Analysis and Visualization of Hierarchical Graphs

Giorgos Kritikakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Ioannis G. Tollis*

Analysis and Visualization of Hierarchical Graphs

Abstract

In this work, we developed the Path-Based Framework (PBF). PBF is a recent graph drawing framework that resembles but also differs from the classical Sugiyama technique. PBF is based on the concepts of path and chain decomposition. We extended that idea. We draw all edges, apply edge bundling, minimize the height using a compaction technique, and reduce the width by applying algorithms similar to task scheduling. As a result, we present a generic framework suitable for hierarchical graph drawings.

Furthermore, we explore cutting-edge path and chain decomposition algorithms and applications. Our algorithms are linear or almost linear, and our results are very close to the optimum.

More precisely, we will show how to create a sub-optimal chain decomposition of a DAG (directed acyclic graph) in almost linear time. The number of vertex-disjoint chains our algorithm creates is very close to the minimum. The time complexity of our algorithm is $O(|E| + c * l)$, where c is the number of path concatenations and l is the longest path of the graph. We will give a detailed explanation in the following sections. This fundamental concept has a wide area of applications. We will focus on a few of them. We will extensively describe how to solve the transitive closure of graphs and answer queries in constant time by creating an indexing scheme. Our method needs $O(k_c * |E_{red}|)$ time and $O(k_c * |V|)$ space. The factor k_c is a sub-optimal number of chains, E_{red} is the set of non-transitive edges, and $|V|$ is the number of nodes. Moreover, we show that $|E_{red}|$ is bounded, $|E_{red}| \leq width * |V|$, and we illustrate how to find a subset of E_{tr} (the set of transitive edges) without calculating the transitive closure. Using our theory, we can enhance every transitive closure technique. We accompany our approach and algorithms with extensive experimental work. Our experiments reveal that our methods are not merely theoretically efficient since the performance is even better in practice.

Keywords: Algorithms, graph algorithms, performance, chain decomposition, path decomposition, transitive closure, transitive reduction, hierarchy, query processing, DAG, data structures, network analysis.

Τίτλος

Περίληψη

Σε αυτό το έργο έχουμε αναπτύξει το $Path-based-Framework$ (PBF). Το PBF είναι ένα πρόσφατο πλαίσιο οπτικοποίησης ιεραρχικών γραφημάτων που μοιάζει αλλά επίσης διαφέρει από το κλασικό πλαίσιο τεσσάρων φάσεων του $Sugiyama$. Το PBF βασίζεται στη ιδέα της διάσπασης του γράφου σε κανάλια και μονοπάτια. Επεκτείναμε αυτή την ιδέα. Ζωγραφίζουμε όλες τις ακμές, εφαρμόζουμε επικάλυψη ακμών, ελαχιστοποιούμε το ύψος, και μειώνουμε το πλάτος του γραφήματος εφαρμόζοντας τεχνικές όμοιες με αυτές του χρονο-προγραμματισμού εργασιών. Ως εκ τούτου, παρουσιάζουμε ένα γενικό μοντέλο οπτικοποίησης ιεραρχικών γραφημάτων.

Ακόμη,εξερευνήσαμε αλγορίθμους αιχμής για διάσπαση γράφων σε μονοπάτια και κανάλια. Οι αλγόριθμοι μας είναι γραμμικοί ή σχεδόν γραμμικοί, και τα αποτελέσματα τους είναι πολύ κοντά στο βέλτιστο. Επιπρόσθετα, αναπτύξαμε ένα πλαίσιο οπτικοποίησης ιεραρχικών γραφημάτων που βασίζεται στην διάσπαση σε μονοπάτια και κανάλια και μας βοηθάει να αποκαλύψουμε κρίσιμες πτυχές των ιεραρχιών ενός γράφου.

Ακριβέστερα, θα δείξουμε πώς να δημιουργήσουμε μια υποβέλτιστη διάσπαση σε κανάλια ενός άκυκλου κατευθυνόμενου γραφήματος σε σχεδόν γραμμικό χρόνο. Ο αριθμός των καναλιών που δημιουργεί ο αλγόριθμος μας, τα οποία δεν μοιράζονται κοινούς κόμβους, είναι πολύ κοντά στο ελάχιστο. Η χρονική πολυπλοκότητα του αλγορίθμου μας είναι $O(|E| + c * l)$, όπου c είναι ο αριθμός των καναλιών και l ο αριθμός της μεγαλύτερης διαδρομής του γράφου. Θα δώσουμε αναλυτική εξήγηση στα επόμενα κεφάλαια. Αυτή η θεμελιώδης έννοια έχει ένα ευρύ φάσμα εφαρμογών. Θα επικεντρωθούμε σε μερικές από αυτές. Θα περιγράψουμε εκτενώς πώς να λύσουμε το πρόβλημα της μεταβατικής κλειστότητας και πώς να απαντάμε ερωτήματα σε σταθερό χρόνο δημιουργώντας ένα γνωστό σχήμα από δείκτες. Η μέθοδος μας χρειάζεται $O(k_c * |E_{red}|)$ χρόνο και $O(k_c * |V|)$ χώρο. Ο όρος k_c είναι το μέγεθος μιας υποβέλτιστης διάσπασης καναλιών, ο όρος E_{red} είναι το σύνολο των μη-μεταβατικών ακμών του γράφου, και ο όρος $|V|$ υποδηλώνει τον αριθμό των κόμβων. Επιπλέον θα δείξουμε πως το $|E_{red}|$ φράζεται, $|E_{red}| \leq width * |V|$, και θα περιγράψουμε πως μπορούμε να βρούμε ένα υποσύνολο του E_{tr} (σύνολο μεταβατικών ακμών) χωρίς να υπολογίσουμε τη μεταβατική κλειστότητα. Οι μεθοδολογίες μας συνοδεύονται από εκτενής πειράματα. Τα πειράματα μας δείχνουν ότι οι αλγόριθμοι μας δεν είναι απλώς αποδοτικοί στη θεωρία. Στη πράξη η απόδοση είναι ακόμα ποιό μεγάλη.

Λέξεις κλειδιά: Αλγόριθμοι, αλγόριθμοι γράφων, απόδωση, ιεραρχίες γράφων, διάσπαση γράφου σε κανάλια, διάσπαση γράφου σε μονοπάτια, συνένωση μονοπατιών, μεταβατική κλειστότητα, συμπιεσμένη μεταβατική κλειστότητα, μεταβατική αφαίρεση, διαχείριση ερωτημάτων, σχήμα δεικτών, ιεραρχικά γραφήματα, πειραματική εργασία, ‘Ακυκλοι γράφοι, δομές δεδομένων, ανάλυση δικτύων.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον καθηγητή κ. Τόλλη Ιωάννη, ο οποίος ήταν ο επόπτης μου κατα τη διάρκεια των μεταπτυχιακών μου σπουδών. Ήταν το άτομο που μοιράζομουν τις σκέψεις μου και τα ερευνητικά μου ενδιαφέροντα. Επίσης θα ήθελα να ευχαριστήσω κ. Παναγιώτη Λιονάκη για την εξαίρετη συνεργασία μας, τα μέλη της επιτροπής, και τέλος, νιώθω την ανάγκη να ευχαριστήσω την οικογένειά μου για την αγάπη, κατανόηση, και υποστήριξή τους.

