places where node u is mentioned in X. This corresponds to some X_r, but we do not now where. Then we analyze B to determine whether this occurrence of u is inside component L, M, or R. In cases L and M, we use B again to delimit components M and R, and output all the nodes of X_r in those components. If u is in component R, instead, there is nothing to output in the case of out-neighbor queries.

An interesting advantage of our compressed structure is that it enables the retrieval of the actual dense subgraphs found on the graph. For instance, we are able to recover cliques and bicliques in addition to navigating the graph. Algorithm in Figure 4.14 shows how easy it is to recover all cliques and bicliques stored in the compressed structure. This information can be useful for mining and analyzing Web and social graphs. The time complexity is $O(|output| \cdot \log \sigma)$.

Note that we only report, in this simplified algorithm, pure cliques and bicliques. A slight modification would make the algorithm extract the clique $S \cap C$ that is inside dense subgraph

Input: Sequence X, Bitmap B and vertex u
Output: List of in-neighbors of u
$in \leftarrow \varepsilon$
$occur \leftarrow rank_X(u, |X|)$
for $i \leftarrow 1$ **to** $occur$ **do**
  $y \leftarrow select_X(u, i)$
  $p \leftarrow select_B(0, y + 1)$
  $o \leftarrow p - y$ { $= rank_B(1, p)$ }
  $m \leftarrow o \bmod 3$
  if $m = 2$ **then**
    $s \leftarrow select_B(1, o - 1) - (o - 1) + 1$
    $e \leftarrow select_B(1, o + 1) - (o + 1)$
  else if $m = 0$ **then**
    $s \leftarrow select_B(1, o - 2) - (o - 2) + 1$
    $e \leftarrow select_B(1, o) - o$
  else
    $s \leftarrow 1$
    $e \leftarrow 0$
  end if
  for $j \leftarrow s$ **to** e **do**
    $d \leftarrow access_X(j)$
    if ($d \neq u$) **then**
      $in \leftarrow in : d$
    end if
  end for
end for
return in

Figure 4.13: Algorithm for getting in-neighbors of u

Table 4.5: % of edges belonging to bicliques and dense subgraphs (dense) with respect to the total number of edges. Also the sequence length ratio of representing dense subgraphs (X) and bicliques (S and C).

Dataset	%Edges bicliques	%Edges dense	$\|X\|/(\|S\|+\|C\|)$
eu-2005	91.30	91.86	0.90
indochina	93.29	94.51	0.91
uk-2002	90.80	91.41	0.92
arabic-2005	94.16	94.61	0.85
sk-2005	94.83	95.29	0.92
enron	46.28	48.47	0.95
dblp-2011	49.88	65.51	0.94
LiveJournal-SNAP	53.77	56.37	0.95
LiveJournal2008	54.17	56.51	0.94
enwiki-2013	62.31	64.43	0.95

$H(S, C)$, or the bicliques $(S - C, C)$ or $(S, C - S)$.

Another interesting query could be computing the density of the dense subgraphs stored in H. Let us use a definition given by Definition 4.2.3. The density of a clique is always 2. The density of a biclique (S, C) is $\frac{2 \cdot |S| \cdot |C|}{(|S|+|C|)(|S|+|C|-1)}$. Algorithm of Figure 4.15 computes the density of all dense subgraphs and reports all dense subgraphs with a density over a given γ.

Some of other possible mining queries are the following:

- Get the number of cliques where node u participates. We just count the number of times node u is in the M component of X. The algorithm is similar to, say, the Algorithm of Figure 4.12, yet it just identifies the component where u is and it increments a counter whenever this component is M.
- Get the number of bicliques where node u participates. This is basically the same as the previous query, yet this time we count when node u is in components L or R. If u is in L it is a *source* and if it is in R it is a *center*.
- Get the number of subgraphs. We just compute the number of 1s in B and divide this number by 3. This is because for every dense subgraph in X there are 3 1s in B, as shown in Figure 4.10.

4.4.2 Dense Subgraph Mining Effectiveness

We experiment with graphs presented in Table 2.1.

We used our dense subgraph discovery algorithm with parameters $ES = 500, 100, 50, 30, 15, 6$, discovering larger to smaller dense subgraphs. We used *threshold* $= 10$ for eu-2005, enron, dblp-2011, and enwiki-2013 *threshold* $= 100$ for indochina-2004, uk-2002, LiveJournal-2008 and LiveJournal-SNAP and *threshold* $= 500$ for arabic-2005.

Input: Sequence X, bitmap B and vertex u
Output: List of *allcliques* and *allbicliques*
$allcliques \leftarrow \langle\rangle$
$allbicliques \leftarrow \langle\rangle$
$n \leftarrow rank_B(1, |B|)$
$cur \leftarrow 1$, $p1 \leftarrow 0$
while $(cur < n)$ **do**
  $p2 \leftarrow select_B(1, cur + 1)$
  $p3 \leftarrow select_B(1, cur + 2)$
  $p4 \leftarrow select_B(1, cur + 3)$
  if $(p2 - p1 = 1 \wedge p4 - p3 = 1)$ **then**
    $s \leftarrow p2 - (cur + 1) + 1$
    $e \leftarrow p3 - (cur + 2)$
    $clique \leftarrow \emptyset$
    for $(i \leftarrow s$ **to** $e)$ **do**
      $clique \leftarrow clique \cup \{access_X(i)\}$
    end for
    $allcliques.add(clique)$
  else if $(p3 - p2 = 1)$ **then**
    $s \leftarrow p1 - cur + 1$
    $m \leftarrow p2 - (cur + 1)$
    $e \leftarrow p4 - (cur + 3)$
    $biclique.S \leftarrow \emptyset$, $biclique.C \leftarrow \emptyset$
    for $(i \leftarrow s$ **to** $m)$ **do**
      $biclique.S \leftarrow biclique.S \cup \{access_X(i)\}$
    end for
    for $(i \leftarrow m + 1$ **to** $e)$ **do**
      $biclique.C \leftarrow biclique.C \cup \{access_X(i)\}$
    end for
    $allbicliques.add(biclique)$
  else
    other type of dense subgraph
  end if
  $cur \leftarrow cur + 3$, $p1 \leftarrow p4$
end while
return $allcliques, allbicliques$

Figure 4.14: Algorithm for listing all cliques and bicliques

Input: Sequence X, bitmap B and density γ
Output: List ls of dense subgraphs with density at least γ
$ls \leftarrow \langle\rangle$
$n \leftarrow rank_B(1, |B|)$
$cur \leftarrow 1$, $p1 \leftarrow 0$
while $(cur < n)$ **do**
  $p2 \leftarrow select_B(1, cur + 1)$
  $p3 \leftarrow select_B(1, cur + 2)$
  $p4 \leftarrow select_B(1, cur + 3)$
  $V \leftarrow p4 - p1 - 3$
  $E \leftarrow (p3 - p1 - 2) \cdot (p4 - p2 - 2)$
  $g \leftarrow E/(V \cdot (V - 1)/2)$
  if $(g \geq \gamma)$ **then**
    $ls.add((cur + 2)/3)$
  end if
  $cur \leftarrow cur + 3$, $p1 \leftarrow p4$
end while
return ls

Figure 4.15: Algorithm for listing dense subgraphs with $density > \gamma$

Table 4.5 also gives some performance figures on our dense subgraph mining algorithm. We show the fraction of edges represented in bicliques versus the edges captured in dense subgraphs. We also show the ratio of the length of the sequence representing dense subgraphs (X), with respect to the length of the sequences used with bicliques ($|S| + |C|$). On Web graphs (where we give the input to the mining algorithm in natural order), 91%–95% of the edges are captured in dense subgraphs, which would have been only slightly less if we had captured only bicliques, as in Buehrer and Chellapilla [29]. Finding dense subgraphs, however, captures the structure of social networks much better than just finding bicliques, improving the percentage of edges captured from 46%–55% to 48%–65%. Note also that the fraction of edges in dense subgraphs is much lower on social networks, which anticipates the well-known fact that Web graphs are more compressible than social networks. The ratio in the length of the sequences in Web graphs show that using dense subgraphs allows capturing a similar number of edges using a shorter sequence than using bicliques. In the case of social networks the difference is not much, but dense subgraphs still capture more edges than using only bicliques.

Table 4.6 complements this information with the fraction of cliques, bicliques, and other dense subgraphs, with respect to the total number of dense subgraphs found, as well as their average size. This shows that pure cliques are not very significant, and that more than half of the times the algorithm is able to extend a biclique to a more general dense subgraph, thereby improving the space usage.

The following experiments consider the final size of our representation. For the component H we represent sequence X using WT or GMR, and for bitmap B we use RG or RRR. These

Table 4.6: Fraction and average size of cliques, bicliques, and the rest of dense graphs found.

Dataset	Cliques		Bicliques		Dense subgraphs	
	fraction	size	fraction	size	fraction	size
eu-2005	7.19%	7.44	46.67%	18.67	46.14%	20.73
indochina-2004	6.53%	5.18	34.55%	22.47	58.92%	20.54
uk-2002	3.56%	4.47	42.16%	17.84	54.28%	21.92
arabic-2005	3.76%	4.32	42.09%	23.05	54.15%	22.44
sk-2005	2.40%	4.21	58.07%	23.26	39.52%	24.92
enron	0.07%	3.33	67.20%	13.09	32.73%	20.75
dblp-2011	18.22%	3.95	27.76%	8.37	54.02%	6.91
LiveJournal-SNAP	2.41%	3.47	57.99%	9.64	39.60%	10.53
LiveJournal-2008	2.37%	3.44	59.77%	9.75	37.86%	10.47
enwiki-2013	0.62%	3.30	68.73%	15.12	30.65%	11.14

implementations are obtained from the library *libcds*[1]. In particular, we used version 10.0. For WT we used the variant "without pointers". For the component R we use either k2tree [23] or MPk [39], the improvement over the proposal of Maserrat and Pei [85]. Although we use the most recent version of the k2tree, we use it with natural node ordering to maintain consistency between the node names in H and R. An alternative would have been to use BFS ordering for both, that is, reordering before applying the dense subgraph mining, but this turned out to be less effective.

Table 4.7 shows how the compression evolves depending on parameter ES, on graph dblp-2011. ES values in Tables 4.7 and 4.8 represent the last value we consider in the ES list. For instance, $ES = 100$, in Table 4.7, means that we use the sequence of values $ES = 500, 100$. As ES decreases, we capture more dense subgraphs, yet they are of lower quality, thus their space saving decreases. To illustrate this we show the length $|X| = \sum_r |S_r| + |C_r| - |S_r \cap C_r|$, the number of bytes used to represent X and B ("$|H|$ in bytes", using WT for X and RRR for B), and the total edges represented by H (RE $= \sum_r |S_r| \cdot |C_r|$). All these indicators grow as ES decreases. Then we show the size of R in bytes (using representation MPk, with the best k for R), which decreases with ES. As explained, what also decreases is RE$/|X|$, which indicates the average number of edges represented by each node we write in X. Finally, we write the overall compression performance achieved in bpe, computed as $bpe = (bits(H) + bits(R))/|E|$. It turns out that there is an optimum ES value for each graph, which we use to maximize compression.

Tables 4.8 and 4.9 compare the compression we achieve with the alternatives we have chosen for Web and social graphs. We show the last ES value used for discovering dense subgraphs, the ratio RE$/|X|$, and the compression performance in bpe obtained on Web and social graphs. We use WT and RRR where the sampling parameter is 64 for compressing H. For compressing R, we use *k2tree* for Web graphs and MPk for social networks, which gave the best results (with enron and enwiki-2013, where using *k2tree* on R provides better compression than MPk, as displayed).

[1] Available at `https://github.com/fclaude/libcds`

Table 4.7: Evolution of compression as ES decreases, for the dblp-2011 dataset.

	ES				
	500	100	50	30	15
$\|X\|$	6.6K	75.8K	232.6K	456.8K	1.05M
$\|H\|$ in bytes	47.4K	168.0K	487.9K	950.9K	2.20M
RE	165.8K	636.0K	1.24M	1.92M	3.25M
$\|R\|$ in bytes	7.05M	6.88M	6.70M	6.50M	6.00M
RE/$\|X\|$	25.12	8.38	5.33	4.20	3.09
bpe	8.47	8.41	8.58	8.89	9.79

Table 4.8: Compression performance for Web graphs, compared to other techniques. DSM refers to DSM-ES10-T10+k2tree. Values in bold represent the best compression in bpe.

	$G = H \cup R$			k2treeBFS	DSM
Dataset	ES	RE/$\|X\|$	bpe	bpe	bpe
eu-2005	6	7.29	2.67	3.22	**2.11**
indochina-2004	6	14.17	1.49	1.23	**0.87**
uk-2002	6	8.50	2.52	2.04	**1.53**
arabic-2005	6	11.56	1.85	1.67	**1.08**
sk-2005	6	11.22	2.12	1.91	**1.55**

Table 4.9: Compression performance for social networks, compared to other techniques. BVLLP refers to BV adapted to support out/in-neighbor queries using the LLP node ordering of the graphs. k2tree refers to applying k2tree on the remainder graph without applying another node ordering algorithm. k2treeLLP refers to applying k2tree on the graphs using LLP node ordering. Values in bold represent the best compression in bpe.

	$G = H \cup R$			MPk	k2treeLLP	BVLLP
Dataset	ES	RE/$\|X\|$	bpe	bpe	bpe	bpe
enron (with k2tree)	6	2.06	**10.07**	17.02	10.31	18.30
enron	6	2.06	15.42	17.02	10.31	18.30
dblp-2011	100	8.38	**8.41**	8.48	9.83	10.13
LiveJournal-SNAP	500	12.66	**13.02**	13.25	17.35	17.22
LiveJournal-2008	100	4.88	**13.04**	13.35	13.63	17.84
enwiki-2013 (with k2tree)	6	1.88	**14.25**	18.73	14.65	23.30
enwiki-2013	6	1.88	15.98	18.73	14.65	24.74

We compare the results with standalone *k2treeBFS* on Web graphs, *k2treeLLP* on enron and enwiki-2013, and MPk on the other social networks.

Our technique does not obtain space gains on Web graphs compared to *k2treesBFS*. Moreover, the variant DSM-ES10-T10+k2treeBFS of Section 3.3.3, also included in the table, is even better.

On social networks, the gains of our new technique are more modest with respect to MPk. However, we show next that our structure is faster too. Moreover, there are no other competing techniques as on Web graphs. Our development of Section 3.3.3 does not work at all (it reduces less than 1.5% of edges, while increasing nodes when introducing virtual ones). The next best result is obtained with BV (which is more effective than GB and AD for social networks).

We note that BV is unable to retrieve in-neighbors. To carry out a fair comparison, we follow BV authors suggestion [16] for supporting out-in/neighbor queries. They suggest to compute the set E_{sym} of all symmetric edges, that is, those for which both (u, v) and (v, u) exist. Then they consider the graph $G_{sym} = (V, E_{sym})$ and $G_d(V, E - E_{sym})$, so that storing G_{sym}, G_d, and the transpose of G_d enables both types of queries. The space we report in Table 4.9 for BV considers this arrangement and, as anticipated, it is not competitive.

4.4.3 Space/time Performance

Figure 4.16 shows the space/time tradeoffs achieved on dblp-2011 and LiveJournal-SNAP graphs considering only the H component. We test different ES parameters. We use WT and GMR for the structures that represent X and RRR for B. These are indicated in the plots as WT-r and GMR-r. The sampling parameter for RRR is 16, 32, and 64, which yields a line for each combination. Along this section we measure out-neighbor query times, as in-neighbor queries perform almost identically. We observe that using WT provides more compression than GMR, but it requires more time.

The plots show how using increasing ES improves space and time simultaneously, until reaching the optimum space. Using a larger ES value also implies fewer iterations on the dense subgraph extraction algorithm, which dominates construction time.

We now consider our technique on social networks, representing H and R, the latter using either k2tree or MPk, and compare it considering space and time with the state of the art. This includes standalone k2trees with BFS and natural order, MPk with the best k and, as a control value, BV with out/in-neighbor support. Now our time is the sum of the time spent on H and on R. We represent H using our best alternatives based on DSM-ESx-WT-r and DSM-ESx-GMR-r.

Figure 4.17 compares the results on social networks. Figure 4.18 shows a closeup of the best alternatives for dblp-2011 and LiveJournal-Snap datasets. While on enron and enwiki-2013, k2tree with natural order is the best choice when using little space. On the other networks our combination of *DSM* and MPk is the best, slightly superseding standalone

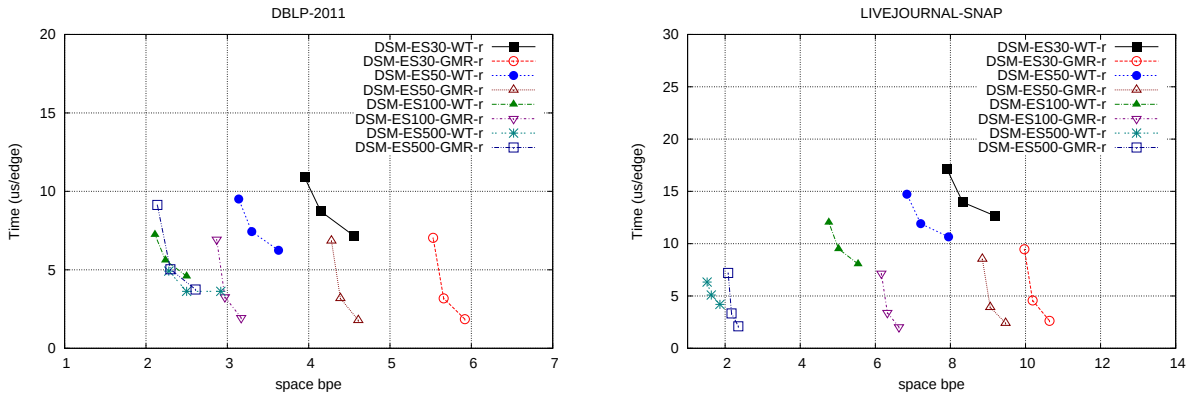


Figure 4.16: Space/time efficiency with out-neighbor queries on social networks, for various ES values (only component H is considered for dense subgraphs)

MPk in both space and time. In addition, it is important to note that BVLLP provides the best random access time (an order of magnitude better), but the compression efficiency is not as good as with the other techniques.

Figures 4.19 and 4.20 carry out a similar study on Web graphs. In Figure 4.19 we also show that, on these graphs *DSM* improves significantly in space with respect to detecting only bicliques ("BI"), while the time is similar. This comes from the fact that the length of the sequence X, which represents H, is shorter than the sequences used for representing bicliques only (see Table 4.5). Figure 4.20 shows that the structure proposed in this section is dominated in space and time by that proposed in Section 3.3. Yet, we remind that the structure we propose in this section is able to answer various mining queries related to the dense subgraphs found, easily and using no extra space.

4.5 Conclusions

This chapter presents two compression schemes for Web and social networks. Both schemes are based on extracting subgraphs and then representing them using compact data structures based on bitmaps and sequences. The first scheme extracts bicliques ($S \cap C = \emptyset$) and represents the graph by the collection of bicliques (H) and the rest of the graph (R). The collection of bicliques is represented by two symbol sequences X_s and X_c and two bitmaps B_s and B_c. The second scheme improves on the first by extracting dense subgraphs ($S \cap C \neq \emptyset$), and extending the previous representation using only a symbol sequence X and a bitmap B. These representations allow out/in-neighbor queries plus mining queries over the sequences. The first scheme includes queries based on bicliques and the second scheme provides a wider range of queries including listing cliques. We show that the second scheme provides better space/time efficiency than the first scheme for Web and social networks. We also show

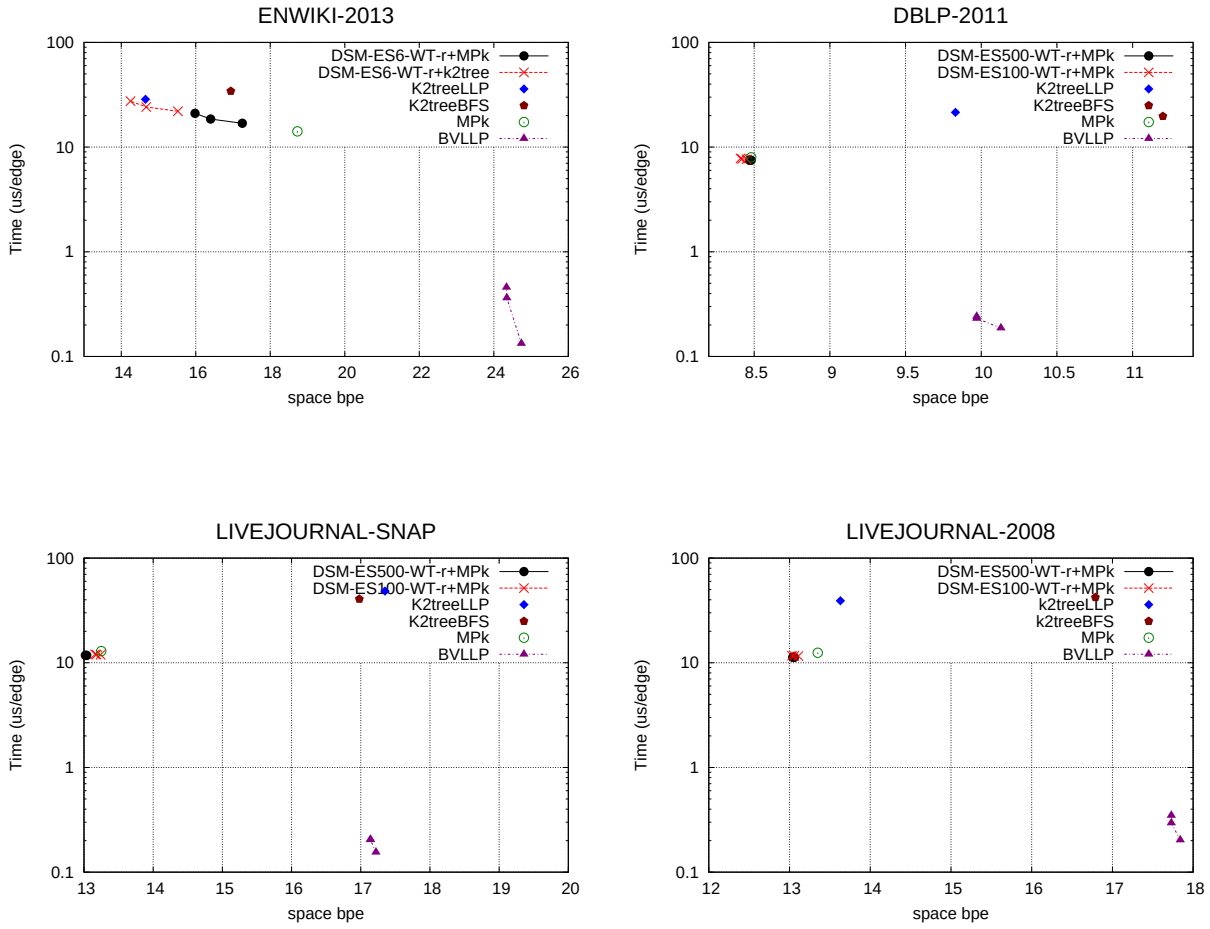


Figure 4.17: Space/time tradeoffs for social networks using dense subgraphs.

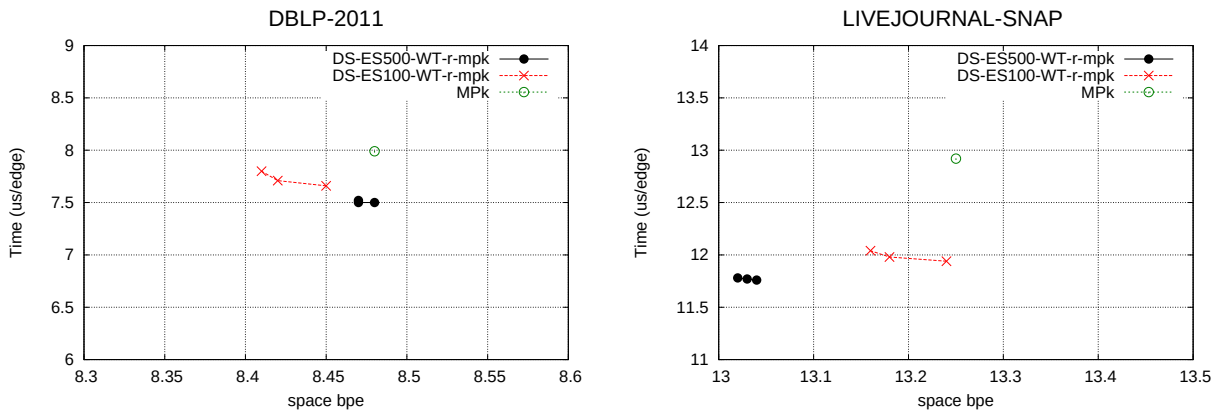


Figure 4.18: Space/time tradeoffs for social networks using dense subgraphs in more detail for dblp-2011 and LiveJournal-SNAP.

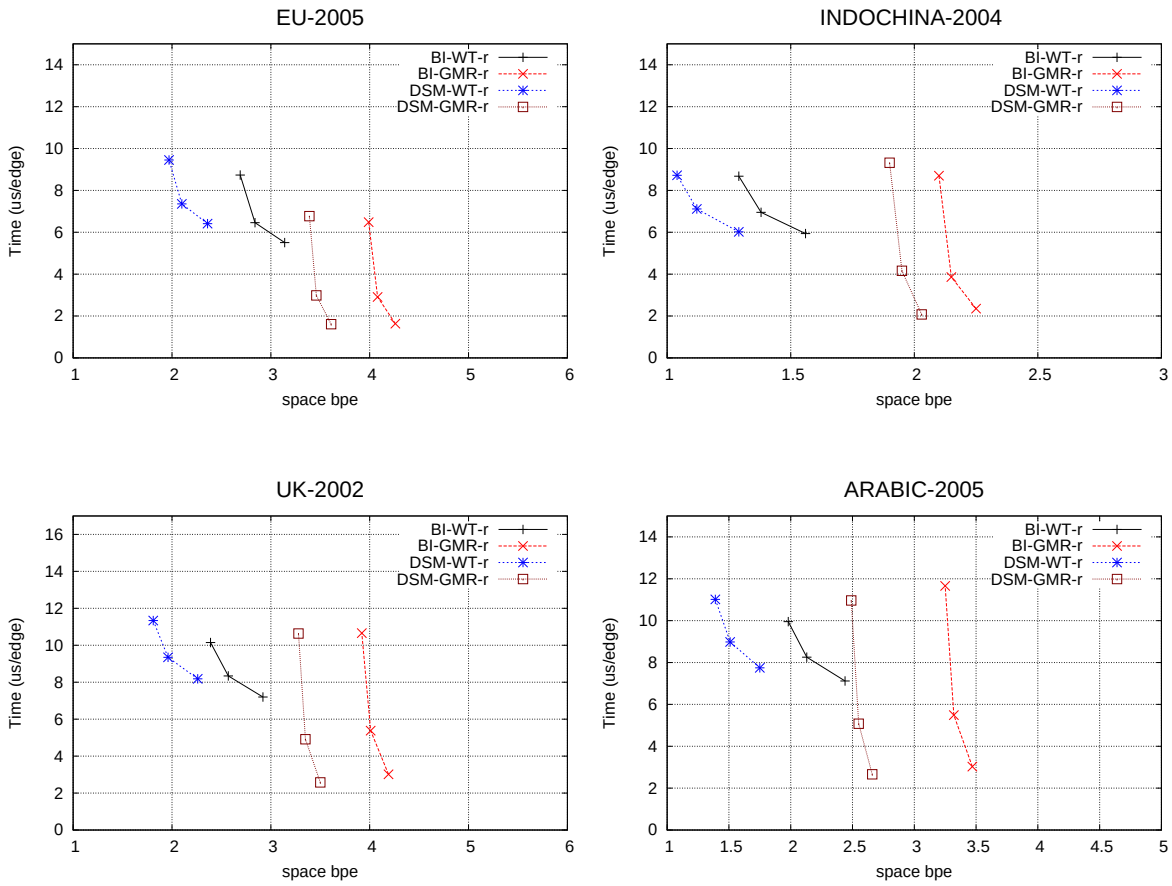


Figure 4.19: Space/time efficiency with out-neighbor queries on Web graphs, for various sequence representations (only component H is considered) using dense subgraphs.

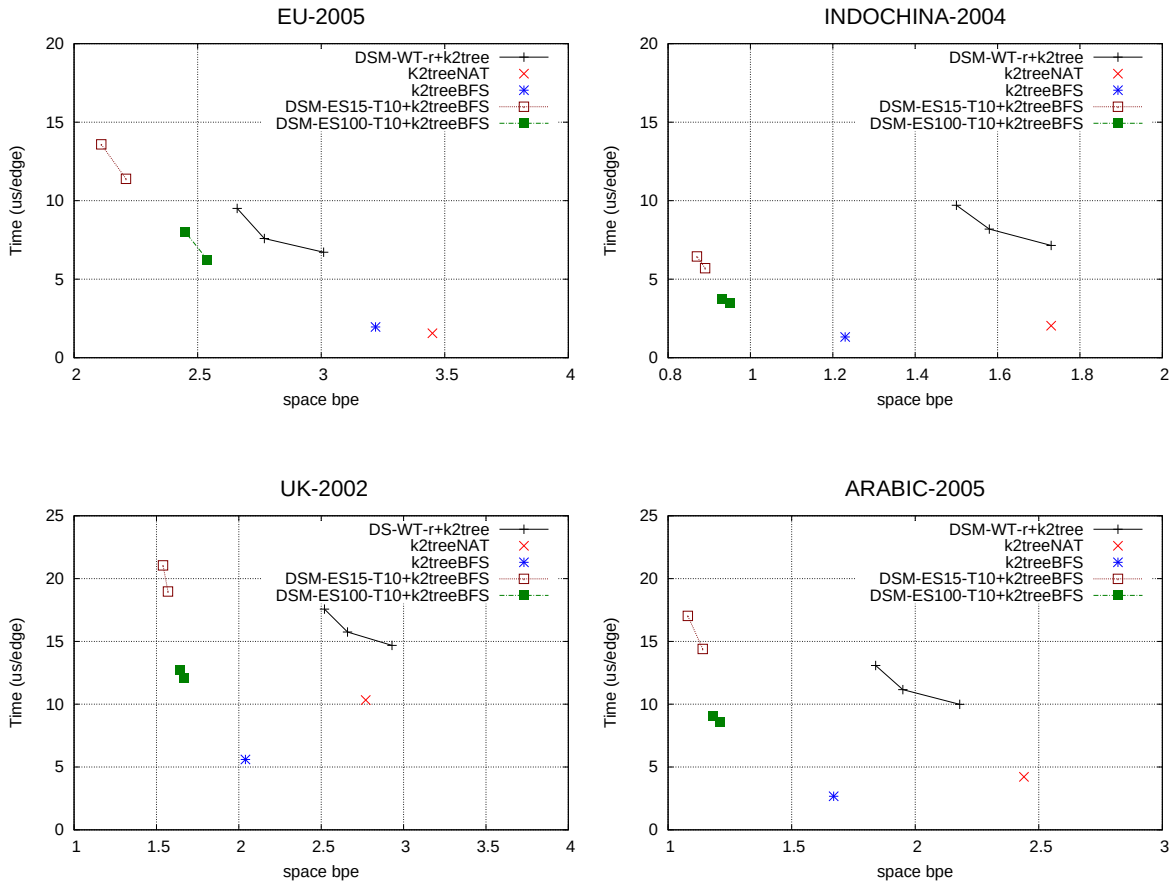


Figure 4.20: Space/time tradeoffs for Web graphs using dense subgraphs.

that using dense subgraphs with virtual nodes and k2tree with BFS node ordering on Web graphs provides better space/time tradeoff than the scheme based on sequences and bitmaps. However, the scheme based on sequences provide a much richer set of queries than the scheme based on virtual nodes. In the context of social networks, we show that the second scheme combined with MPk [39] or k2tree [22] improves on MPk and k2tree performance. However, the compression efficiency on social networks is still far from the efficiency achieved on Web graphs.

Chapter 5

Streaming and External Memory Algorithms

Discovering and extracting relevant patterns from graphs are difficult tasks. These tasks are becoming increasingly challenging as graphs are growing in size at a great rate. This scenario requires to process information more efficiently, including the need of processing data that cannot fit in main memory. Typical approaches for processing data using limited main memory include the streaming and external memory models. However, finding relevant graph patterns such as cliques and complete bipartite graphs are even more difficult under such models.

This chapter describes a semi-streaming and an external memory algorithm for extracting the dense subgraphs described in Chapter 2. However, in this Chapter we consider only bipartite cores, which means we do not add self-loops to adjacency lists in advanced. We describe a semi-streaming algorithm based on the Hierarchical Heavy Hitter problem, which enables to process the graph using limited memory in a few passes. The external memory algorithm is based on the External R-way Merge-sort, which is used for finding clusters and reordering the graph based on bipartite clusters.

5.1 Related Work

We first review the streaming model of computation and some solutions proposed in the context of massive data including large graphs, and then we do the same with the external memory model.

5.1.1 Streaming Models

The streaming model is an important model of computation for processing massive datasets [91, 93]. The main idea of the model is based on regarding the flow of data as a stream that is

read sequentially once or a few times, and each time the stream is read only a small amount of memory can be stored about it. The most restrictive streaming model allows $O($ polylog $n)$ space and a single or a few passes over the data. One important feature of a streaming algorithm is that the algorithm can defer action until a group of stream items arrive; there is no restriction to process each data item immediately. This model has been successfully applied in many problems such as computing statistics, norms, histograms, frequent items (Heavy Hitters), etc. See [93] for a complete review of algorithms and applications.

In the context of graphs, a graph stream is defined as a stream of edges $E = e_1, e_2, ..., e_m$ describing a graph $G(V, E)$ on $n = |V|$ vertices. Providing streaming solutions for graph problems has been more difficult, and only a few have been successfully proposed, such as counting triangles [12] using only one pass over the graph. The main difficulty comes from the constrains of limited memory with the input access patterns of graph algorithms.

To address graph problems in the streaming model, several more relaxed models have been proposed, such as the semi-streaming model which allows $O(n$ polylog $n)$ space, which means that we have space to store the vertices and information related to them but not all the edges, and one or a few sequential read-only passes through the graph [58, 93, 116]. The w-stream model is similar, but it allows temporary streams to read/write on disk [103, 116]; the stream-sort model, which also allows intermediate streams that can be sorted in requiring only one pass [6, 116]; and the stream-with-annotations model, which behaves as a semi-streaming algorithm, but assumes a helper that can be queried for a small number of annotations [31].