στους γονείς μου

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 On Graph Hierarchies

The arrival of new technologies, advanced sensors, and the increasing tendency of people to interact and use them, passively or actively, has led us to manage, analyze, and interpret an enormous amount of data. To achieve that, we develop more efficient and faster tools and methods. Graph theory is a critical mathematical modeling method employed in several applications of technology. In this work, we explore graph hierarchies.

Hierarchical and often directed acyclic graphs are the de facto representation for many applications in various domains including research and business. Such graphs often represent hierarchical relationships between objects in a structure or in a more complex network such as in PERT applications [21]. The analysis and visualization of these directed (often acyclic) graphs has received significant attention recently.

We developed a general-purpose hierarchical graph drawing framework that departs from the classical four-phase framework of Sugiyama and produces readable drawings. We call it Path-Based Framework since it is based on Path Decomposition. In addition to [59], we draw all edges, apply edge bundling, minimize the height using a compaction technique, and reduce the width of the drawing by applying algorithms similar to task scheduling.

Furthermore, in this work, we developed a cutting-edge chain decomposition technique. Several solutions that find the optimum chain decomposition have been proposed [44, 24, 17, 14]. Finding the optimum solution is time-consuming and not applicable for large graphs. We present a heuristic that finds a chain decomposition close to the optimal in almost linear time. Chain decomposition has a wide area of applications as in distributed computing [43, 70], in bioinformatics [10, 39], in graph visualization [59], it can facilitate answering reachability queries [44, 66, 47], and many more. We focus on answering reachability queries. We bound the transitive edges and propose linear time preprocessing steps that facilitate every transitive closure algorithm. The experiments show the efficiency of our proposals.

Answering efficiently reachability queries is an important research topic mostly driven by various arising real-world applications, such as graph databases, GIS, web mining, social network analysis, ontologies, and bioinformatics.

Definitions and Abbreviations

- **DAG:** Directed acyclic graph (DAG or dag) is a directed graph with no directed cycles.
- **Path/Chain:** In a path every vertex is connected by a direct edge to its successor, while in a chain any vertex is connected to its successor by a directed path which may be an edge. The vertices of a path/chain are in ascending topological order.
- **Paths/Chains decomposition of a DAG:** Let G = (V,E) be a DAG. A path/chain decomposition of G is a set of vertex-disjoint paths/chains. The decomposition includes all vertices of G. There is an example of a path and a chain decomposition in figure 3.1.
  - k_p: We use this abbreviation to refer to the number of paths of a path decomposition of a graph.
  - k_c: We use this abbreviation to refer to the number of chains of a chain decomposition of a graph.
- **Width:** The maximal number of mutually unreachable vertices of the graph [23].
  - The number of chains in a minimal chain decomposition of a graph is equal to its width.
- **Transitive edge:** An edge (v_1, v_2) of a DAG G is transitive if there is a path longer than one that connects v_1 and v_2.
- **DAG G(V,E):** A DAG G. V represents the set of nodes and E the set of edges.
  - E_{tr} : The set of all transitive edges. $E_{tr} \subset E$.
  - E'_{tr} : A subset of E_{tr}.
  - E_{red} : $E_{red} = E - E_{tr}$, $E_{red} \subseteq E$.
  - $G = (V, E_{red})$: The transitive reduction [6] of $G = (V, E)$. The transitive reduction is unique for DAGs. It contains the minimum number of edges needed to form the same transitive closure with $G = (V, E)$.
- **Sink vertex**: A vertex with no outgoing edges.
- **Source vertex**: A vertex with no incoming edges.

Chapter 2

Path Based Framework

2.1 Introduction

Hierarchical graphs are very important for many applications in several areas of research and business because they often represent hierarchical relationships between objects in a structure. They are directed (often acyclic) graphs and their visualization has received significant attention recently [19, 49, 56]. Sugiyama, Tagawa, and Toda proposed a four-phase framework for producing hierarchical drawings of directed graphs [67]. This is known in the literature as the Sugiyama framework. Most problems involved in the optimization of various phases of the Sugiyama framework are NP-hard. An experimental study of four algorithms specifically designed for DAGs was presented in [20]. A new framework to visualize directed graphs and their hierarchies which departs from the classical four-phase framework of Sugiyama is introduced in [58, 59]. It computes readable hierarchical visualizations in two phases by hiding (*abstracting*) some selected edges while maintaining the complete reachability information of a graph. In this paper we present polynomial time algorithms that follow the main framework of [59]. The produced drawings contain all edges of the input graph and attempt to optimize the height, width and number of bends of the resulting drawing.

The Sugiyama Framework consists of four main phases [67]: (a) Cycle Removal, (b) Layer Assignment, (c) Crossing Reduction, and (d) Horizontal Coordinate Assignment. The reader can find the details of each phase and several proposed algorithms to solve various of their problems and subproblems in [19, 49], and the recent Handbook [56]. The new framework of [59] departs from the typical Sugiyama framework and it consists of two phases: (a) Cycle Removal, (b) the path/chain decomposition and hierarchical drawing step. This framework is based on the idea of partitioning the vertices of a graph into *paths/chains*, drawing the vertices in each path vertically aligned on some x-coordinate and then drawing the edges between vertices that belong to different paths.

The Sugiyama framework has been extensively used in practice, as manifested by the fact that various systems are using it to implement hierarchical drawing techniques. Several systems such as *AGD* [60], *da Vinci* [29], *GraphViz* [34], *Graphlet* [41], *dot* [33], *OGDF* [18], and others implement this framework in order to draw directed graphs. Commercial software such the Tom Sawyer Software TS Perspectives [2] and yWorks [3] essentially use this framework in order to offer automatic visualizations of directed graphs. The comparative study of [20] concluded that the Sugiyama-style algorithms performed better in most of the metrics. For more recent information regarding this framework see [56].

Even though it is very popular, the Sugiyama framework has several limitations: as discussed above, most problems and subproblems that are used to optimize the results in various steps of each phase have turned out to be NP-hard. Additionally, several of the heuristics employed to solve these problems give results that are not bound by any approximation. Furthermore, the required manipulations in the graph often increase substantially its complexity, e.g., up to $O(nm)$ dummy vertices may be inserted in a directed graph $G = (V, E)$ with n vertices and m edges. The overall time complexity of this framework (depending upon implementation) can be as high as $O((nm)^2)$, or even higher if one chooses algorithms that require exponential time. Finally, another important limitation of this framework is the fact that heuristic solutions and decisions that are made during previous phases (e.g., crossing reduction) will influence severely the results obtained in later phases. Nevertheless, previous decisions cannot be changed in order to obtain better results.

By contrast, in the main framework of [59] most problems of the second phase can be solved in polynomial time. If a path decomposition contains k paths, the number of bends introduced is at most $O(kn)$ and the required area is at most $O(kn)$. In order to minimize the number of crossings between cross edges and path edges the authors suggest checking all possible $k!$ permutations of the k paths which may be reasonable for small values of k [58]. However, edges between non consecutive vertices in a path, called *path transitive edges* are not drawn in this framework.

In this paper we present algorithms that are based on the framework of [59] and offer experimental results comparing them to the results obtained by running the hierarchical drawing module of OGDF [18], which is based on the Sugiyama framework. Since the "cycle removal" is required in both frameworks, we focus our experiments on the case where the input graph G is acyclic (DAG). Our algorithms run in almost linear time, and provide better upper bounds than the ones given in [59]: (a) the height of the resulting drawings is equal to the length of the longest path of G, which is often significantly lower than $n - 1$. (b) The *path transitive edges* are drawn by our algorithms in such a way that the required extra number of columns is minimized for each path (see Section 3).

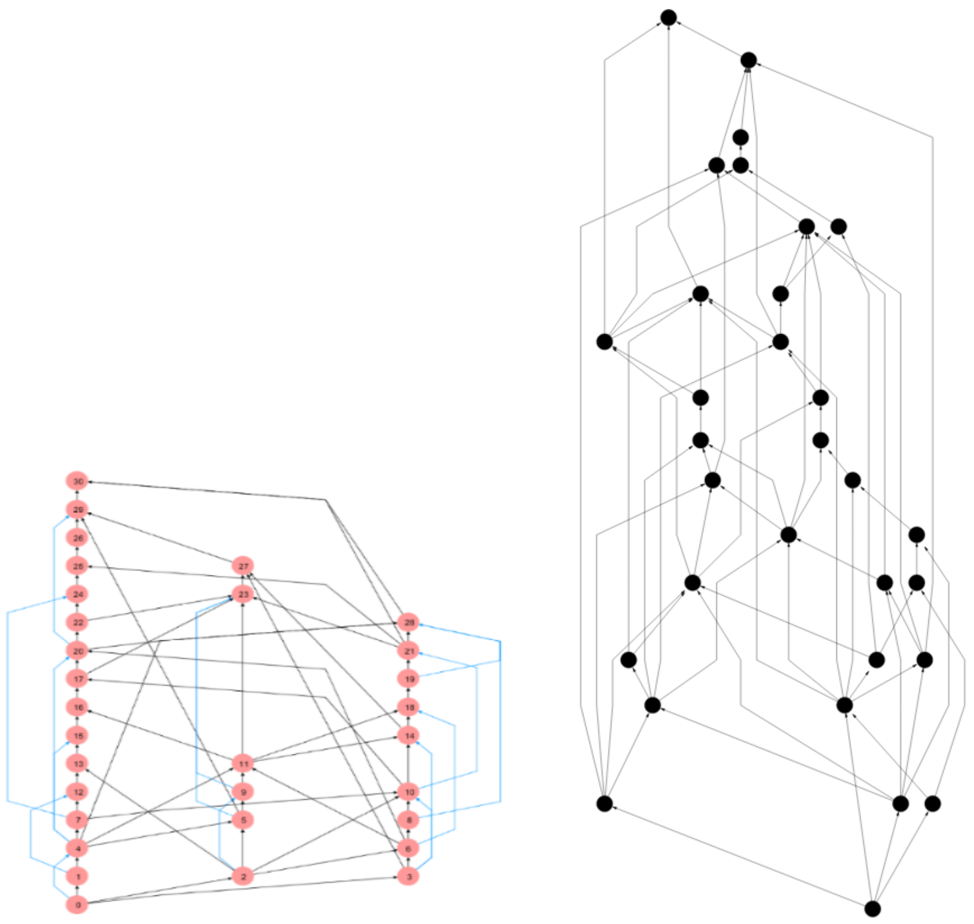
The experimental results show that the drawings produced by our algorithms have a significantly lower number of bends and are much smaller in area than the ones produced by OGDF (see Section 4). On the other hand, the drawings of OGDF have a lower number of crossings when the input graphs are relatively sparse. However, when the graphs are a bit denser (e.g., average degree higher than five) our drawings have less crossings. Of course, it is expected that OGDF would be better than our algorithms in the number of crossings since OGDF places a significant weight in minimizing crossings, whereas we do not explicitly minimize crossings. Thus our algorithms offer an interesting alternative to visualize hierarchical graphs. Finally, we present an $O(m + k \log k)$ time algorithm that computes a specific order of the paths that further reduces the total edge length, and number of crossings and bends in sparse DAGs.

2.2 Overview of the Two Frameworks

In order to motivate our discussion about the two frameworks considered in this paper we present Figure 2.1 that shows a DAG G drawn by these two frameworks: Part (a) shows a drawing Γ of G computed by our algorithms that customize the path-based framework of [59]; it is implemented in Tom Sawyer Perspectives [2] (a tool of Tom Sawyer Software); part (b) shows the drawing of G computed by OGDF. The graph consists of 31 nodes and 69 edges. The drawing computed by our algorithms has 74 crossings, 33 bends, width 14, height 16, and area 224. On the other hand, OGDF computes a drawing that has 72 crossings, 64 bends, width 42, height 16 and area 672. The width and height reported by OGDF are 961 and 2273, respectively. We had to normalized these figures in order to have a reasonable comparison, as will be discussed later. As can be observed by these two drawings, the two frameworks produce vastly different drawings with their own advantages and disadvantages.

The Path Based Hierarchical Drawing Framework follows an approach to visualize directed acyclic graphs that hides some edges and focuses on maintaining their reachability information [59]. This framework is based on the idea of partitioning the vertices of the graph G into (a minimum number of) *chains/paths*, that we call *chain/path decomposition* of G, which can be computed in polynomial time. Therefore, it is orthogonal to the Sugiyama framework in the sense that it is a vertical decomposition of G into (vertical) paths/chains. Thus, most resulting problems are *vertically contained*, which makes them simpler, and reduces their time complexity. This framework does not introduce any dummy vertices and keeps the vertices of a path *vertically aligned.* By contrast, the Sugiyama framework performs a horizontal decomposition of a graph, even though the final result is a vertical (hierarchical) visualization.

Let $S_p = \{P_1, ..., P_k\}$ be a path decomposition of G such that every vertex $v \in V$ belongs to exactly one of the paths of S_p. Any path decomposition naturally splits the edges of G into: (a) *path edges* that connect consecutive vertices in the same path, (b) *cross edges* that connect vertices that belong to different paths, and (c) *path transitive edges* that connect non-consecutive vertices in the same path. Given S_p the main algorithm of [59], call it **Algorithm PBH**, draws the vertices of each path P_i *vertically aligned* on some x-coordinate depending on the order of path P_i. There is one column between paths that is reserved for the bends (if any)



(a) (b)

Figure 2.1: In (a) we show the drawing Γ based on G as computed by Tom Sawyer Perspectives which follows our proposed framework. In (b) we show the drawing of the graph G as computed by OGDF.

of some cross edges. Therefore, the total width of the resulting drawing is $2k - 1$. The y-coordinate of each vertex is equal to its order in any topological sorting of G. Hence the height of the resulting drawing is $n - 1$. In the algorithms of [59] path transitive edges are omitted from the final drawing.