Feigenbaum et al. [58] propose semi-streaming algorithms for finding approximations to the unweighted maximum bipartite matching problem (matching with the largest number of edges) with an approximation ratio $2/3 - \varepsilon$ in $O(\frac{\log 1/\varepsilon}{\varepsilon})$ passes using $O(n \log n)$ memory, where $0 < \varepsilon < 1/3$. The proposed algorithm is based on first finding a bipartition, then using a matching of the graph, and then finding a set of simultaneous length-3 augmenting paths. The maximum bipartite matching algorithm increases the size of the matching by repeatedly finding a set of simultaneously length-3 augmenting paths. They also provide a semi-streaming algorithm for finding a weighted matching. They use edge weights of edges in the stream and compare them with the sum of the weights of the edges in the current matching M. If the incoming weight is greater than twice the sum of weights of M, the new edge is added to M. With this algorithm, they show that using $O(n \log n)$ storage they can construct a weighted matching that is at least $1/6$ of the optimal size.

The idea of the w-stream model is that, while performing a pass over the stream, it is possible to output a temporary stream. Then, it is possible to have a sequence of streams $S_1, S_2, ...S_p$ for p passes, where the input of the i-th pass is S_{i-1} and the output of the pass is S_i, where S_{i-1} and S_i are streams and the output of the computation is stream S_p. Demestrescu et al. [52] show that the single-source shortest path problem in directed graphs can be solved in the w-stream model in $O(n \log^{3/2} n)/\sqrt{s})$ passes. For undirected connectivity they propose an $O(n \log n)/s$ passes algorithm.

The stream-sort model is formalized by Aggarwal et al. [6]. They provide randomized algorithms for undirected s-t-connectivity. The problem is to detect if there is a path from vertex s to vertex t, given a graph stream of length n. They show that, using a randomized algorithm over the stream-sort model, it is possible to detect connectivity using $O(\log n)$

passes and $O(\log n)$ space. The algorithm consists of labeling the vertices of the graph with a random number (r_i) of $3 \log n$ bits. Each vertex is then labeled by the smallest number among its neighbors and itself. All vertices that receive the same number are merged together. This process is repeated a logarithmic number of times and in the end the graph has no edges and each vertex represents a connected component. Then, if vertices s and t are in the same component, then there is a path between them. The streaming-sort algorithm is used to determine the lowest value of random numbers among its neighbors and itself. On each pass, the algorithm writes temporary streams. First, it writes (v_i, r_i) followed by all edges with start point v_i, replacing v_i with r_i (on each edge) and writing a temporary stream E'. Then a pass over E' generates a stream composed by (v_i, r_i) followed by all edges with endpoints v_i. Using a linear pass over this last stream, it is possible to determine the lowest r_i for each v_i, since for each (v_i, r_i) there is a list of edges sorted by v_i and where the start point of each of those is an r_i obtained in E'. It is then possible to obtain the lowest r_i for each v_i. This new vertex mapping is used for the next iteration on the graph.

Aggarwal et al. [4] propose a model for dense pattern mining using summarization of graph streams. They define dense patterns based on node-affinity and edge-density of patterns in a general way. Their approach removes small and large adjacency lists a priori because the dense pattern mining definition does not consider them as relevant. On the other hand, running time is in the order of thousand processed edges per second. More recently, Sariyüce et al. [106] propose incremental streaming algorithms for k-core decomposition, where a k-core is defined as a maximal connected subgraph in which every vertex is connected to at least k nodes in the subgraph. The core decomposition of a graph is the problem of finding the set of maximum k-cores of all vertices in the graph. Thus, an algorithm to find k-cores of a graph removes all vertices with degree less than k with their corresponding adjacency edges. The authors propose streaming algorithms supporting insertion and removal of edges for dynamic networks. The algorithms require reordering unprocessed vertices in subgraphs.

Another interesting computational model is the sliding window [46]. The sliding window model is similar to the streaming model in that there is only limited memory available for processing and ideally, the algorithm should go through the input only once. The difference is that in the sliding window model the algorithm may only consider recent data. There are two types of algorithms in this approach. One is *sequence-based*, in which there is a window of size k moving over the k most recently arrived data. The other is *time-based*, where windows of duration t consist of elements whose arrival timestamp is within a time interval t of the current time. The algorithms under this model must hold two properties: The input stream is accessed in sequential order and the order of the data elements is not controlled by the algorithm.

5.1.2 External Memory Model

External memory algorithms define memory layouts that are suitable for graph algorithms, where the goal is to exploit locality in order to reduce I/O costs, and in particular random access to disk [115]. This model consists of a single processing unit with an available memory of size M and a number of parallel disks (D) capable of storing a much larger amount of data. The model also considers that data can be transfered from disk to memory (read

operation) and from memory to disk (write operation) through a block size of B items, which are consecutive on disk.

Two fundamental primitives of the model are *scanning* and *sorting*. Scanning is the operation of streaming N items from disk to main memory with I/O complexity

$$scan(N) = \Theta(\frac{N}{DB}),$$

while the sorting operation on N items has complexity

$$sort(N) = \Theta(\frac{N}{DB} \log_{M/B} \frac{N}{B}).$$

The results presented in this chapter consider an external sort algorithm, R-way mergesort, which has such I/O complexity [61].

Several algorithms have been proposed to solve graph problems using the external memory model [115], such as maximal cliques, approximate d-degeneracy ordering [65], traversal algorithms and graph connectivity [76], computing connected components, and maximal matchings in undirected graphs [2]. Goodich and Pszona [65] propose an algorithm for approximating the d-degeneracy ordering to a $(2+\varepsilon)$ d-degeneracy of G in $O(sort(dn))$ I/Os, without knowing the value of d in advance. They also provide an external algorithm for listing maximal cliques based on the Bron-Kerbosch [28] algorithm improved by Tomita [113] and Eppstein et al. [57]. They build an external version of Eppstein algorithm using the $(2+\varepsilon)$ d-degeneracy ordering that uses d-degeneracy ordering of G and can list all its maximal cliques in $O(3^{d/3}sort(dn))$ I/Os.

An interesting available library is STXXL (Standard Template Library for Extra Large Datasets) [49, 50]. STXXL is an implementation of C++ standard template library STL for processing large datasets. It supports parallel disks, overlapping disk I/O and computation, and pipelining. Among other applications, the library has been used for solving text processing and graph problems, such as algorithms for suffix array construction [48], BFS graph traversal [7], minimum spanning tress and connected components [51], etc.

5.2 Heuristics and Algorithms

This section describes streaming and external memory algorithms for discovering and extracting bipartite cores. The bipartite core pattern is described in Definition 5.2.1.

Definition 5.2.1 A *Bipartite core* $H(S, C)$ of $G = (V, E)$ is a graph $G'(S \cup C, S \times C)$, where $S, C \subseteq V$, and $S \cap C = \emptyset$.

Similar bipartite cores are defined by Kumar [80].

The discovery algorithm consists of 4 steps. The first step computes P fingerprints (hash values) for each adjacency matrix building a matrix of $|V|$ rows and $|P|$ columns. On the second step, similar rows group together, where each group identifies a cluster (represented by a set of adjacency lists). Finally the mining algorithm identifies and extracts bipartite cores from the adjacency lists belonging to the same cluster, where the out-neighbors of each adjacency list are sorted first by frequency. This is an iterative process where on each iteration a set of bipartite cores are extracted from the graph. Figures 2.1 shows the general algorithm and Figure 2.3 shows an example with the 4 steps.

The next section describes a streaming algorithm that is used for clustering purposes within the bipartite core discovery algorithm.

5.2.1 Heavy Hitter and Hierarchical Heavy Hitter Problems

The Heavy Hitter problem is one of the most studied in data streams and has been used as a subroutine in more advanced data stream computations. The idea is to find the most frequent items in a given sequence. The two main types of algorithms for solving this problem are based on *Counters* and based on *Sketches*. Counter-based algorithms track a subset of items and monitor counts associated with them, while sketch-based algorithms use linear projections of the input items to vectors and solve the frequency estimation problem using those vectors. Therefore, an important difference between these two types is that sketch algorithms do not store items from the input sequence explicitly, whereas counter-based algorithms do.

The Heavy Hitter (HH) problem formal definition [44] is as follows :

Definition 5.2.2 Given a (multi) set S of size N and a threshold ϕ, a Heavy Hitter (HH) is an element whose frequency in S is no smaller than $\lfloor \phi N \rfloor$. Let f_e denote the frequency of each element e in S. Then $HH = \{e | f_e \geq \lfloor \phi N \rfloor\}$.

The problem of finding HH in data streams has been studied extensively (a good survey is [45]) and are based on summary structures that estimate element frequencies. Cormode et al. [43] provide a comparison of different algorithms that solve the problem. They show that the *Space saving*, which is a counter-based algorithm, is the best algorithm in terms of accuracy and efficiency. The algorithm consists of keeping k (item, count) pairs stored in a dictionary, initialized with the first k distinct items and their exact counts. Every time an arriving *item* is found in the dictionary (T), its counter is incremented; otherwise the algorithm chooses the item with the smallest count and replaces the item incrementing its current count. This approach for replacing items in the set may seem counterintuitive, since the new item may start with a overestimated counter, but the result is that, if T is large enough, all HH will appear in the final dictionary. This algorithm provides a good frequency estimation with an error of n/k. The time cost is bounded by the dictionary operation of finding the item and of getting the item with the smallest count.

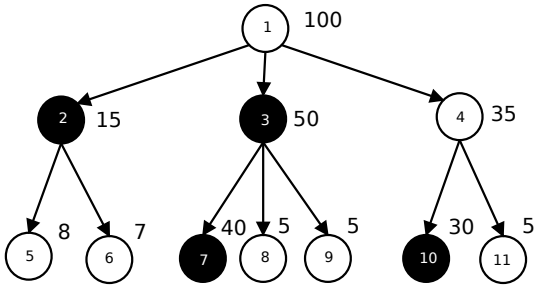


Figure 5.1: Example of Hierarchical Heavy Hitters.

Another problem related to finding HH in data streams is finding Hierarchical Heavy Hitters (HHH). In the HHH problem the idea is to find Heavy Hitters in a hierarchy. The problem consists of finding elements in the hierarchy that have a number of descendants no smaller than a given fraction ϕ after discounting descendants elements in the hierarchy that are already HHs. In other words, the idea is to find elements in the hierarchy such that their frequency (HHH count) exceeds ϕN, where the frequency is the sum of all descendants elements which have no HHH ancestors. Figure 5.1 shows an example of HHH, for a total of elements of 100; Heavy Hitters are shown as black nodes. In the example, $\phi = 0.1$ and $N = 100$, then $\phi N = 10$. Nodes 7 and 10 are Heavy Hitters because their frequencies are greater that ϕN. On the other hand, nodes 2 and 3 also conform Heavy Hitters because their children that are not Heavy Hitters themselves aggregate their frequencies as a group up to ϕN.

Formally the Hierarchical Heavy Hitter (HHH) problem is defined as follows [44]:

Definition 5.2.3 Given a (multi)set S of elements from a hierarchical domain D of height h, let $elements(T)$ be the union of elements that are descendants of a set of prefixes T of the domain hierarchy. Given a threshold ϕ, HHH is inductively defined. HHH_0, the hierarchical Heavy Hitters at level zero, are the Heavy Hitters of S. Given a prefix p at level i in the hierarchy, $F(p)$ is defined as $\sum f(e) : e \in elements(p) \wedge e \notin elements(\bigcup_{l=0}^{i-1} HHH_l)$. Then HHH_i is the set of hierarchical Heavy Hitters at level i, that is, the set $\{p|F(p) \geq \lfloor \phi N \rfloor\}$. The set HHH is $\bigcup_{i=0}^{h} HHH_i$.

The HHH problem cannot be solved exactly over data streams in general, therefore we follow the approximation given in previous works [44, 112]. We use the approximation algorithm given by Thaler et al. [112], which presents several advantages over previous algorithms. Such approximation improves time and space required to process each update and outputs the HHH with estimated frequencies. The approximation is based on the *Space saving* algorithm.

We show that the HHH problem can be used in the clustering algorithm. Every node in the hierarchy is a hash value of the clustering scheme, and heavy hitters represent the pair of hash values $(h1, h2)$, (using $P = 2$ in the clustering [70]) with highest count values. In

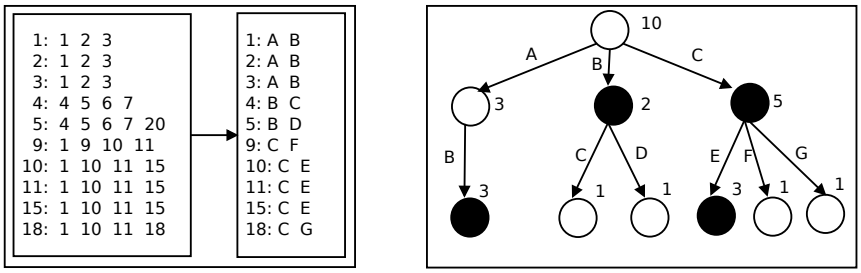


Figure 5.2: Example of the clustering algorithm seen as a Hierarchical Heavy Hitter problem, using $\phi = 0.2$, with $N = 10$.

order to find clusters, besides counts, the algorithm collects vertex ids associated to hashes that are Heavy Hitters. The algorithm also uses a global bitmap to insert vertex ids only once, and limits the number of vertex ids collected with a threshold. This approach avoids sorting the $P \times V$ hash matrix. Instead, it finds clusters using the Heavy Hitters found with the HHH approximation given by Thaler et al. [112]. Figure 5.2 shows an example of the clustering algorithm seen as a HHH problem. Black circles represent Heavy Hitters in the hierarchy. Bipartite cores are extracted using the algorithm that uses HHH in the clustering stage, as depicted in Figure 5.3.

5.2.2 Extracting Bipartite Cores using R-Way External Merge-sort

The R-way external merge-sort works in two phases [61]. The first is a "run formation" phase, where N input data are streamed in main memory using memory pieces of size M. Each piece of size M is sorted, having at the end of the phase N/M sorted runs. The second phase is the "merge phase", where groups of R runs are merged together. Runs in the merge phase are sorted using buffers of size B. The merge phase might take more than one pass; in each pass one buffer of size B from each run is maintained in main memory and one buffer is used for streaming out sorted runs. Sorting R runs is done using a Heap data structure. Since the memory usage of the sorting algorithm is bound to M and the buffer size is B, then $R = \lceil \frac{M}{B} - 1 \rceil$ buffers for input and one for output. The overall I/O performance of the algorithm is $O(N/B \log_{M/B} N/M)$, which is a $sort(N)$ primitive in the external memory model.

The scheme presented uses R-way external merge-sort in two different ways: One for sorting the matrix of $P \times V$ hashes, for the clustering phase and the other for reordering the input graph by cluster id. During the clustering phase, the algorithm computes hashes and sorts them by columns using external merge-sort; cluster ids are defined based on the conditions of pairs of hashes (given that $P = 2$). The vertices ids of each cluster are used for

```
1: Streaming Algorithm
2: bc ← ∅
3: bcccoll ← ∅
4: stream.init(thr, counters)
5: for i ← 1 to Iters do
6:    for (v, adj) ∈ G do
7:       fingers ← computeFingerprints(v, adj, P)
8:       stream.update(v, fingers)
9:    end for
10:   hhhArray ← stream.output()
11:   for hhh ∈ hhhArray  do
12:      cluster ← hhh.vertices
13:      bc ← mine(cluster)
14:      bccoll.add(dss)
15:   end for
16: end for
```

Figure 5.3: Streaming algorithm

```
1: External memory Algorithm
2: bc ← ∅
3: bcccoll ← ∅
4: msFinger.init(M, B, K)
5: for i ← 1 to Iters do
6:    for (v, adj) ∈ G do
7:       fingers ← computeFingerprints(v, adj, P)
8:       msFinger.add(v, fingers)
9:    end for
10:   msFinger.extmerge(fsortFingers)
11:   clusters ← msFinger.getClusters(fsortFingers)
12:   msPermute ← permute(M, B, K, clusters, fGraph)
13:   for cluster ∈ fGraph  do
14:      bc ← mine(cluster)
15:      bccoll.add(dss)
16:   end for
17: end for
```

Figure 5.4: External memory algorithm based on R-way external merge-sort

sorting the input graph based on cluster ids. In summary, this external memory algorithm requires two external *sorts*, one over the matrix $P \times V$ and one for permuting the graph based on vertex id. Therefore, the algorithm is $O(sort(2V) + sort(V + E))$, that is, $O(sort(V + E))$ I/O complexity. This complexity does not consider the mining part of the algorithm.

For the mining phase, the reordered graph is scanned by cluster. For this part of the extraction algorithm we try two different approaches. The first approach uses whatever main memory requires each cluster, which requires at most $O(V_c + E_c)$ memory, where V_c is the number of vertices in the cluster and E_c the number of edges. This scheme is labeled *extmem* in all figures. Such external algorithm is shown in Figure 5.4.

The second approach consists of limiting the memory M for processing the reordered graph, which applies the mining algorithm over the sequential elements read in such memory. This approach resembles the sliding window model, however, the implemented algorithm does not consider neither cluster overlap nor approximations and then the results can be improved with a smarter algorithm.

5.3 Experimental Evaluation

This section describes the results obtained by applying the algorithms presented in Section 5.2. All algorithms are implemented in C/C++ using gcc/g++ version 4.4.5 on a Linux PC with 16 processors Intel Xeon at 2.4 GHz, with 64 GB of RAM and 12 MB of cache. The source code provided by Thaler at *http://people.seas.harvard.edu/∼tsteinke/hhh/* is adapted for the scheme presented. All the experiments use the Web and social graphs in Table 2.1 (described in Chapter 2), and a minimum size of bipartite core of 6 ($|S| \times |C| = 6$) to avoid including cores that are too small.