Another advantage of the Path-Based Framework is that it works for any given path decomposition. Therefore, it can be used in order to draw graphs with user-defined or application-defined paths, as is the case in many applications, see for example [21, 28]. If one desires automatically generated paths, there are several algorithms that compute a path decomposition of minimum cardinality [42, 52, 57, 65]. Using a path decomposition with a small cardinality may improve the performance of our algorithm in terms of area, bends, number of crossings and computational time. Since certain critical paths are important for many applications, it is extremely important to produce clear drawings where all such paths are vertically aligned. For the rest of this chapter, we will assume that a path decomposition of G is given as part of the input to the algorithm.

OGDF is a self-contained C++ library of graph algorithms, in particular for (but not restricted to) automatic graph drawing. The hierarchical drawing implementation of the Sugiyama framework in OGDF is implemented following [31, 64]. The Sugiyama framework in OGDF according to uses the following default choices: For the first phase of Sugiyama, it uses the *LongestPathRanking* (a ranking module that determines the layering of the graph, i.e., the assignment of vertices into layers) which implements the well-known longest-path ranking algorithm. Next, it performs crossing minimization by using *BarycenterHeuristic*. This module performs two-layer crossing minimization and is applied during the top-down and bottom-up traversals [18]. The crossing minimization is repeated 15 times, and keeps the best. Each repetition (except for the first) starts with randomly permuted nodes on each layer. Finally it computes the final coordinates with *FastHierarchyLayout* which computes the final layout of graph. The two hierarchical drawings shown in Figure 2.1 demonstrate the significant differences in philosophy between the two frameworks.

2.3 An Algorithm for Computing Compact Drawings

We present an extension of the framework of [59] by (a) compacting the drawing in the vertical direction, and (b) drawing the path transitive edges that were not drawn in [59]. This approach naturally splits the edges of G into three categories, *path edges*, *cross edges*, and *path transitive edges* that are drawn differently. This clearly adds to the understanding of the user and allows a system to show the different categories separately without altering the user's mental map.

2.3.1 Compaction

Let $G = (V, E)$ be a DAG with n vertices and m edges. Following the framework of [58, 59] the vertices of V are placed in a unique y-coordinate, which is specified by a topological sorting. Let T be the list of vertices of V in ascending order based on their y-coordinates. We start from the bottom and visit each vertex in T in ascending order. For every vertex v in this order we assign a new y-coordinate, $y(v)$, following a simple rule that compacts the height of the drawing: "If v has no incoming edges then we set its $y(v)$ to 0, else we set $y(v)$ equal to $a+1$, where a is the *highest* y-coordinate of the vertices that have edges incoming into v."

Algorithm 1 takes as input a DAG G, and a path based hierarchical drawing Γ_1 of G computed by Algorithm PBH and it produces as output a new, compacted, path based hierarchical drawing Γ_2 with height L, where L is the length of a longest path in G. Clearly this simple algorithm can be implemented in $O(n+m)$ time. Figure 2.2 shows an example of two hierarchical drawings of the same graph: Γ_1 is before compaction and Γ_2 is after compaction.

Algorithm 1 Compaction(G, Γ_1)
Input: A DAG $G = (V, E)$, and a path based hierarchical drawing Γ_1 of G computed by Algorithm PBH
Output: A compacted path based hierarchical drawing Γ_2 with height L, where L is the length of a longest path in G.

```
1: For each $v \in G$:
     • Let $E_v$ be the set of incoming edges, $e = (w, v)$, into $v$:
         a. if $E_v = \emptyset$ then:
              • $y(v)$=0
         b. else:
              • $y(v)$=max{$y$-coordinates of vertices $w$ with $(w, v) \in E_v$} + 1
```

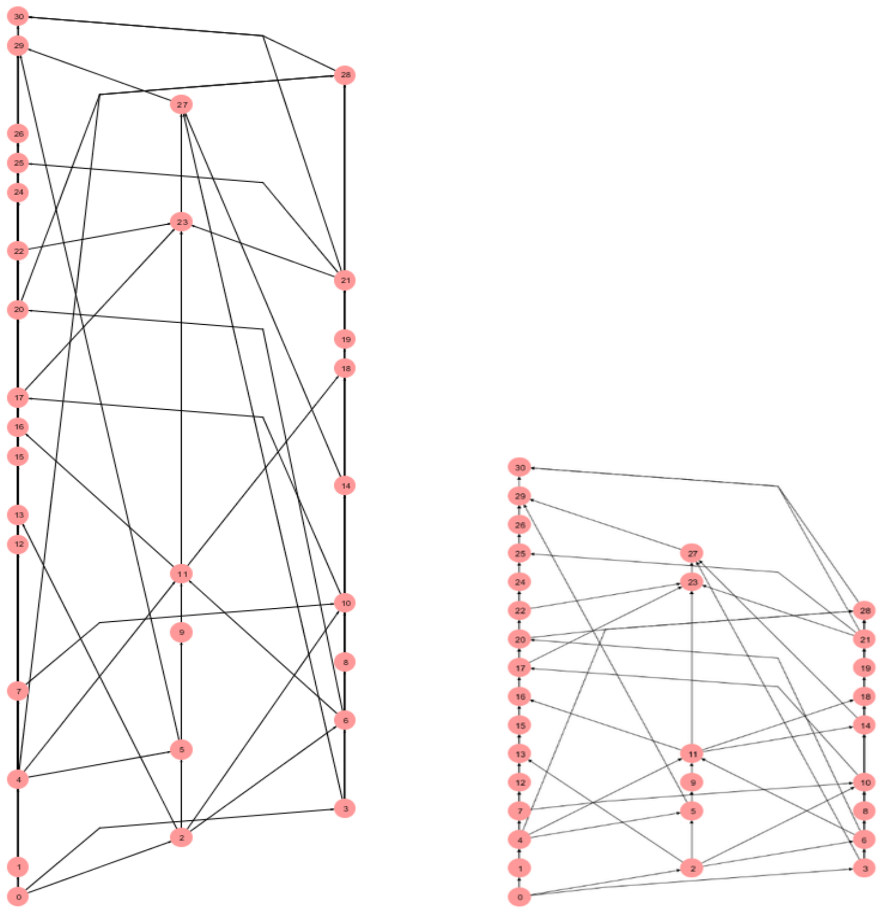
Notice that the first case of the if-statement, is executed only for the first vertex (source) of some paths. Clearly, the rest of the vertices have at least one incoming edge since they belong to some path where every vertex is connected to its predecessor. This is the case for the "else" part. The compacted y-coordinate for the rest of the vertices will always be equal to "max {y coordinates of adjacent vertices to it} +1". Based on these statements and the fact that the drawing after compaction is also a path based hierarchical drawing, we have the next two simple lemmas.

Lemma 2.3.1. *Two vertices of the same path cannot have the same y-coordinate.*

Lemma 2.3.2. *For every vertex v with $y(v) \neq 0$, there is an incoming edge into v that starts from a vertex w such that $y(v) = y(w) + 1$.*

Based on these lemmas the height of the compacted drawing of the graph G is at most L:

Theorem 2.3.3. *Let $G = (V, E)$ be a DAG with n vertices and m edges. Algorithm Compaction computes in $O(n+m)$ time a hierarchical drawing Γ_2 of G with height L, where L is equal to the length of a longest path in G.*



(a) (b)

Figure 2.2: DAG G of Figure 2.1 drawn without its path transitive edges: (a) drawing Γ_1 is computed by Algorithm PBH, and it is the input of Algorithm 1, (b) drawing Γ_2 is the output of Algorithm 1.

Proof. It is clear that the height of the resulting drawing Γ_2 cannot be lower that L, the length of the longest path, due to Lemma 2.3.1 and the fact that all edges go from a vertex with lower to a vertex with higher y-coordinate. Similarly, the height of the resulting drawing Γ_2 cannot be higher that L since that would imply that there is a y coordinate that does not contain a vertex of a longest path. In this case by the initial assumption and Lemma 2.3.2 there is another path that is longer than L. Hence the height of the resulting drawing Γ_2 is equal to L. The time complexity of Algorithm Compaction is immediate from the fact that we visit each vertex exactly once, in the order specified by T and consider all its incoming edges once. □

2.3.2 Drawing the Path Transitive Edges

An important aspect of our work is the preservation of the mental map of the user that can be expressed by the reachability information of a DAG. At this point, we highlight that for every decomposition path, we have a set of path transitive edges that are not drawn by the framework of [58, 59]. In this subsection we show how to draw these edges while preserving the user's mental map of the previous drawing. Additionally, one may interact with the drawings by hiding the path transitive edges at the click of a button without changing the user's mental map of the complete drawing.

Now we will describe an algorithm that draws the path transitive edges using the minimum extra width (minimum extra number of columns) for each decomposition path. The steps of the algorithm are briefly described as follows:

1. For every vertex of each decomposition path we calculate the indegree and outdegree based only on path transitive edges, i.e., excluding path edges and cross edges.

2. If all indegrees and outdegrees are zero the algorithm is over, if not, we select a vertex v with the highest indegree or outdegree and we bundle all the incoming or outgoing edges of v, respectively. These bundled edges are represented by an *interval* with starting and finishing points, the lowest and highest y-coordinates of the vertices, respectively.

3. Next, we insert each interval on the left side of the path on the first available column such that the interval does not overlap with another interval (see details below).

4. We remove these edges from the set of path transitive edges, update the indegree and outdegree of the vertices and repeat the selection process.

5. The intervals of the rightmost path, are inserted on the right side of the path in order to avoid potential crossing with cross edges.

6. A final, post-processing step can be applied because some crossings between intervals/bundled edges can be removed by changing the order of the columns containing them.

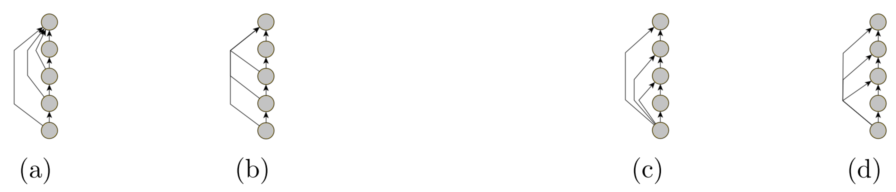


Figure 2.3: Bundling of path transitive edges: (a) incoming edges into node 13, (b) after bundling, (c) outgoing edges from node 16, (d) after bundling.

The above algorithm can be implemented to run in time $O(m + n \log n)$ by handling the updates of the indegrees and outdegrees carefully, and placing the appropriate intervals in a (Max Heap) Priority Queue. As expected, the fact that we draw the path transitive edges increases the number of bends, crossings, and area, with respect to not drawing them.

For each decomposition path, suppose we have a set of b of intervals such that each interval I has a start point, s_I, and a finish point f_I. The starting point is the position of the vertex of the interval with the lowest y-coordinate. Similarly, the finish point f_I is the position of the node of the interval with the highest y-coordinate. We follow a greedy approach in order to minimize the width (number of columns) for placing the bundled edges. The approach is similar to Task Scheduling [36], for placing the intervals. It uses the optimum number of columns and runs in $O(b \log b)$ time, for each path with b intervals. This is done by considering the intervals of each decomposition path in increasing order of their starting points. We select each interval (resp. task) according to its starting point and place it into the first column that can fit (i.e., does not intersect with another interval). If there are no available columns, we allocate a new column and place the interval there. Since the sum of all b's for all paths in a path decomposition is at most n we conclude that the algorithm runs in $O(n \log n)$ time. The proof of correctness is similar to the one for Task Scheduling in [36] and thus it is omitted here.

Theorem 2.3.4. *Let $G = (V, E)$ be a DAG with n vertices and m edges. There is an algorithm that computes a drawing of G bundling the path transitive edges for each path using the minimum number of columns (width) per path. The algorithm runs in $O(m + n \log n)$ time and computes a compact hierarchical drawing of G.*