5.3.1 Performance of Merge-sort

In this section we evaluate the efficiency of using external R-way merge-sort applied to our bipartite core listing. We use the algorithm for sorting the $P \times V$ hash matrix (see the algorithm described in Figure 2.1) and for permuting the input graph based on the clusters identified after sorting the hash matrix. We show the main performance metrics for different datasets, using $B = 8$KB, the best R, and varying the amount of memory M. In all tables, L is the height in the merge-sort tree, counting the first run (where the input graph is partitioned into pieces of size M). Table 5.1 shows the time and memory usage when computing hashes (CS) and sorting the hash matrix (SH). Sorting the hash matrix is only one part of the process of clustering; we also need to consider the memory resources for vertex ids that conform the clusters. Table 5.2 shows the time and memory usage when computing and sorting the hash matrix plus computing the clusters (CS+SH+CL). The comparison between Tables 5.1 and 5.2 shows that we need more memory for actually computing all the clusters in the graph. Table 5.3 shows the time and maximum memory usage when computing and sorting the hash matrix, plus clustering, plus permuting the input graph

Dataset	M(KB)	R	L	W	CS(s)	SH(s)	Mem. usage (MB)
eu-2005	16	7	4	3,456	26.01	9.81	12.2
	32	15	3	2,304	-	6.43	12.3
	128	63	2	1,152	-	3.13	13.1
indochina-2004	16	7	5	35,808	251.10	96.56	12.4
	32	15	4	26,832	-	73.91	12.4
	64	31	3	17,858	-	50.74	12.9
	256	127	2	8,832	-	24.65	14.4
uk-2002	16	7	6	115,520	420.21	311.91	12.6
	32	15	4	69,312	-	193.36	12,7
	65	31	3	46,208	-	125.65	12.8
	512	255	2	23,040	-	62.84	16.2
arabic-2005	16	7	6	142,520	837.20	385.23	12.7
	32	15	4	82,032	-	222.52	12.8
	65	31	3	56,960	-	154.58	13.1
	512	255	2	28,416	-	77.17	16.5

Table 5.1: R-way merge-sort performance for sorting the hash matrix.

based on the clusters found (CS+SH+CL+P). Such step is necessary for actually using the cluster information for the mining process. Finally, Table 5.4 shows all previous times plus mining (CS+SH+CL+P+Mine). As observed in Table 5.4, in terms of resource usage, the best alternative is to set M so that we can obtain $L = 2$, which is the fastest alternative for reordering the graph based on clusters. This is because the mining process requires a large amount of memory when clusters are large. We could limit that amount of memory, and in fact that is what we do in the sliding window algorithm.

5.3.2 Clustering

The next experiment measures the effect on memory and time requirements of the algorithms (described in Figures 5.3 and 5.4) with respect to the original algorithm (described in Figure 2.1 in Chapter 2, which is the algorithm with no memory restriction) just for clustering (using only the first iteration). Figure 5.5 shows the memory and time ratios compared to the original algorithm for all datasets. The results show that using HHH is not very effective in terms of speed, since running times are reduced only in 10-15%. On the other hand, memory ratio is more effective using external merge-sort than using HHH, but it is slower.

5.3.3 Clustering and Mining

This section describes the impact of using HHH and external merge-sort for extracting actual bipartite cores. We evaluate HHH using different numbers of counters (labeled cX, where X is the number of counters), the external memory algorithm using R-way merge-sort (labeled extmem), and the original algorithm (labeled mem), which has no memory restrictions. Running times are measured up to 10 iterations, since no more were necessary. All experiments

Dataset	M(KB)	R	L	CS+SH+CL(s)	Mem. usage (MB)
eu-2005	16	7	4	35.82	19.8
	32	15	3	32.52	19.8
	128	63	2	29.55	19.8
indochina-2004	16	7	5	345.21	63.2
	32	15	4	319.86	63.9
	64	31	3	303.74	63.2
	512	255	2	269.64	64.3
uk-2002	16	7	6	728.27	152.4
	32	15	4	612.34	152.4
	65	31	3	554.96	152.4
	512	255	2	499.71	152.3
arabic-2005	16	7	6	1181.81	167.0
	32	15	4	1038.43	167.5
	65	31	3	985.22	167.4
	512	255	2	938.14	168.4

Table 5.2: R-way merge-sort performance for computing, sorting hash matrix and clustering.

Dataset	M(KB)	R	L	CS+SH+CL+P(s)	Mem. usage (MB)
eu-2005	16	7	5	84.20	51.9
	32	15	4	71.15	51.9
	64	31	3	60.11	52.4
	256	78	2	46.91	54.9
indochina-2004	16	7	6	902.24	312.8
	32	15	5	784.39	312.8
	64	31	4	658.39	313.9
	128	63	3	573.57	315.8
	1024	511	2	467.18	323.9
uk-2002	16	7	7	2,189.12	791.3
	32	15	5	1,925.60	791.6
	65	31	4	1,321.12	791.3
	128	63	3	1,091.78	791.3
	1024	1032	2	810.30	798.9
arabic-2005	16	7	7	3,691.30	966.9
	32	15	5	2,816.20	965.5
	65	31	4	2,566.02	965.8
	256	127	3	2,120.17	967.6
	2,048	160	2	1,550.80	1,006.8

Table 5.3: R-way merge-sort performance for computing, sorting hash matrix, clustering and permuting the graph.

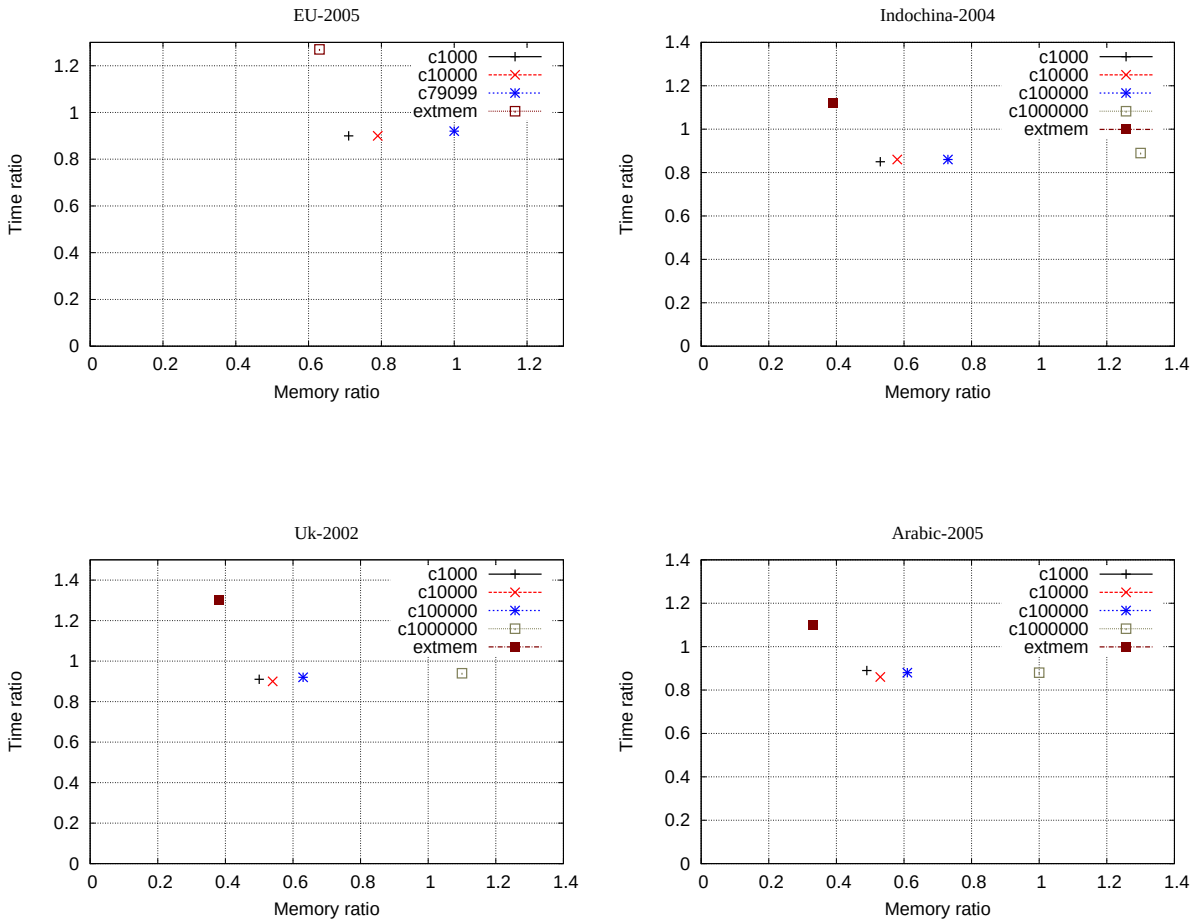


Figure 5.5: Memory and time ratios in a clustering phase using streaming and external merge-sort for fingerprint (hashes) sorting by columns.

Dataset	M(KB)	R	L	CS+SH+CL+P+Mine(s)	Mem. usage (MB)
eu-2005	16	7	5	148.11	99.0
	32	15	4	112.90	107.8
	64	31	3	114.42	110.6
	256	78	2	99.35	113.4
indochina-2004	16	7	6	2,222.71	1,413.5
	32	15	5	2,176.98	1,426.0
	64	31	4	1,901.48	1,412.7
	128	63	3	1,879.09	1,408.4
	1024	511	2	1,484.23	1,114.7
uk-2002	16	7	7	2,772.20	1,531.1
	32	15	5	2,169.35	1,533.2
	65	31	4	1,728.34	1,529.4
	128	63	3	1,569.02	1,528.9
	1024	1032	2	1,400.40	1,529.9
arabic-2005	16	7	7	6,122.45	2,849.7
	32	15	5	6,085.91	2,850.8
	65	31	4	4,300.93	2,850.4
	256	127	3	4,827.42	2,839.7
	2,048	160	2	3,694.76	2,843.9

Table 5.4: R-way merge-sort performance for computing, sorting hash matrix, clustering, permuting and mining the graph.

involving the external model are set to use an amount of memory that enables the use of a value for the parameter R so that the R-way merge-sort does the merge in 1 pass ($L = 2$). The performance is presented in terms of running time, number of edges in all bipartite cores, and bipartite core density rate (represented by RE/X in Section 4.4.2). The total number of edges is given by the edges represented in all bipartite cores on each iteration. The density rate is given by $edges/nodes = \sum \frac{|S|\times|C|}{|S|+|C|}$.

Figure 5.6 shows running times, number of recovered edges, and *edges/node* ratio for different Web graphs and social networks. HHH is able to capture more dense bipartite cores, as seen in *edges/nodes* figures on Web graphs, but not on social networks. Since HHH is an approximation for finding heavy hitters in a hierarchy, it is not able to recover all bipartite cores. On the other hand, using external merge-sort allows for actually obtaining the same precision in terms of total edges and ratio of *edges/nodes* in found bipartite cores, using less memory and higher running times.

Second, Figure 5.7 shows the performance ratios in terms of running time, memory usage and edge ratio with respect to the original algorithm for one and 5 iterations.
More specifically,

time_ratio is the ratio between the execution time of the memory reduced algorithms and the execution time of the original algorithm (the algorithm with no memory restrictions).

edge_ratio is the ratio between the number of edges participating in the extracted bipar-

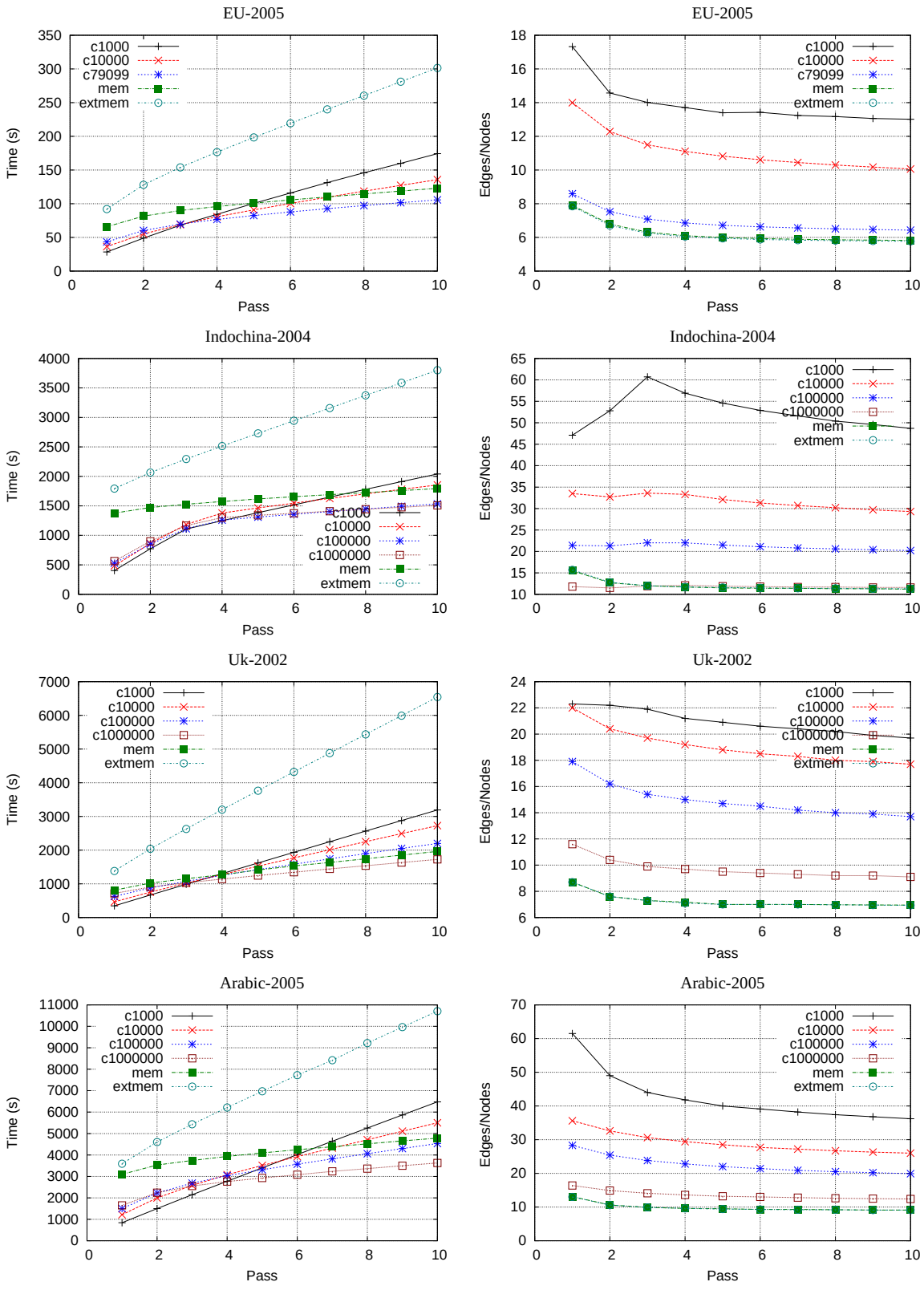


Figure 5.6: Running time, number of edges, and bipartite core sizes (*edges/nodes*) extracting bipartite cores using streaming and external merge-sort over Web and social graphs.

tite cores using the memory reduced algorithms and the number of edges achieved using the original algorithm, and

memory_ratio is the ratio between the amount of memory used by the memory reduced algorithms and the original algorithms.

The final experiment considers applying only the mining algorithm over graphs using URL ordering used by Boldi and Vigna to compress Web graphs [19] versus the ordering by clusters based on bipartite cores, that is, graphs are obtained using external merge-sort. The aim is to see how the mining algorithm behaves when streaming edges given by a previous order using only a *sliding window* of edges used in the mining algorithm [46]. The *sliding window* model is similar to the streaming model in that there is only limited memory available for processing and ideally the algorithm should go through the input only once. The difference is that in the *sliding window* model the algorithm can consider only recent data. Figure 5.8(line charts) shows the results when using a *sliding window* of 1,000, 2,000, and 5,000 edges on Web graphs (eu-2005, and arabic-2005) and a *sliding window* of 50, 100, and 1,000 for social networks (dblp, and LiveJournal). It is seen that when using HHH it is possible to find larger bipartite cores than mining a number of edges given in the sliding window. Figure 5.8 (bar charts) also shows time, edge, and memory ratios achieved at iteration 5 for this experiment compared with the original algorithm and HHH. We observe that the *sliding window* algorithm requires less memory than the other alternatives.

5.4 Conclusions

This chapter presents different schemes to extract bicliques from graphs reducing memory consumption. The streaming algorithm presented is based on an approximation of the Hierarchical Heavy Hitter (HHH) problem, which is used to find large clusters. This work also provides a solution that uses R-way external merge-sort to find clusters and reorder the input graph based on such clusters. After the clustering phase, a mining algorithm is applied to extract bipartite cores. Finally, we present a scheme that simply applies the mining algorithm, using a sliding window of edges, over an input graph stream using URL node ordering and an input graph stream sorted by clusters. The HHH solution is able to extract larger bicliques, but requires more memory than the other schemes. The external memory solution is able to extract bicliques at the same precision of the original algorithm while using less memory, yet using more time. The solution applying mining over a sorted graph is the best in terms of memory and time usage, but provides less precision than the external model. A possible alternative to study in the future is to see whether an improved mining algorithm using a sliding window of adjacency lists can take advantage of some stream orders.

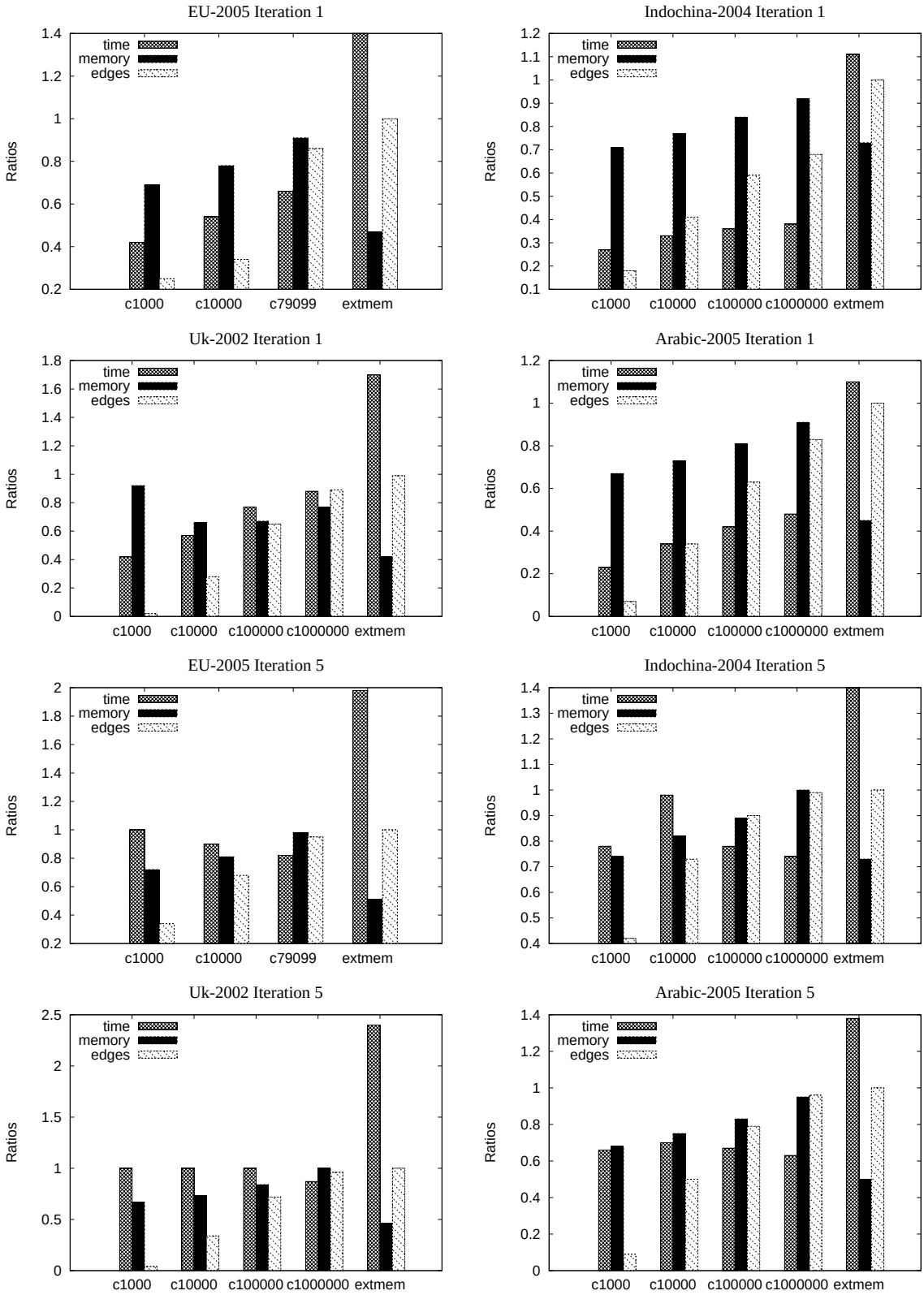


Figure 5.7: Memory, time, and edges in bipartite cores ratios extracting bipartite cores using streaming and external merge-sort for iteration 1 and 5.

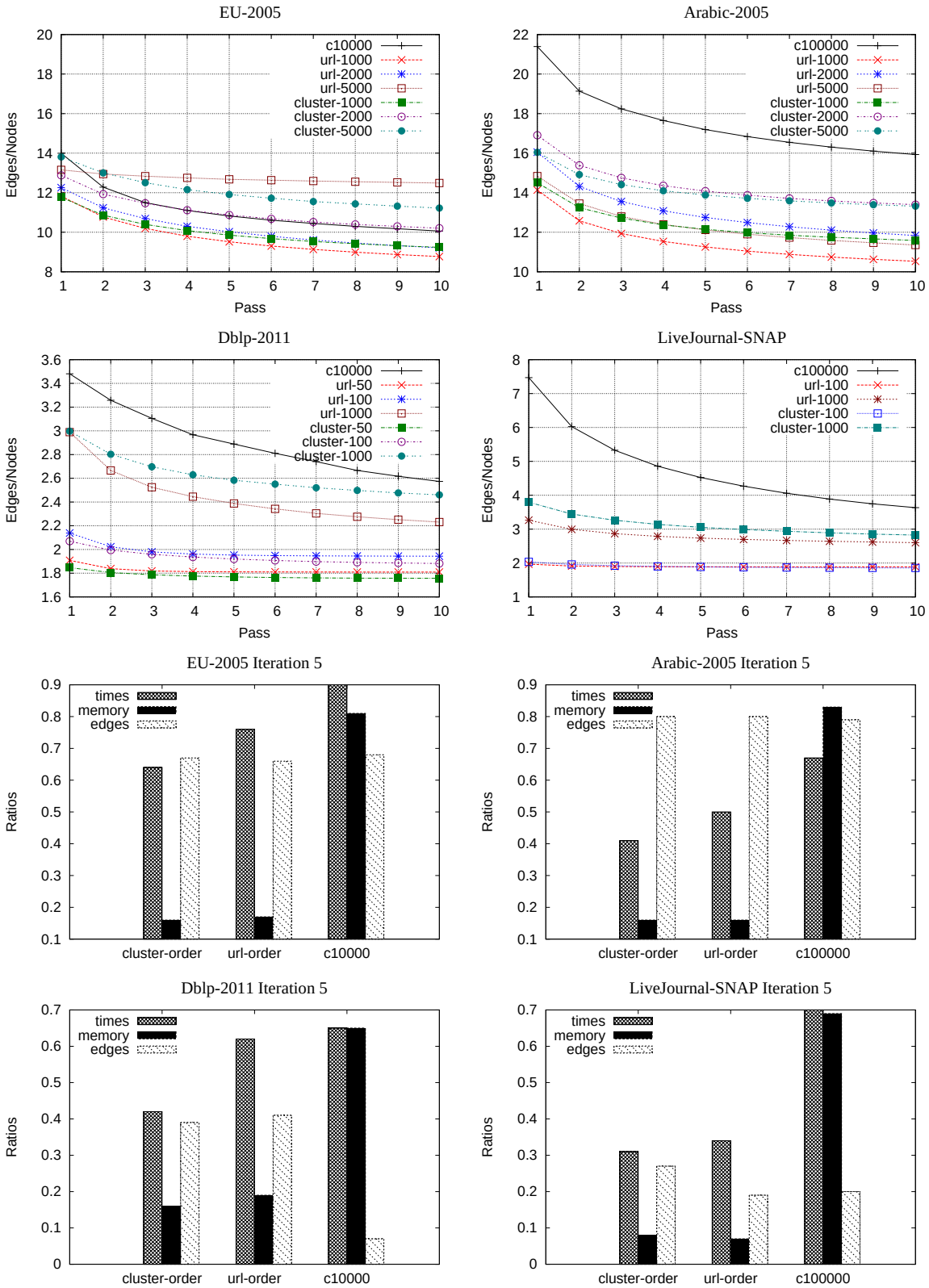


Figure 5.8: Bipartite core sizes (edges/nodes) applying the mining algorithm over input graphs ordered by URL and ordered by clusters (line charts). Memory, time, and edges in bipartite cores ratios when extracting bipartite cores by mining edges read from url and cluster ordered input graphs (bar charts).

Chapter 6

Discovering Dense Subgraphs in Parallel

In the last decade, various solutions have been proposed to address some of the problems associated with large graphs. One possible approach is to use distributed systems where distributed memory is aggregated to process the graph. In particular, many cluster-based systems have appeared in recent years for different application areas. Pregel [84] is a graph system that works on BSP model (Bulk Synchronous Parallel Model); Pegasus [74] is a graph mining library over Hadoop, which is the free implementation of MapReduce [47]; Gbase [73] provides a parallel indexing mechanism for graph mining operations based on Hadoop/MapReduce. There are also proposals that include counting triangles, which is useful for computing clustering coefficient [111]. Pace [94] discusses important differences between BSP and MapReduce and shows that algorithms that require to process local data in multiple rounds are more efficient using BSP than MapReduce. We use this finding for implementing our parallel algorithms using BSP, which in addition provides a theoretical base for analyzing computation, communication and synchronization costs in a parallel system.

In this chapter, our goal is to address the problems of dealing with large Web and social graphs using distributed/parallel systems and compressed structures. In particular, we propose scalable parallel algorithms for finding common graph patterns, which we use for compressed representations that provide access to out/in-neighbors and to enable mining queries without the need of decompression. The main contributions described in this chapter are:

- We present a scalable parallel algorithm for reducing the number of edges of the original Web graphs by finding dense subgraphs and adding virtual nodes. As described in chapter 4 we have shown that using this edge-reduced graph with BFS (Breath first traversal) ordering and k2tree [81] we achieve the best space compression on Web graphs. The parallel algorithm we propose exploits locality and provides good speedup, load balance, and scalability using BSP over a cluster-based system.
- We present a scalable iterative parallel algorithm for extracting dense subgraphs that exploits locality of adjacency lists and uses dynamic load balanced for maximizing processor utilization avoiding idle times. With this approach, we are able to improve speedup and resource utilization for discovering dense subgraphs on Web and social networks. We have shown on a previous work [69] that representing these dense sub-

graphs with compact data structures provides the best space/time tradeoffs for social networks. This representation also enables mining queries such as retrieving cliques, complete bipartite subgraphs (bicliques) and other related graph patterns.

6.1 Related Work

In this section, we review related work aiming to manage large graphs. In particular, we focus on distributed and parallel systems aiming to access large graphs in their original or compressed form, using different techniques.

Parallel and distributed data management has received a lot of attention in recent years due to the success of MapReduce, a distributed framework [47], and Hadoop, its open source counterpart [1]. MapReduce is simple to use, provides high throughput and it is suitable for simple data. Pregel [84], on the other hand, aims at processing graphs and it uses vertex-based computation under the BSP model. However, MapReduce and Pregel require hundreds or thousands of machines in order to process large graphs [109]. Some works, built on top of MapReduce/Hadoop, target specific graph problems such as counting triangles (used for computing clustering coefficient) [111], computing max-cover [35], and computing densest subgraphs [11]. Representative works that build complex systems using MapReduce include Pegasus [74], Gbase [73], and Trinity [108]. Pegasus, Pregel and Gbase focus on large graph querying/mining. Pegasus is built on top of Hadoop and Pregel is built using BSP model, over other Google clusters using underlying infrastructure such as GFS, BigTable, etc. Trinity also provides vertex-based computation, but the underlying infrastructure is based on a specially-designed distributed memory storage to provide low latency data access and high throughput message passing supporting synchronous and asynchronous communication models.

MapReduce computations on graphs depend heavily on interprocessor bandwidth, as graph structures are sent over the network iteration after iteration. On the other hand, Pregel improves upon MapReduce by passing computation results instead of graph structures among processors. The popularity of MapReduce as a parallel framework has created the need of studying its relationship to other major parallel computation models such as BSP and PRAM [94, 75]. Pace [94] discusses the differences between MapReduce and BSP and shows that algorithms that require to process data locally in multiple steps cannot be efficiently implemented in MapReduce if the number of supersteps is not constant.

6.2 Using Hadoop

This section describes a distributed approach using the MapReduce framework over the Hadoop implementation. The MapReduce paradigm enables the execution of distributed algorithms defined mostly as three functions. (1) a local function that takes a single data item and output a message. (2) an aggregation function to combine pairs of messages and (3) sometimes a third function for post processing. The local function is applied to input data

items independently and in parallel, and the aggregation function can be applied to pairs of messages in any order. Local functions (Mappers) are executed on local data items and basically map data items to $< key, value >$ pairs combining them to a set of intermediate $< key, value >$ pairs. Aggregation functions (Reducers) reduce a set of intermediate values which share a key to a smaller set of values.

Parallel processing occurs when multiple mappers execute over different local data items and reducers execute over different intermediate data generated by mappers. The sequential part is necessary because reducers must wait for mappers before executing. To simplify fault tolerance, in the MapReduce paradigm, Mappers materialize their output to disk before the Reducers can consume it, therefore the MapReduce paradigm has been mostly used for computing batch jobs.

This section describes the Map and Reduce functions that discover dense subgraphs using the distributed framework. Here, we extract dense subgraphs based on the algorithm described in Figure 2.3. The presented solution is iterative and consists of chains of functions with the form *map1() - reduce() - map2()*. These functions compute the steps described in Section 2.3. Specifically $map1()$ computes the clustering done in $Step1$, the $reduce()$ function computes the algorithm defined in $Step2$, and function $map2()$ computes $Step3$ and $Step4$. The $reduce()$ function groups all adjacency lists that share hashes and maps all adjacency lists associated with each cluster. The $map2()$ function just allows executing the mining algorithm over clusters in parallel. The algorithms for such functions are described in Figure 6.2, Figure 6.3, and Figure 6.4. Figure 6.2 uses the $computeFingerprints()$ function, which basically computes P fingerprints (hash values) for each adjacency list. Therefore, Figure 6.2 *emits* a set of pairs $< key, value >$ for each adjacency list, where the *key* is the set of *P hash values* and the *value* is the adjacency list itself. The *reduce* function groups together equal *keys* with corresponding adjacency lists coming from different parallel mapper executions forming clusters. The output of the reducer is taken by the $map2$ function so that for each cluster, it builds a prefix tree after sorting adjacency lists by edge frequency (and discards edges of frequency of 1). Each prefix tree can provide one or more dense subgraphs, which are extracted from the graph. An example of how this algorithm works is given in Figure 6.1.

6.2.1 Results

We evaluated the algorithms using the eu-2005 dataset (from Table 2.1) on a cluster of 6 nodes, where each node is a CPU Intel(R) Xeon(R) of 2.40 GHz and 4MB cache. We use up to 15 iterations for extracting dense subgraphs and measure the running time and number of edges belonging to dense subgraphs with respect to the number of edges of the input graph (% Edges). We did not use more than 15 iterations since the gain became insignificant after that. Figure 6.5 shows our results with 1, 5, 10 and 15 iterations. Comparing the running time and recovered edges (% Edges) achieved with the hadoop version against the in-memory implementation (eu-mem in Figure 6.5), the parallel results show that our distributed algorithm is not effective. There are two main reasons for the poor performance of our algorithm. First, our in-memory algorithm loads the graph in main memory and then does the computations in an iterative way, keeping the state from one iteration to the other in

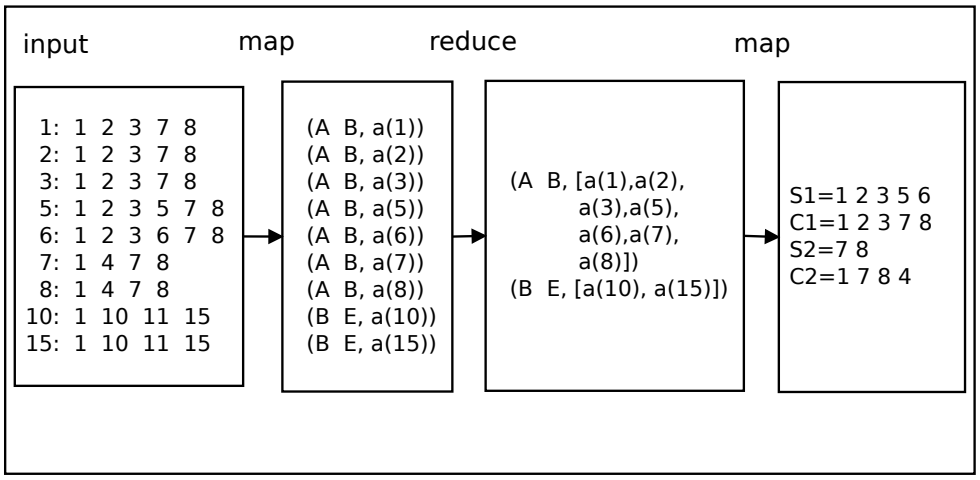


Figure 6.1: Example of the dense subgraph discovery using MapReduce using one iteration.

Input: $< a(u), \emptyset >$
Output: $< k, a(u) >$
Let $a(u)$ be an adjacency list for vertex u.
key $k \leftarrow computeFingerprints(a(u))$
$emit(k, a(u))$

Figure 6.2: $map1()$ function: Compute Hashes (fingerprints).

memory. Our distributed algorithm is also iterative, however the hadoop approach does not allow keeping data locally in memory from one iteration to the next. Therefore we need to write into HDFS (Hadoop Distributed File System) from one iteration to the next, which increases the running time of our implementation. Second, our hadoop algorithm is only able to capture clusters with pair of hashes, that is, with the same $(h1, h2)$, and then it does not detect pairs of hashes with only the first hash in common (as it is shown in Figure 5.2). Then, this prevents our distributed algorithm finding the same amount of dense subgraphs when decresing the number of edges. It is clear that MapReduce is a powerful framework for processing massive data. In our context, it is a clear that our algorithm should be improved to avoid keeping state locally between iterations in order to improve performance and solution quality.