2.4 Experimental Results and Comparisons

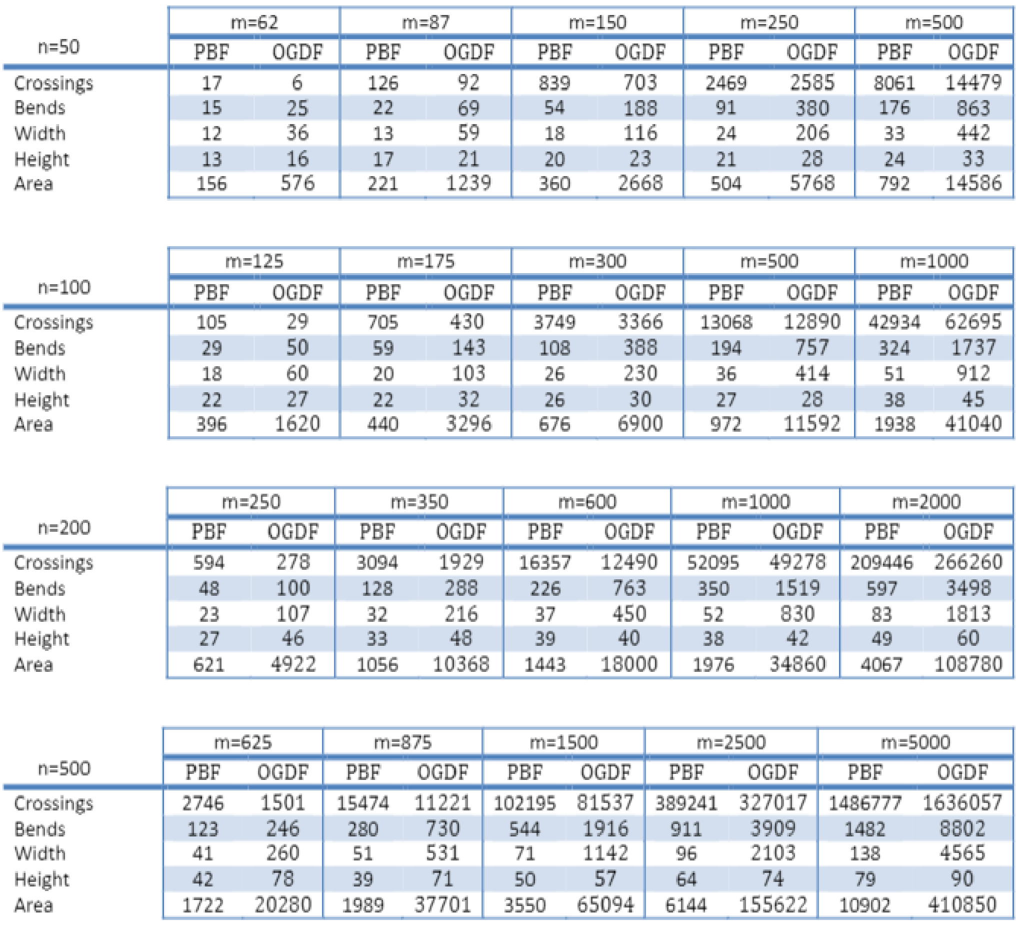
We performed experiments in order to compare the results produced by the two frameworks on different DAGs with varying number of nodes and edges. We use

20 DAGs that were produced in a random, but controlled, fashion in order to have small and large DAGs, but with a predefined average degree. Furthermore, in order to evaluate the performance of the two drawing frameworks, we use the following standard metrics:

- **Number of crossings.**
- **Number of bends.**
- **Width of the drawing:** The total number of distinct x coordinates that are used by the framework.
- **Height of the drawing:** The total number of distinct y coordinates that are used by the framework.
- **Area of the drawing:** The area of the enclosing rectangle.

Figure 2.4 shows a table that contains the results of our experiments based on these metrics for PBF as implemented in TS Perspectives [2] compared to the results produced by OGDF. In order to be consistent with the experimental settings of OGDF, we used the default parameters. In the experiments that we present in this section we see that in all cases our approach gives better results than the ones produced by OGDF with respect to the number of bends, width, height, and as expected the total area of the drawings. For the number of bends we observe that our proposed technique produces bends that are a small fraction of n, whereas OGDF produces bends that are proportional to m. The bar charts shown in Figure 2.5 show how the number of bends grows as the DAGs grow in size and average degree and provide a clear evidence that the number of bends for PBF is significantly lower than OGDF in all cases. On the other hand, the drawings of OGDF have a lower number of crossings when the input graphs are relatively sparse. However, when the graphs are a bit denser (e.g., average degree higher than five) our drawings start having less crossings. Since the two frameworks use a different coordinate system, for a fair comparison between them we chose to count as height of a drawing the number of different layers (or different y-coordinates) and as width the number of different x-coordinates of nodes and bends, used by each system. In other words, we normalize the two coordinate systems by mapping them on a "grid."

In general, our experiments show that PBF produces readable drawings with very good results almost in all metrics, except for the number of crossings. Additionally, it clearly partitions the edges into three distinct categories, and vertically aligns certain paths, which can be user defined. This can be a great advantage in certain applications and therefore it seems to be an interesting alternative, as also shown in Figure 6 for a larger example. PBF does not perform any crossing reduction step, in contrast to OGDF which offers crossing minimization algorithms by default (also required by the Sugiyama framework), which are run several times in order to keep the best result.



n=50	m=62		m=87		m=150		m=250		m=500	
	PBF	OGDF	PBF	OGDF	PBF	OGDF	PBF	OGDF	PBF	OGDF
Crossings	17	6	126	92	839	703	2469	2585	8061	14479
Bends	15	25	22	69	54	188	91	380	176	863
Width	12	36	13	59	18	116	24	206	33	442
Height	13	16	17	21	20	23	21	28	24	33
Area	156	576	221	1239	360	2668	504	5768	792	14586

n=100	m=125		m=175		m=300		m=500		m=1000	
	PBF	OGDF	PBF	OGDF	PBF	OGDF	PBF	OGDF	PBF	OGDF
Crossings	105	29	705	430	3749	3366	13068	12890	42934	62695
Bends	29	50	59	143	108	388	194	757	324	1737
Width	18	60	20	103	26	230	36	414	51	912
Height	22	27	22	32	26	30	27	28	38	45
Area	396	1620	440	3296	676	6900	972	11592	1938	41040

n=200	m=250		m=350		m=600		m=1000		m=2000	
	PBF	OGDF	PBF	OGDF	PBF	OGDF	PBF	OGDF	PBF	OGDF
Crossings	594	278	3094	1929	16357	12490	52095	49278	209446	266260
Bends	48	100	128	288	226	763	350	1519	597	3498
Width	23	107	32	216	37	450	52	830	83	1813
Height	27	46	33	48	39	40	38	42	49	60
Area	621	4922	1056	10368	1443	18000	1976	34860	4067	108780

n=500	m=625		m=875		m=1500		m=2500		m=5000	
	PBF	OGDF	PBF	OGDF	PBF	OGDF	PBF	OGDF	PBF	OGDF
Crossings	2746	1501	15474	11221	102195	81537	389241	327017	1486777	1636057
Bends	123	246	280	730	544	1916	911	3909	1482	8802
Width	41	260	51	531	71	1142	96	2103	138	4565
Height	42	78	39	71	50	57	64	74	79	90
Area	1722	20280	1989	37701	3550	65094	6144	155622	10902	410850

Figure 2.4: Results on *number of crossings, bends, width, height* and *area* for *PBF* and *OGDF* for all DAGs in our study.