Input: $< k, a(u) \subseteq A >$
Output: $< k, B >$
Array B // form a cluster
for ($a(u) \in A$) **do**
  $B.add(a(u))$
end for
$emit(k, B)$

Figure 6.3: $reduce()$ and $map2()$ functions: Build Clusters based on Hashes.

Input: $< k, clusters \subseteq C >$
Output: Set of dense subgraphs ($denseSubs$)
$denseSubs \leftarrow \emptyset$
for (cluster $c \in C$) **do**
  Adjacencylists $r \leftarrow SortEdgesByFrequency(c)$
  PrefixTree $p \leftarrow BuildPrefixTree(r)$
  $denseSubs \leftarrow p.mine()$
end for

Figure 6.4: Cluster Mining.

6.3 BSP Approach

Our first parallel algorithm (Figure 6.6) consists of representing Web graphs reducing edges by a factor between 5 and 10, adding only a small percentage of virtual nodes (between 10 and 15 %). We apply other compression techniques over this edge-reduced graph for supporting out/in-neighbor queries. We show that, for Web graphs, we are able to achieve competitive tradeoff using AD [9] for out-neighbor queries and best space at some time cost using k2tree [22] with BFS ordering over the edge-reduced graph [70]. However, this approach does not work well on social networks. Our second parallel algorithm (Figure 6.7) extracts dense subgraphs, which are the basis for our second compressed structure representation (described in Section 4.4). The compression scheme represents the collection of dense subgraphs in compressed form using compact data structures. In this scheme, we apply other compression techniques over the remainder of the graphs. We show in Section 3.3.3 that we achieve the best space/time tradeoff using this approach on social networks supporting out/in-neighbor queries.

6.3.1 BSP Model and its Cost Analysis

The BSP model provides an efficient parallel distributed memory model where it is possible to consider several relevant parameters of a real parallel computer system. A BSP computer is defined by P processors, each with its local memory, connected via a point-to-point communication link. BSP algorithms proceed in supersteps, in each of which processors receive input data, perform asynchronous computation over its data, and communicate output at the end. Supersteps are synchronized at the end using barriers. An algorithm designed in BSP is

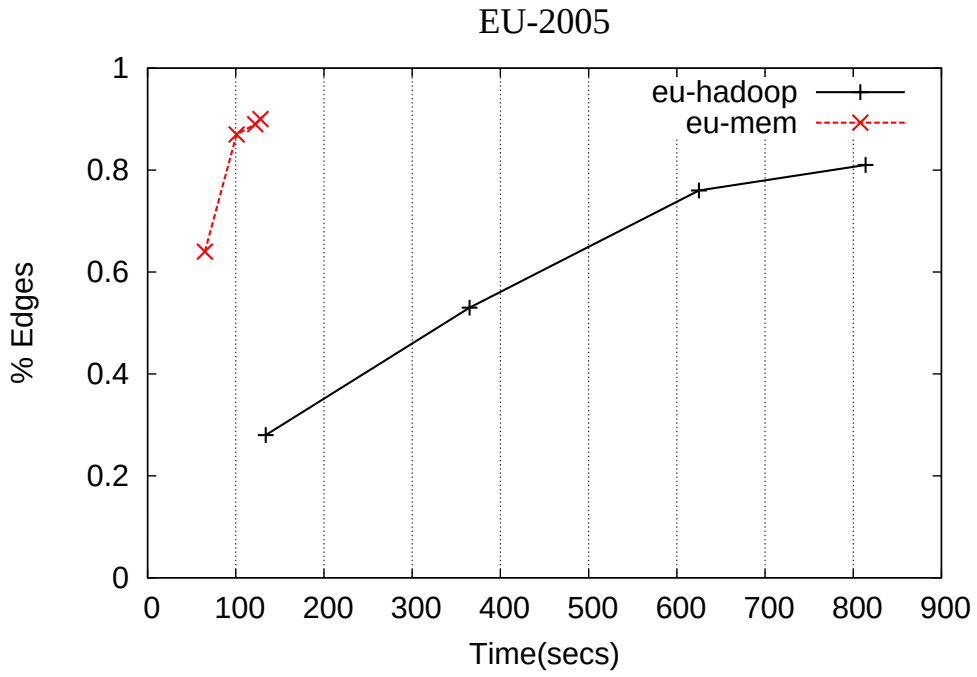


Figure 6.5: Running time and number of edges belonging to dense subgraphs for different iterations using Hadoop.

measured by three main features: *computation*, *communication*, and *synchronization* costs. The cost model is given by $W + H \times g + L$, where W is the maximum cost of computation on a processor, H is the maximum input/output communicated among processors, g is the *latency* cost and L is the synchronization cost.

Description and Cost Analysis of Edge-reducing Algorithm

Figure 6.6 describes our parallel DSM for reducing edges and adding virtual nodes for compressing Web graphs. During **Step 0** each processor processes G_p in parallel locally. Each iteration finds all clusters on G_p and on each cluster, the mining algorithm discovers dense subgraphs of the type H with components (S,C) of size at least ES. For each subgraph discovered, we create local virtual node ids ($localVnodes$) and use them to separate sets (S,C) and replace it on G_p. Since we want to have only one global edge-reduced graph, we need to have virtual nodes globally defined. We cannot define virtual nodes locally because node ids are distributed in all partitions and using BFS ordering locally would make the same node have different ids on different processors. Adding local virtual node id mapping information for each partition would destroy compression. Therefore, we use virtual node ids globally, and then apply BFS. In **Step 0** all processors send a tuple with $(lVnodeInit, numberLVnodes)$ (which are the first local virtual node id and the number of local virtual nodes) to processor 0. In **Step 1** processor 0 relabels local virtual nodes to global ids and sends that information to all processors. Relabeling is done by changing the $gVnodeInit$ based on the number of virtual nodes found in previous processed processor tuple, that is $gVnodeInit_i = vninit$ and $gVnodeInit_{i+1} = \sum numberLVnodes_i$. In **Step 2** all processors receive tuples with global

Input: G_p,ES, T
Output: Reduced $RG(|V + VN|, E2)$ graph
Each processor reads its data partition
{**Step 0**}
for $(i \leftarrow 0$ **to** $T - 1$ $)$ **do**
  $clusters = FindClusters()$
  for $(c \in clusters)$ **do**
    $Sets(S, C) = FindDenseSubs(c, ES)$
    $localVnodes = DefineSets(S, C)$
    $Replace(G_p, Sets(S, C), localVnodes)$
    $AddVnodes(G_p, Sets(S, C), localVnodes)$
  end for
end for
$sendLocalVnodeMsg()$
$sync()$
{**Step 1**}
if $(proc == 0)$ **then**
  $lvnodes = RecibeMsgs()$
  $gvnodes = ProcVNodeGlobal(lvnodes)$
  $sendGlobalVnodes(gvnodes)$
end if
$sync()$
{**Step 2**}
$gvnodes = RecieveMsgs()$
$replaceVnodes(G_p, lvnodes, gvnodes)$
return RG

Figure 6.6: Parallel edge-reduction algorithm with virtual nodes

virtual node ids and each processor replaces local virtual node ids for global ones. Therefore, the total cost is $O(T(\frac{|E|}{P}\log\frac{|E|}{P}) + Pg + L + |V + VN|)$. The cost analysis is as follows:

- **Step 0**. The cost of computing clustering on G_p and discovering dense subgraphs on each cluster is $O(T\frac{|E|}{P}\log\frac{|E|}{P})$, where $|E|$ is the number of edges in G, P the number of available processors, and T the number of iterations. The cost of sending one tuple per processor to processor 0 is $O(Pg + L)$.
- **Step 1**. The cost of processing global virtual node ids from local ids is $O(P)$ and sending local virtual node ids to all processors is $O(Pg + L)$.
- **Step 2**. Finally each processor replaces local virtual node ids for global virtual node ids in G_p. Considering all processors the cost is $O(|V + VN|)$. This complexity indicates that the algorithm does not scale well with respect V, however, since $V \ll E$ the scalability is good with respect to the size of the graph $(V + E)$.

DSM for subgraph extraction and dynamic load balance

Figure 6.7 describes our parallel algorithm for extracting dense subgraphs using dynamic load balance. This is an iterative algorithm, where each iteration has several steps. In **Step 0** each processor computes in parallel clustering and mining, and extracts dense subgraphs and periodically sends its workload information to processor 0. Processor workload tuple is given by ES and $numDSs$, where ES is the current size of the dense subgraphs that are mined and $numDSs$ is the number of subgraphs at the current iteration. Function $period()$ determines how often processors send their load. In **Step 1** processor 0 receives local load from all processors, computes a global load tuple containing ($minpid$,$maxpid$,$minES$,$maxES$,$minDSs$,$maxDSs$), and decides whether load balance is to be performed and the amount of data to move. If it decides to apply load balance, it sends a global load balance tuple to all processors. In **Step 2** each processor receives the global balance tuple and heavier processors sends portion of their data to the lighter processors. **Step 0** is computed T times and each processor sends workload tuples to processor 0 T_p times. During **Step 1** processor 0 computes workload tuples and decides whether heavier processors will send a portion of its G_p data (M) to lighter processors. The decision to apply load balance depends on the distance between $(maxES, minES)$ and $(minDSs, maxDSs)$ among processors is over a given threshold, which can happen T_d times. The total cost is $O(T(\frac{|E|}{P}\log\frac{|E|}{P}) + T_p(Pg + L + P) + T_d((P + M)g + L))$. The cost analysis for one iteration is as follows:

- **Step 0**. The cost of computing clustering on G_p and discovering dense subgraphs is $O(\frac{|E|}{P}\log\frac{|E|}{P})$, where $|E|$ is the number of edges in G, and P the number of available processors. In this step all processors send local workload tuples to processor 0 in time $O(Pg + L)$ periodically.
- **Step 1**. Later, processor 0 computes a global load tuple in $O(P)$ time and sends it to all processors in time $O(Pg + L)$ if it decides to perform load balance.
- **Step 2**. Heavier processors send M data to lighter processors and the cost is $O(Mg+L)$, where M is a portion of G_p data.

Input: G_p, $esArray$, T, $threshold$.
Output: Dense subgraph collection
Each processor reads its data partition
$ES = esArray.first()$
{**Step 0**}
for $(i \leftarrow 0$ **to** $T)$ **do**
  $clusters = FindClusters()$
  for $(c \in clusters)$ **do**
    $Sets(S, C) = FindDenseSubs(c, ES)$
    $numDSs = |Sets(S, C)|$
    $WriteToDisk(Sets(S, C))$
  end for
  if $(i == period())$ **then**
    $sendLoadMsg()$
    $sync()$
    {**Step 1**}
    if $(proc == 0)$ **then**
      $ProcessLoad()$
      $sendDistInfo()$ (to all procs)
    end if
    $sync()$
  end if
  {**Step 2**}
  $sendData()$
  if $(numDSs < threshold)$ **then**
    $ES = esArray.next$
  end if
end for

Figure 6.7: Parallel dense subgraph extraction with dynamic load balancing.

6.4 Experimental Evaluation

We perform different experiments over Web and social graphs described in Table 2.1 (described in chapter 2). We use the natural order for input graphs in all our experiments. We implemented parallel algorithms using C++ and BSP over a cluster with at most 64 processors. Each processor is an Intel 2.66 GHz, with 24 GB of RAM and 8 MB of cache.

We study the performance of our parallel DSM with virtual nodes and parallel DSM for extracting dense subgraphs using dynamic load balance. We analyze the effect of using different number of processors in terms of compression efficiency, running times, compression speed (*original bytes/total running time*), and speedup. We also study the performance in terms of the percentage of edges represented in extracted dense subgraphs.

6.4.1 Performance of Parallel DSM for Edge Reduction

We first evaluate dividing the input graph into parts. We found that processing partitions of equal number of edges gives us a more balanced processor work load than using an equal number of vertices. Therefore, we divide original graphs by balancing the number of edges contained by complete lists of out-neighbors.

Figure 6.8 shows parallel running times and compression performance (bpe) using different number of processors for different Web graphs. We include running times for computing DSM-ESx-T10 (where $ES = x$ for finding dense subgraphs of at least size x, and $T = 10$ i.e. 10 iterations); and the running time for achieving the complete compression structure, which consists of two parts: DSM-ESx-T10 builds a graph with fewer edges and virtual nodes (RG), and k2treeBFS applies BFS and k2tree over RG. As observed, the running time improves greatly without affecting compression. These results suggest that there is a great amount of locality of reference in adjacency lists. Therefore, finding dense subgraphs in parallel locally does not affect compression. Figure 6.8 shows that the cost of applying k2treeBFS, which is sequential, has more impact on larger graphs. This is seen on the offset visible on Arabic (top figure) and on the compression speed curve for the same dataset.

Figure 6.9-(left) shows the speedup achieved using different number of processors with DSM with virtual nodes (K2treeBFS not included). We observe that the speedup is higher for larger Web graphs, which suggests that larger graphs have more memory access penalties using only one processor. In other words, larger graphs take more advantage of memory aggregation in the cluster system.

6.4.2 Performance of Parallel DSM for Dense Subgraph Extraction

We study the compression efficiency for different processors when extracting dense subgraphs. We measure the effect of using dynamic load balancing, speedup, edge extraction and compression efficiency for Web and social graphs. Figure 6.9-(right) shows that using dynamic load balance recovers more edges. It is more effective in social networks than in Web graphs,

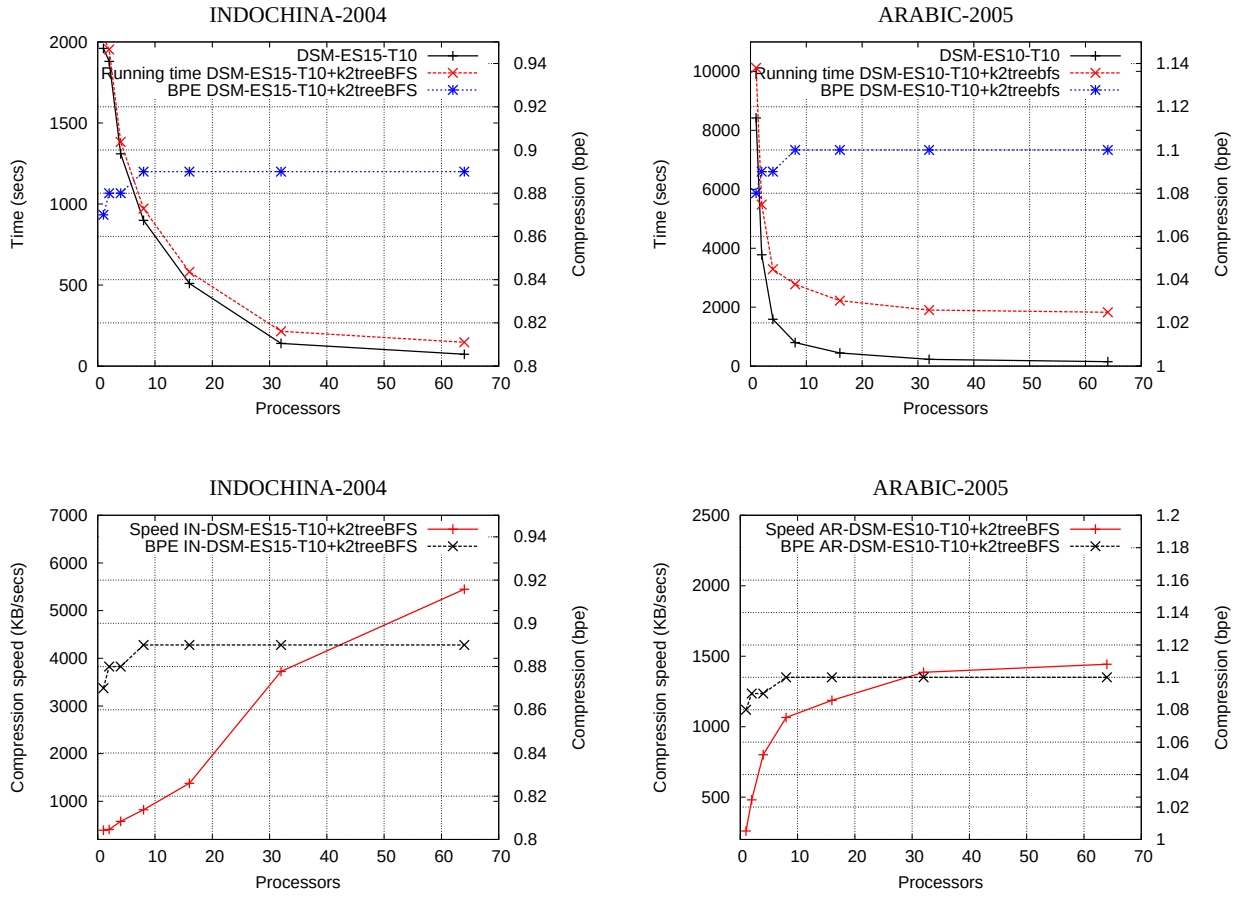


Figure 6.8: Parallel running time with corresponding compression (top) and compression speed in KB/secs (bottom).

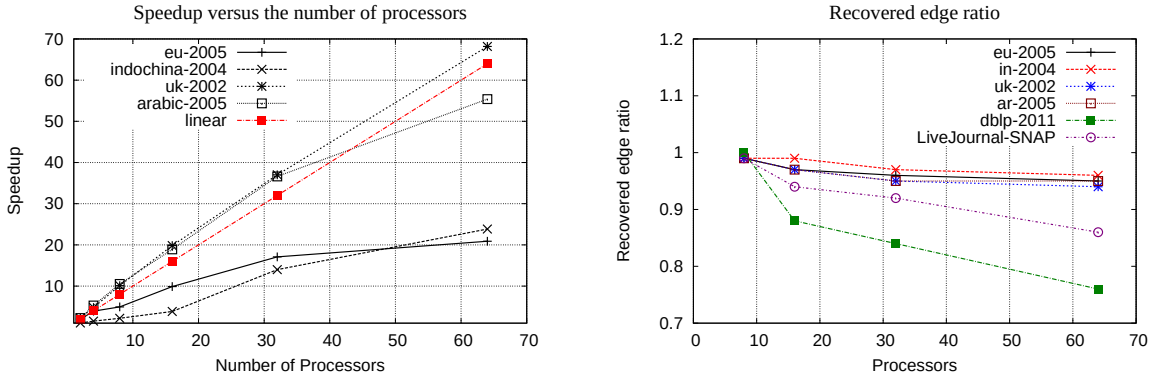


Figure 6.9: Speedup using different number of processors (left). Recovered edge ratio for Web and social graphs (right).

having greater gain with more processors. We compared both schemes by computing % edges recovered without load balance divided by the % of edges recovered with load balance using the same number of iterations. When using dynamic load balance, the processor sends work load tuples every 10 iterations and processor 0 decides to move data from heavier to lighter processors when the difference between $(minES, maxES, minDSs, maxDSs)$ is over a given *threshold*.

Figure 6.10 shows the speedup and Figure 6.11 the edge ratio achieved by using 8 and 64 processors on the parallel extraction compared with the sequential execution for Web and social graphs. We use 100 iterations for extracting dense subgraphs on Web graphs and dblp-2011 and 200 iterations on LiveJournal. We also show the differences on the number of edges extracted using the parallel and sequential algorithms. We measure such difference using the *Edge ratio*, which we define by the number of edges extracted in parallel divided by the edges recovered in the sequential execution. We observe that we are able to extract more than 90% of edges and at the same time achieve good speedups in Web graphs. However, extracting edges in parallel for social networks is less effective, as shown by the *Edge ratio* value. Figure 6.12 shows the running time for DSM with dense subgraph extraction, considering only time for extraction, complete compression time (including MPk), and the compression achieved for social networks using H and R with MPk [39]. This figure also shows that the sequential part of the compression construction slows down the compression time.

6.5 Conclusions

This chapter presents distributed and parallel parallel algorithms for DSM, a sequential algorithm for compressing Web and social graphs via discovering dense subgraphs [70]. We consider a distributed approach using the MapReduce paradigm implemented by Hadoop and then consider a parallel approach using BSP (using the BSPonMPI library).

We first design and evaluate a distributed algorithm based on the MapReduce paradigm. However, we found that such algorithm is not efficient, mainly because our algorithm is iterative and Hadoop does not allow to keep data locally between iterations, and then data must be written/read to/from the Hadoop Distributed File System (HDFS) at each iteration. It is clear that MapReduce is a powerful paradigm, but our algorithm does not exploit well its features. A possible approach for studying in the future is to design a MapReduce algorithm from scratch instead of starting from a distributed version of our sequential algorithm.

Then, we designed parallel algorithms using the BSP model. This model allows one to keep data locally in memory between iterations. Our first parallel algorithm based on BSP uses DSM with virtual nodes for reducing the number of edges. This algorithm exploits locality of reference of adjacency lists in Web graphs. Applying BFS ordering and k2tree over parallel edge-reduced Web graphs does not degrade compression efficiency.

Our second algorithm extracts dense subgraphs in parallel using dynamic load balance. Both algorithms provide good speedup and compression efficiency. However, since both

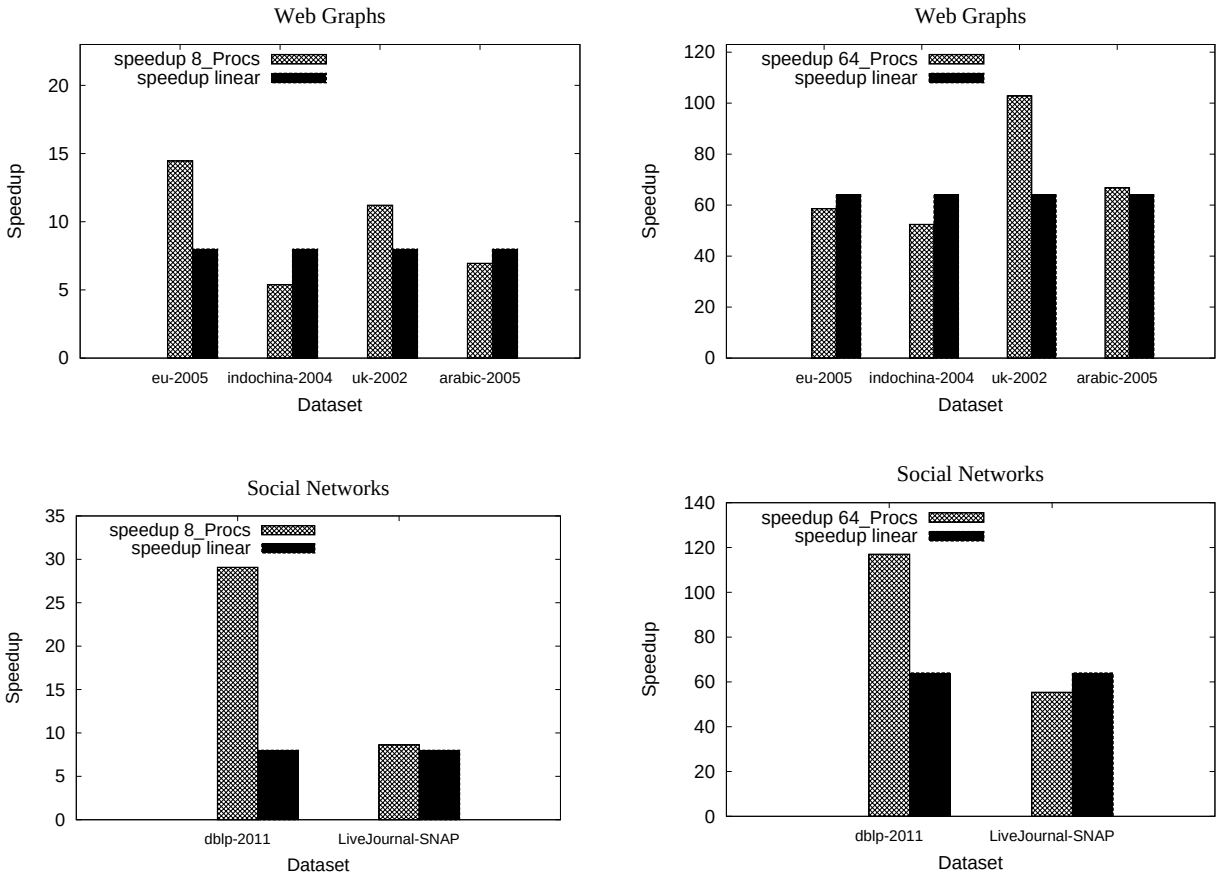


Figure 6.10: Speedup captured by dense subgraphs for Web graphs and social networks, using 100 and 200 iterations, for parallel extraction versus the sequential algorithm.

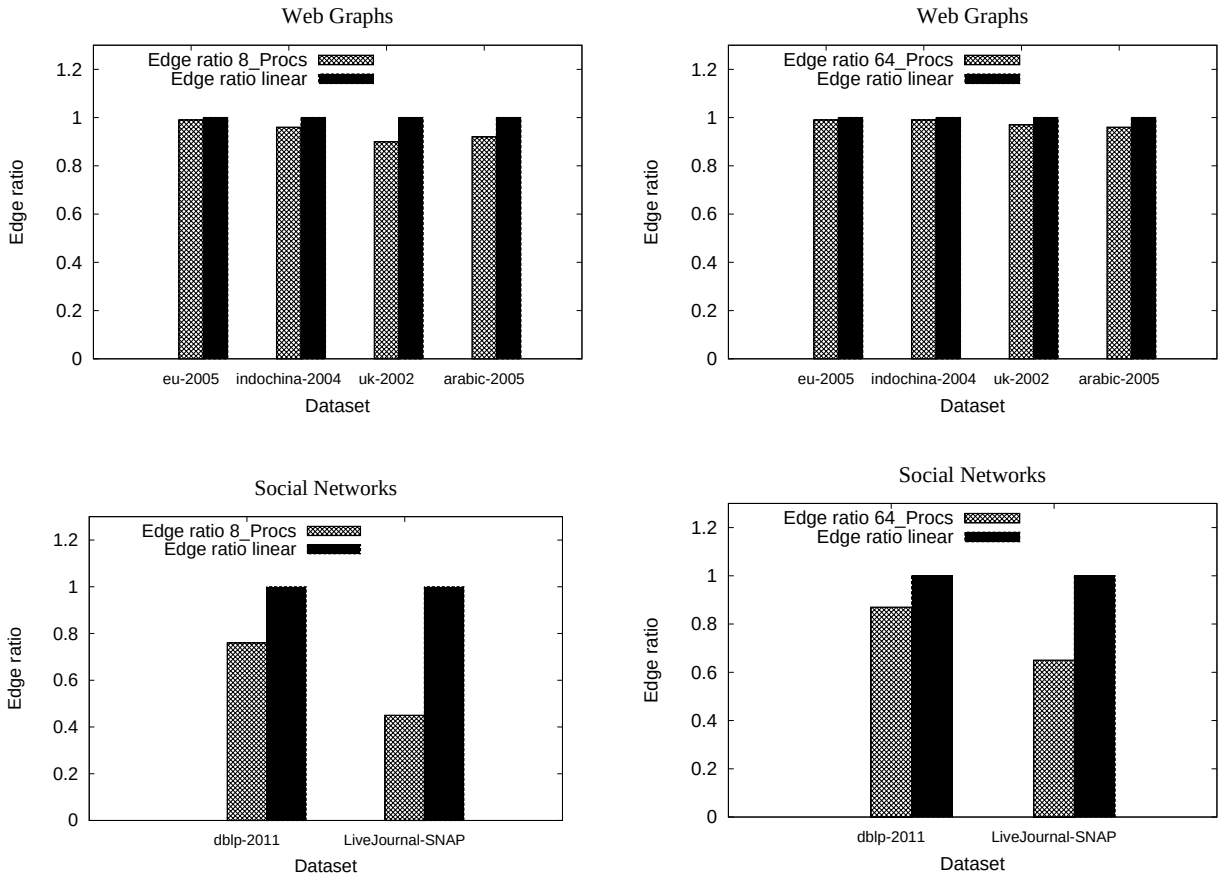


Figure 6.11: Edge ratio captured by dense subgraphs for Web graphs and social networks, using 100 and 200 iterations, for parallel extraction versus the sequential algorithm.

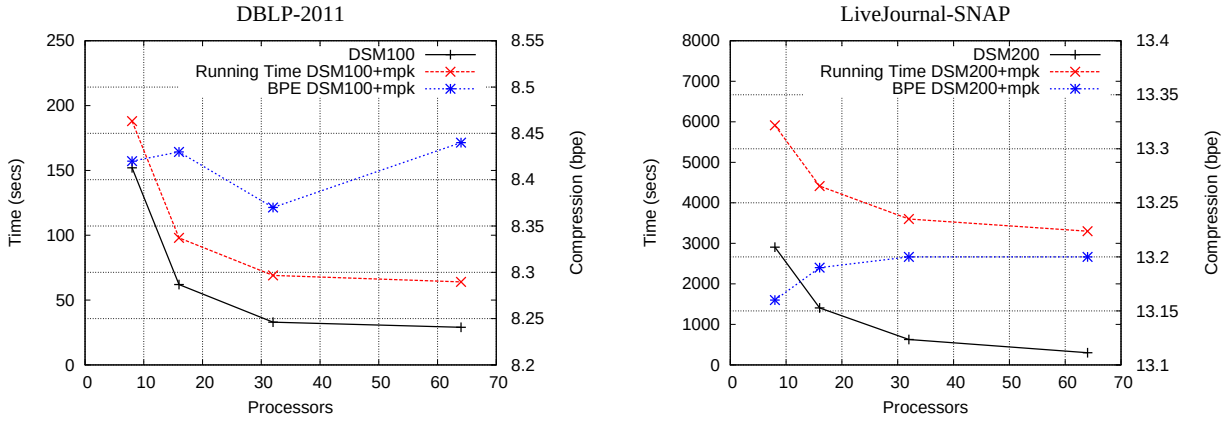


Figure 6.12: Parallel running time for extracting dense subgraphs and the bpe achieved for social networks.

algorithms are used with other sequential compression techniques such as *k2tree* [81] and *mpk* [39], the compression speed is limited by those bottlenecks.

Chapter 7

Conclusions and Future Work

This thesis aims to manage large graphs such as Web and social graphs, which are modeled by unlabeled edges. We mainly propose compression schemes that support out-neighbor and in/out-neighbor navigation. We also show that the algorithms we use for finding regularities provide efficient running times and are friendly in streaming, external memory and distributed settings.

Our work proposes different compression schemes for Web and social graphs based on dense substructures in the form of complete bipartite graphs with and without set overlaps. The proposed schemes use these patterns in two different contexts. First, they are used for reducing the number of edges of the original graph by adding virtual nodes and then applying other compression techniques. Second, they are used in a structure formed by a collection of patterns and the remaining graph, where the collection of patterns are implicitly represented using sequences and bitmaps and the remaining graph is compressed by applying other compression techniques. In addition, we have studied the performance of our algorithms in other settings, such as parallel/distributed, streaming and external memory.

7.1 Main Contributions

We first provide an approach that uses dense substructures based on complete bipartite graphs without set overlap (biclique) and with virtual nodes. This scheme consists of adding a virtual node to connect the two sets in a biclique. Applying this idea iteratively over the graph reduces the total number of edges on snapshots of real graphs between 4 and 10 times. For out-neighbor navigation, we show that using other compression schemes such as BVLLP [19, 18, 16] and AD [9] over the reduced graph provide better compression efficiency than applying the compression scheme over the original graph. We found that applying node ordering LLP over edge reduced graphs, before applying the compression scheme, improves compression slowing random access times slightly with respect to applying BVLLP over the original graph. We also found that applying AD, which includes applying BFS node ordering by default, improves compression efficiency compared with applying BVLLP over the edge reduced graph, but random access times increase. Factoring edges is also efficient when

supporting in/out-neighbor navigation applying k2tree [22] over the edge reduced graph. In both cases, retrieving original in/out-neighbors is slower since virtual nodes must be recursively replaced with original edges. This scheme provides good compression on Web graphs, but it does not work well on social networks.

Later, we improve this scheme by using dense subgraphs, which are complete bipartite graphs that allow set overlaps, where overlaps represent cliques. We show that using this pattern and BFS node ordering over edge reduced Web graphs and then k2tree provides the best Web graph compression supporting in/out-neighbors, achieving compression between 0.9 and 1.5 bpe on real Web graph snapshots. This solution also provides better access times than using URL node ordering, but it is slower than applying k2tree over the original graph using BFS ordering. Again, this approach does not work well on social networks.

We also provide a second approach for using bicliques and dense subgraphs. First, we provide a compressed structure that represents the input graph as a collection of bicliques plus the remaining graph. We represent the collection of bicliques as two symbol sequences and two bitmaps that support rank/select and access operations. This representation allows us to define different types of community queries discussed in Chapter 3. Applying this representation plus k2tree over the remaining graph provides competitive compression efficiency on Web graphs, but the approach with virtual nodes works better. However, in the case of social networks this approach improves the compression and neighbor access times achieved by k2tree with natural or LLP ordering (i.e. without using BFS ordering before applying k2tree).

We improve the representation based on bicliques using dense subgraphs, which are complete bipartite subgraphs with set overlap. In this case, we define only one symbol sequence, based on three components, and one bitmap for representing the collection of dense subgraphs. Again, we represent the remaining graph using k2tree. This representation provides better compression efficiency than the one achieved using bicliques on Web graphs. In fact, it improves the compression between 0.25 and 1 bpe on real Web graphs (as seen in Figure 4.19 in Chapter 4). In the context of social networks, it is possible to achieve better space/time efficiency than the state-of-the-art techniques, when combining this implicit representation of dense subgraphs with the remaining graph compressed with MPk [39]. One important observation is that more than 91% of the edges of Web graphs are represented in either bicliques or dense subgraphs. However, on social networks, such percentages drop to about 50-60%. Besides, the size of the subgraphs are smaller in social networks than on Web graphs. Another observation is that the compression efficiency of social networks (with bpe between 8.5 and 13), is still far from the one achieved on Web graphs (with bpe between 0.9 and 2.5). Therefore, it is an open question whether it is possible to improve the compression of social networks.

We show that finding bicliques or dense subgraphs is friendly in the context of streaming and external memory models of computation. It is possible to apply a streaming algorithm that find heavy hitters in a hierarchy for extracting large bicliques. This alternative is not very efficient in terms of memory usage and running times, but it is able to find large subgraphs. A possible way to improve this approach is to use algorithms for computing just heavy hitters in the clustering step and then apply a smarter mining algorithm to deal with possible less

precise clusters. This scheme should improve resource usage considerably. We also design and implement an external memory algorithm based on the external R-way merge-sort algorithm for clustering and sorting by clusters the input graph. This algorithm is more efficient in terms of memory, but it is about twice as slow as the original algorithm. We also show results that compare the application of the sliding-window model over a graph using URL node ordering and over the graph ordered by clusters. However, we do not apply a mining algorithm taking into account that clusters might overlap in the same window. Therefore, improving the mining algorithm probably will improve the quality of the results.