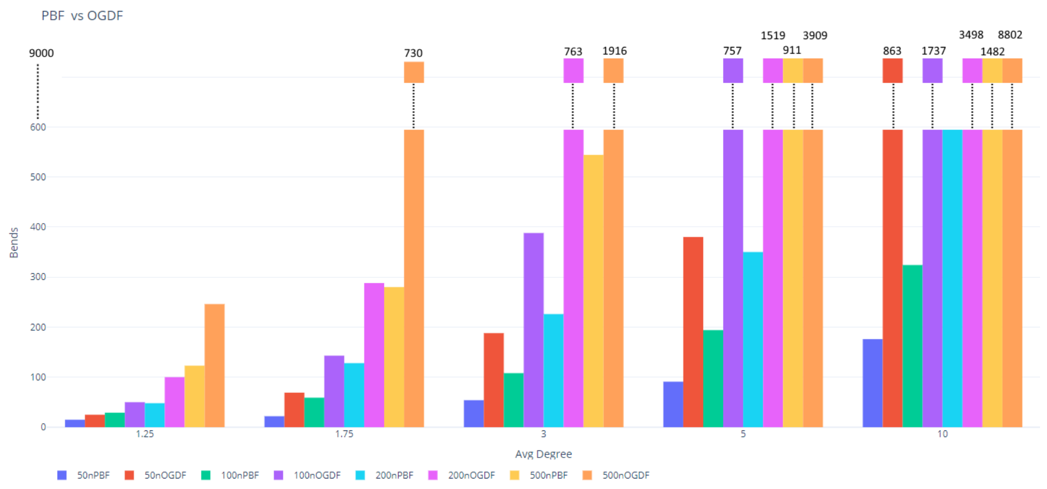


Figure 2.5: Results on the *number of bends for PBF and OGDF for all DAGs in our study.*

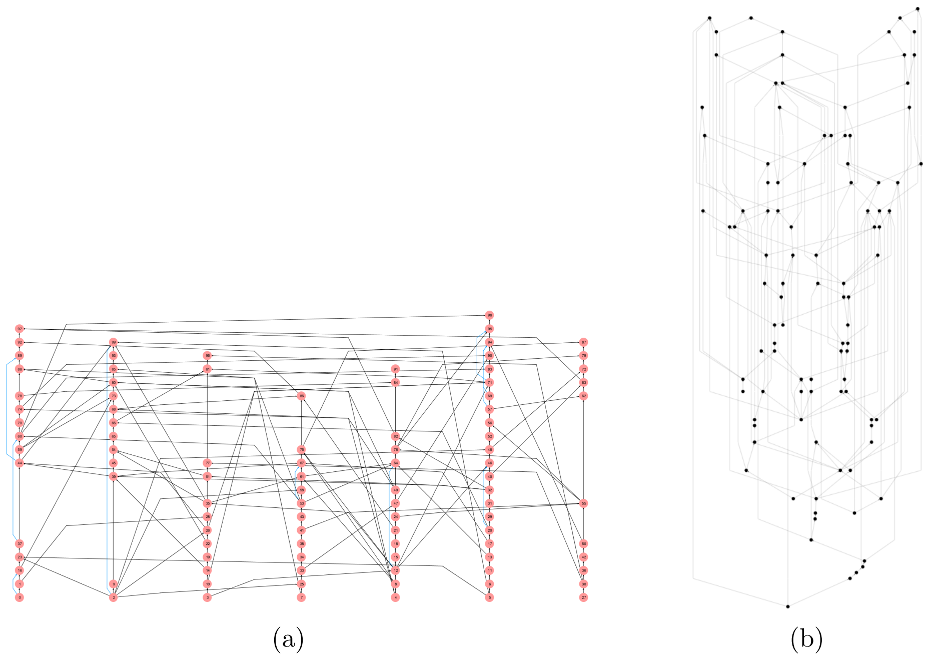


Figure 2.6: An example of a DAG with 100 nodes and 175 edges drawn with (a) PBF, and (b) OGDF.

2.4.0.1 A Heuristic for Ordering the Paths:

As described in [58], one way to minimize the number of crossings between cross edges and path edges (and path transitive edges, now) is to check all possible $k!$ permutations of the k paths. In order to reduce the number of crossings between the cross edges and path (transitive) edges, we implemented a heuristic that aims to reduce the number of paths crossed by cross edges. Our fast and simple approach is described below.

We create an undirected *path graph* by placing a node for each path P. For any pair of paths P_1 and P_2 we find the total number of cross edges between them, c, and we insert an (undirected) edge between the nodes corresponding to paths P_1 and P_2 with weight equal to c. Hence, the weight, c, of edge (P_1, P_2) is the sum of the number of cross edges directed from P_1 to P_2 plus the number of cross edges from P_2 to P_1. We do this for all cross edges between all paths. Next, we order the paths following a greedy process: We find the maximum-weight edge and we place the corresponding paths next to each other. We remove the edge from the path graph and continue with this process until it contains no edges. If we select an edge such that both paths are already placed, we simply delete this edge and proceed. If we select an edge such that one of the two paths is not already placed, then we place it at the rightmost (or leftmost) side of the placed path, depending upon which side has the least number of paths placed. This algorithm uses data structures similar to Kruskal's [51] algorithm for computing a minimum (maximum) spanning tree and it can be implemented in $O(m + k \log k)$ time. We performed some limited experiments on sparse graphs (with average degree 1.25, 1.75, and 3) using this path ordering algorithm, and we found out that the produced drawings have lower number of crossings, bends, and edge length. Unfortunately, for denser graphs the results are inconclusive.

2.5 Conclusions and Open Problems

We present algorithms and experimental results comparing two hierarchical drawing frameworks: (a) the path-based framework and (b) OGDF, which is based on the Sugiyama technique. Our compaction algorithm runs in linear time, and produces drawings with height equal to the length of a longest path of G instead of $n - 1$ which is the height of drawings produced in [59]. In this implementation we present an algorithm to bundle and draw the path transitive edges of G in $O(m + n \log n)$ time, which is an extension of the original path based framework [59]. The experimental results show that the drawings produced by our algorithms have significantly lower number of bends and are much smaller in area than the ones produced by OGDF, but they have more crossings for sparse graphs. Thus our algorithms offer an interesting alternative when we visualize hierarchical graphs. They focus on showing important aspects of the graph such as critical paths, path transitive edges, and cross edges. For this reason, this framework is particularly useful in graph visualization systems that encourage user interaction.

There are several interesting open problems: 1) Find better algorithms to order the paths. 2) Find techniques to reduce the number of crossings. 3) Allow some extra vertical space between selected vertices in order to make the visualization more visually appealing.

Chapter 3

Path/Chain Decomposition

3.1 Introduction

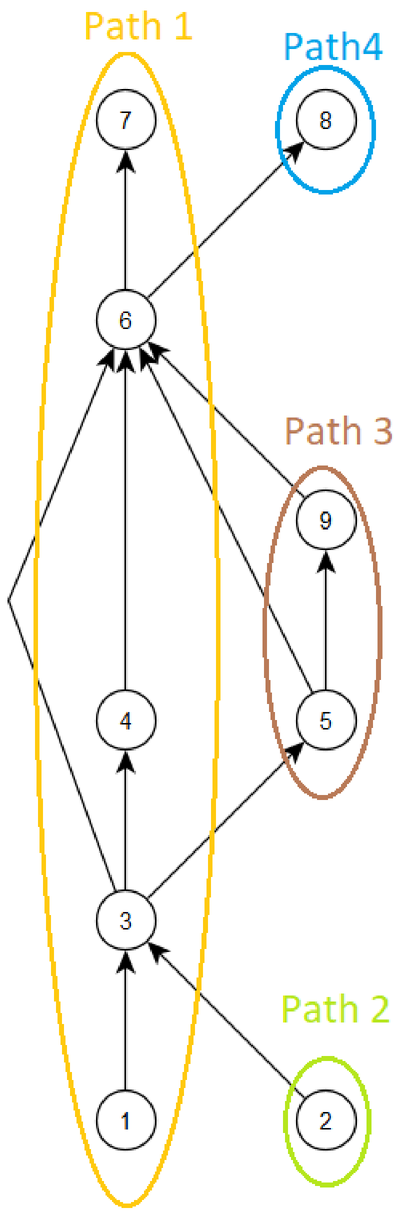
Searching for efficient ways to decompose the graph into chains, we could not find an efficient solution that scales on large graphs. An efficient chain decomposition has many applications and can facilitate many algorithms and systems. In this work, we develop an almost linear chain decomposition algorithm that produces a set of chains with almost minimum cardinality. We use the notion of chain decomposition to offer bounds to the transitive edges and explore how it facilitates in transitive closure problem.