We also show that the algorithms we use for discovering and extracting bicliques and dense subgraphs are scalable in a parallel/distributed setting. We designed and implemented parallel algorithms using the BSP model. We provide a parallel algorithm for discovering dense subgraphs for applying the scheme with virtual nodes. The algorithms exploit locality and provide good scalability and speedup. We also present a parallel algorithm for extracting dense subgraphs and apply load balancing for keeping all processors busy.

7.2 Future Research

Our main lines of future research are described below:

- Finding other graph patterns that could be good for compression, taking into account the edge density, size, and number of patterns in the graph. Some of the patterns that could be explored include communities with high edge density (number of edges within the community) and low inter community edge density (number of edges across clusters). It would be worth seeing whether it is possible to use such patterns more generally than the dense subgraphs we define. A possible approach for discovering such patterns include defining an objective function that quantifies the properties of communities using the function to define an algorithm that allows one to assign nodes to communities optimizing the objective function. We could extend our discovery algorithm to consider such patterns or design other algorithms. In addition, once patterns are found, the next step for achieving compression and navigation is to design representations that favor subgraphs with high values of the objective function.
- All our results show that Web graphs are much more compressible than social networks. In this context, just recently, Chierichetti et al. [34] show that the best known web graph models require $\Omega(\log n)$ bits per edge on average. Therefore, they do not account for the compressibility of such graphs. The authors present a model for Web graphs that has $O(1)$ entropy per edge and at the same time preserves other known properties, such as power-law distribution and large number of communities. However, they also show that some models for social networks have large entropy, which suggests that these graphs are incompressible. They show that the compressibility of Web graphs comes from the small value of the average of *edge length* or the average of *gaps*. The *edge length* is the absolute value of the distance between the endpoints of an edge (i.e. $|v - u|$, of the edge (u, v)) and a *gap* is the absolute value between the vertices in an ordered adjacency list. For instance, if vertex x has an adjacency list with vertexes $z_1, ..., z_j$ in this order, the *gaps* are given by $|z_{i-1} - z_i|$, where $1 \leq i \leq j$. In fact,

other works [9, 70] that use BFS node ordering apply something similar for improving compression. In both cases BFS node ordering is applied starting at a random node. Therefore, in this context, a possible line of research is to look for better alternatives of node ordering that tend to minimize the average of *edge length* or *gaps*. We could start by looking at different ways of applying BFS for defining node orderings using synthetic graphs. For instance, we could apply undirected BFS, start at nodes with higher degrees, start at nodes participating in largest dense subgraphs, defining the order in the position of nodes in the same level based on the degree of the nodes in the level, etc. This study might help in finding patterns that are more relevant for designing graph models that are more suitable for compressing social networks.

- Finding subgraphs or patterns in graphs has many applications in different areas, such as social networks, biological data analysis such as Protein-interaction networks (PIN), recommender systems, transport routing, etc. One interesting future work will focus on studying algorithms theoretically and experimentally to find different patterns in graphs with the goal of counting, enumerating and listings different kinds of graph patterns. Some interesting patterns may be dense or sparse. Some dense patterns include kd-clique, where the shortest path from any vertex to another vertex is no more than k and paths may go out outside the pattern; k-cores, where every vertex connects to at least k other vertices in the pattern; k-plex, where each vertex is missing no more than $k-1$ edges to its neighbors; etc. Sparse patterns may include loops, parallel paths, etc. A related topic is *graph searching*, where given a pattern G_p and a graph G we must check whether G_p matches G and identify all matched subgraphs. Usually G_p is small and G is large.
- Another path of research is to extend the algorithms proposed in this work for detecting dense subgraphs for graphs with labeled edges. In this context, there are many applications that are or can be modeled by these graphs.
- Another line of future work is considering algorithms that aim not only to build compressed structures in a parallel/distributed setting, but also to solve queries over the compressed representation. The idea is to propose algorithms that allow incremental compression so that the compressed structure can grow in an online fashion where query algorithms must be able to give answers considering dynamic updates over the compressed representation.
- In chapter 5 we propose an efficient external memory algorithm for reordering the graph based on dense subgraphs. We also show that we can apply our mining algorithm over such graphs using a sliding-window model approach, where we can fix the number of edges to consider in the mining algorithm. Our preliminary work just applies the mining algorithm considering that all edges belong to the same cluster. However, this is not the general case, since clusters have different numbers of edges. Therefore, the quality of the extracted patterns can be improved by using a smarter mining algorithm that detects in an online manner whether there is more than one cluster to consider in each window.

Bibliography

[1] Hadoop information. `http://hadoop.apache.org`.

[2] J. Abello, A. L. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[3] M. Adler and M. Mitzenmacher. Towards compressing Web graphs. In *Data Compression Conference (DCC)*, pages 203–212, 2001.

[4] C. C. Aggarwal, Y. Li, P. S. Yu, and R. Jin. On dense pattern mining in graph streams. *Proceedings of the Very Large Database Endowment (PVLDB)*, 3(1):975–984, 2010.

[5] C. C. Aggarwal and H. Wang, editors. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*. Springer, 2010.

[6] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Foundations of Computer Science (FOCS)*, pages 540–549, 2004.

[7] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory BFS implementation. In *Algorithm Engineering and Experiments (ALENEX)*, 2007.

[8] V. N. Anh and A. Moffat. Local modeling for Webgraph compression. In *Data Compression Conference (DCC)*, page 519, 2010.

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

[10] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *High Performance Computing (HiPC)*, pages 465–476, 2005.

[11] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and MapReduce. *Proceedings of the Very Large Database Endowment (PVLDB)*, 5(5):454–465, 2012.

[12] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Symposium on Discrete Algorithms (SODA)*, pages 623–632, 2002.

[13] L. Becchetti, C. Castillo, D. Donato, R. A. Baeza-Yates, and S. Leonardi. Link analysis for Web spam detection. *ACM Transactions on the Web (TWEB)*, 2(1), 2008.

[14] K. Bharat, A. Z. Broder, M. R. Henzinger, P. Kumar, and S. Venkatasubramanian. The connectivity server: Fast access to linkage information on the Web. *Computer Networks*, 30(1-7):469–477, 1998.

[15] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[16] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *World Wide Web (WWW)*, pages 587–596, 2011.

[17] P. Boldi, M. Santini, and S. Vigna. Permuting web and social graphs. *Internet Mathematics*, 6(3):257–283, 2009.

[18] P. Boldi, M. Santini, and S. Vigna. Permuting Web graphs. In *Workshop on Algorithms and Models for the Web Graph (WAW)*, pages 116–126, 2009.

[19] P. Boldi and S. Vigna. The Webgraph framework I: compression techniques. In *World Wide Web (WWW)*, pages 595–602, 2004.

[20] P. Boldi and S. Vigna. Codes for the world wide web. *Internet Mathematics*, 2(4):407–429, 2005.

[21] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[22] N. R. Brisaboa, S. Ladra, and G. Navarro. k2-trees for compact Web graph representation. In *International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 18–30, 2009.

[23] N. R. Brisaboa, S. Ladra, and G. Navarro. Compact representation of Web graphs with extended functionality. *Inf. Syst.*, 39:152–174, 2014.

[24] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.

[25] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the Web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[26] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the Web. *Computer Networks*, 33(1-6):309–320, 2000.

[27] S. Brohée and J. van Helden. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC Bioinformatics*, 7:488, 2006.

[28] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[29] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to Web graph compression with communities. In *Web Search and Data Mining (WSDM)*, pages 95–106, 2008.

[30] M. Cha, A. Mislove, and P. K. Gummadi. A measurement-driven analysis of information propagation in the Flickr social network. In *World Wide Web (WWW)*, pages 721–730, 2009.

[31] A. Chakrabarti, G. Cormode, A. McGregor, and J. Thaler. Annotations in data streams. *Electronic Colloquium on Computational Complexity (ECCC)*, 19:22, 2012.

[32] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Siam International Conference on Data Mining (SDM)*, 2004.

[33] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Special Interest Group On Knowledge Discovery and Data Mining (SIGKDD)*, pages 219–228, 2009.

[34] F. Chierichetti, R. Kumar, S. Lattanzi, A. Panconesi, and P. Raghavan. Models for the compressible web. *SIAM J. Comput.*, 42(5):1777–1802, 2013.

[35] F. Chierichetti, R. Kumar, and A. Tomkins. Max-cover in MapReduce. In *World Wide Web (WWW)*, pages 231–240, 2010.

[36] Y. Cho, L. Shi, and A. Zhang. flownet: Flow-based approach for efficient analysis of complex biological networks. In *IEEE International Conference on Data Mining (ICDM)*, pages 91–100, 2009.

[37] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[38] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.

[39] F. Claude and S. Ladra. Practical representations for Web and social graphs. In *ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1185–1190, 2011.

[40] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187. Springer, 2008.

[41] F. Claude and G. Navarro. Extended compact Web graph representations. In *Algorithms and Applications*, volume 6060 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2010.

[42] F. Claude and G. Navarro. Fast and compact Web graph representations. *ACM*

Transactions on the Web (TWEB), 4(4), 2010.

[43] G. Cormode and M. Hadjieleftheriou. Finding the frequent items in streams of data. *Commun. ACM*, 52(10):97–105, 2009.

[44] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *Very Large Data Bases (VLDB)*, pages 464–475, 2003.

[45] G. Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30(1):249–278, 2005.

[46] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.

[47] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[48] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 12, 2008.

[49] R. Dementiev, L. Kettner, and P. Sanders. *STXXL: Standard template library for XXL data sets*. Springer, 2005.

[50] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Softw., Pract. Exper.*, 38(6):589–637, 2008.

[51] R. Dementiev, P. Sanders, D. Schultes, and J. F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *International Conference on Theoretical Computer Science (IFIP TCS)*, pages 195–208, 2004.

[52] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. *ACM Transactions on Algorithms*, 6(1), 2009.

[53] D. Donato, L. Laura, S. Leonardi, U. Meyer, S. Millozzi, and J. F. Sibeyn. Algorithms and experiments for the Webgraph. *J. Graph Algorithms Appl.*, 10(2):219–236, 2006.

[54] D. Donato, S. Leonardi, S. Millozzi, and P. Tsaparas. Mining the inner structure of the Web graph. In *International Workshop on the Web and Databases (WebDB)*, pages 145–150, 2005.

[55] S. V. Dongen. *Graph clustering by flow simulation*. PhD thesis, University of Uthrecht, 2000.

[56] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the Web. In *World Wide Web (WWW)*, pages 461–470, 2007.

[57] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In *Symposium on Experimental Algorithms (SEA)*, pages 364–375, 2011.

[58] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[59] G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of Web communities. In *Special Interest Group On Knowledge Discovery and Data Mining (SIGKDD)*, pages 150–160, 2000.

[60] S. Fortunato and A. Lancichinetti. Community detection algorithms: a comparative analysis: invited presentation, extended abstract. In *Performance Evaluation Methodologies and Tools (VALUETOOLS)*, page 27, 2009.

[61] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book*. Pearson Education, 2 edition, 2009.

[62] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Very Large Data Bases (VLDB)*, pages 721–732, 2005.

[63] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[64] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *International Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38. Posters., 2005.

[65] M. T. Goodrich and P. Pszona. External-memory network analysis algorithms for naturally sparse graphs. In *European Symposium on Algorithms (ESA)*, pages 664–676, 2011.

[66] S. Grabowski and W. Bieniecki. Tight and simple web graph compression for forward and reverse neighbor queries. *Discrete Applied Mathematics*, 163:298–306, 2014.

[67] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[68] M. A. Hasan, S. Salem, and M. J. Zaki. Simclus: an effective algorithm for clustering with a lower bound on similarity. *Knowl. Inf. Syst.*, 28(3):665–685, 2011.

[69] C. Hernández and G. Navarro. Compressed representation of Web and social networks via dense subgraphs. In *International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 264–276, 2012.

[70] C. Hernández and G. Navarro. Compressed representations for web and social graphs. *Knowl. Inf. Syst.*, 40(2):279–313, 2014.

[71] D. Huffman. A methods for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E.*, pages 1090–1101, 1952.

[72] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University,

1989.

[73] U. Kang, H. Tong, J. Sun, C. Lin, and C. Faloutsos. Gbase: an efficient analysis platform for large graphs. *Very Large Data Bases Journal*, 21(5):637–650, 2012.

[74] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011.

[75] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[76] I. Katriel and U. Meyer. Elementary graph algorithms in external memory. In *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 62–84. Springer, 2002.

[77] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[78] B. Krishnamurthy and J. Wang. On network-aware clustering of Web clients. In *Special Interest Group on Data Communication (SIGCOMM)*, pages 97–110, 2000.

[79] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large-scale knowledge bases from the Web. In *Very Large Data Bases (VLDB)*, pages 639–650, 1999.

[80] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the Web for emerging cyber-communities. *Computer Networks*, 31(11-16):1481–1493, 1999.

[81] S. Ladra. *Algorithms and Compressed Data Structures for Information Retrieval.* PhD thesis, University of Coruña, 2011.

[82] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Data Compression Conference (DCC)*, pages 296–305, 1999.

[83] K. Macropol and A. K. Singh. Scalable discovery of best clusters on large graphs. *Proceedings of the Very Large Database Endowment (PVLDB)*, 3(1):693–702, 2010.

[84] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Special Interest Group On Management of Data (SIGMOD)*, pages 135–146, 2010.

[85] H. Maserrat and J. Pei. Neighbor query friendly compression of social networks. In *Special Interest Group On Knowledge Discovery and Data Mining (SIGKDD)*, pages 533–542, 2010.

[86] J. McPherson, K.-L. Ma, and M. Ogawa. Discovering parametric clusters in social small-world graphs. In *Symposium On Applied Computing (SAC)*, pages 1231–1238, 2005.

[87] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. Graph structure in the web - revisited: a trick of the heavy tail. In *WWW (Companion Volume)*, pages 427–432, 2014.

[88] R. Mishra, S. Shukla, D. D. Arora, and M. Kumar. An effective comparison of graph clustering algorithms via random graphs. *International Journal of Computer Applications*, 22(1):22–27, 2011. Published by Foundation of Computer Science.

[89] A. Mislove, M. Marcon, P. K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Internet Measurement Conference (IMC)*, pages 29–42, 2007.

[90] K. Morik, A. Kaspari, M. Wurst, and M. Skirzynski. Multi-objective frequent termset clustering. *Knowl. Inf. Syst.*, 30(3):715–738, 2012.

[91] J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980.

[92] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 345–356, 2003.

[93] S. Muthukrishnan. Data streams: Algorithms and applications. In *Symposium on Discrete Algorithms (SODA)*, pages 413–413. Society for Industrial and Applied Mathematics, 2003.

[94] M. F. Pace. BSP vs MapReduce. *Procedia CS*, 9:246–255, 2012.

[95] R. Pagh. Low redundancy in static dictionaries with o(1) worst case lookup time. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 595–604, 1999.

[96] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. ANF: a fast and scalable tool for data mining in massive graphs. In *Special Interest Group On Knowledge Discovery and Data Mining (SIGKDD)*, pages 81–90, 2002.

[97] M. Patrascu. Succincter. In *Foundations of Computer Science (FOCS)*, pages 305–313, 2008.

[98] S. Raghavan and H. Garcia-Molina. Representing Web graphs. In *International Conference on Data Engineering (ICDE)*, pages 405–416, 2003.

[99] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[100] K. H. Randall, R. Stata, J. L. Wiener, and R. Wickremesinghe. The Link Database: Fast access to graphs of the Web. In *Data Compression Conference (DCC)*, pages 122–131, 2002.

[101] P. K. Reddy, M. Kitsuregawa, P. Sreekanth, and S. S. Rao. A graph based approach to extract a neighborhood customer community for collaborative filtering. In *Workshop on Databases in Networked Information Systems (DNIS)*, pages 188–200, 2002.

[102] P. Ronhovde and Z. Nussinov. Local multiresolution order in community detection. *CoRR*, abs/1208.5052, 2012.

[103] M. Ruhl. *Efficient algorithms for new computational models*. PhD thesis, Massachusetts Institute of Technology: MIT, 2001.

[104] H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara. A large-scale study of Link spam detection by graph algorithms. In *Adversarial Information Retrieval on the Web (AIRWeb)*, 2007.

[105] K. Saito, M. Kimura, K. Ohara, and H. Motoda. Efficient discovery of influential nodes for SIS models in social networks. *Knowl. Inf. Syst.*, 30(3):613–635, 2012.

[106] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proceedings of the Very Large Database Endowment (PVLDB)*, 6(6):433–444, 2013.

[107] C. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[108] B. Shao, H. Wang, and Y. Li. The trinity graph engine. Technical Report 161291, Microsoft Research, 2012.

[109] B. Shao, H. Wang, and Y. Xiao. Managing and mining large graphs: systems and implementations. In *Special Interest Group On Management of Data (SIGMOD)*, pages 589–592, 2012.

[110] T. Suel and J. Yuan. Compressing the graph structure of the Web. In *Data Compression Conference (DCC)*, pages 213–222, 2001.

[111] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *World Wide Web (WWW)*, pages 607–614, 2011.

[112] J. Thaler, M. Mitzenmacher, and T. Steinke. Hierarchical heavy hitters with the space saving algorithm. In *Algorithm Engineering and Experiments (ALENEX)*, pages 160–174, 2012.

[113] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.

[114] S. van Dongen. Graph clustering via a discrete uncoupling process. *SIAM J. Matrix Analysis Applications*, 30(1):121–141, 2008.

[115] J. S. Vitter. External memory algorithms and data structures. *ACM Comput. Surv.*,

33(2):209–271, 2001.

[116] J. Zhang. A survey on streaming algorithms for massive graphs. In *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 393–420. Springer US, 2010.