In Section 3.2, we present path decomposition approaches, and in Section 3.3 we present chain decomposition and path concatenation. Additionally, we show experiments and evaluate the performance of our heuristic. Furthermore, we examine a few outcomes. In section 3.4, we prove that $|E_{red}| \leq width * |V|$, and see how we can in linear time, remove a subset of transitive edges and bound $|E - E'_{tr}|$ by $k * |V|$ given a path/chain decomposition of size k. Finally, section 3.5 demonstrates how to build a known indexing scheme for computing transitive closure of a graph and we report experimental results.

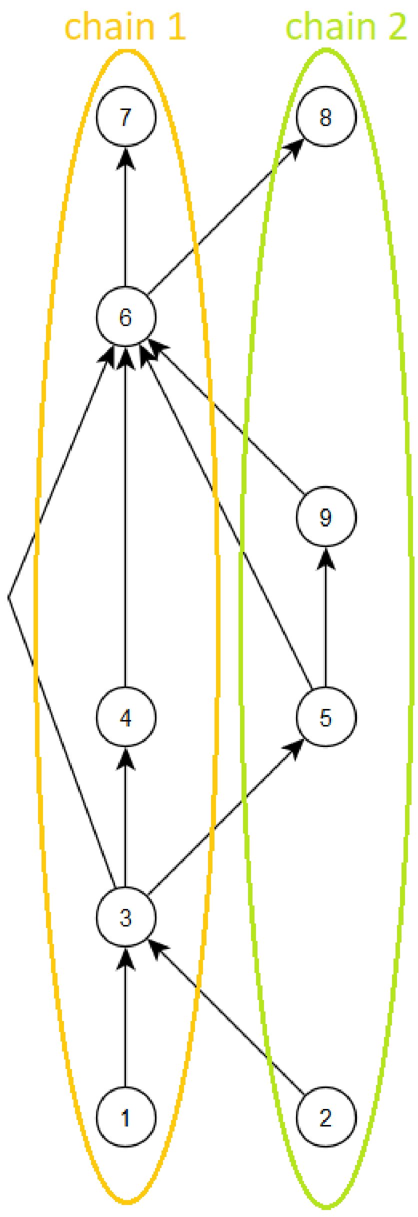
We conducted all the experiments using a laptop PC (Intel(R) Core(TM) i5-6200U CPU, 8 GB of main memory).

3.2 Path Decomposition

Jagadish in [44] categorized path decomposition techniques into two categories. Chain Order Heuristics and Node Order Heuristics. The first constructs the paths one by one, while the second creates the paths in parallel. More precisely, in [44], Jagadish presented chain decomposition heuristics based on Chain Order Heuristic and Node Order Heuristic. He utilized a list of all successors and not only the immediate for each vertex. However, his algorithms require $O(n^2)$ time using the precomputed transitive closure. That is inefficient, especially for large graphs, and we will not examine them further. Our heuristic does not need any



(a) A path decomposition of a graph. It consists of 4 paths.



(b) A chain decomposition of the same graph. It consists of 2 chains.

Figure 3.1: On the left, there is a path decomposition of graph G. On the right, a chain decomposition of G.

precomputation of the transitive closure and decomposes the graph into a number k_c of chains in $O(|E| + c * l)$ time which in practice is almost linear. Factor c is the number of concatenations, and l is the length of a longest path of the graph. We will describe our technique in detail in the next section.

In this section, we describe the linear time algorithms for path decomposition. We use topological sorting and examine the vertices in ascending order.

Chain Order Heuristic

The chain-order heuristic starts from a vertex and keeps on extending the path to the extent possible. The path ends when no more unused immediate successors can be found. As you can see in Algorithm 2, the first for loop finds an unused vertex and creates a path. The inner while loop extends the path.

Algorithm 2 Path Decomposition (CO)

```
procedure ChainOrderHeuristic($G, T$)
INPUT: A DAG $G = (V, E)$, and a topological sorting $T(v_1, ..., v_i, ..., v_N)$ of G
OUTPUT: A path decomposition of G
    $K \leftarrow \emptyset$ //Set of paths
    Mark all nodes unused
    for every unused vertex $v_i \in T$ in ascending topological order do
        $current \leftarrow v_i$
        $C \leftarrow$ new Chain()
        Add $current$ to $C$
        while there is an unused immediate successor $s$ of the current node do
            add $s$ to $C$
            $current \leftarrow s$
        end while
        add $C$ to $K$
    end for
end procedure
```

Node Order Heuristic

The node-order heuristic examines each node and assigns it to an existing path. If there is no matching, then a new path is created for the vertex. Algorithm 3 illustrates the node order heuristic.

Algorithm 3 Path Decomposition (NO)

```
procedure NodeOrderHeuristic($G, T$)
INPUT: A DAG $G = (V, E)$, and a topological sorting $T(v_1, ..., v_i, ..., v_N)$ of G
OUTPUT: A path decomposition of G
    $K \leftarrow \emptyset$ //Set of paths
    for every vertex $v_i \in T$ in ascending topological order do
        if $v_i$ is an immediate successor of the last node of a chain C then
            add $v_i$ to $C$
        else
            $C \leftarrow$ new Chain()
            add $v_i$ to $C$
            add $C$ to $K$
        end if
    end for
end procedure
```

3.3 Chain Decomposition

In this section, we present a path concatenation technique that takes as input any path decomposition and constructs a chain decomposition in $O(|E| + c * l)$ time, where c is the number of path concatenations and l is the longest path of the graph. In order to apply our path concatenation algorithm, we first find a path decomposition of the graph. We can use an already known linear-time algorithm based on Node-Order Heuristic or Chain Order Heuristic.

3.3.1 Path Concatenation

Our concatenation algorithm can work for any path decomposition. Given a graph $G = (V, E)$ and its path decomposition D_p with k_p paths we build a chain decomposition of k_c chains in $O(|E| + (k_p - k_c) * l)$ time, where l is the longest path of G. Since each concatenation reduces the number of chains by one, factor $(k_p - k_c)$ is the number of path concatenations.

For every path, we start a reverse DFS lookup function from the first vertex of the chain, looking for the last vertex of another chain traversing the edges backward. The DFS lookup function is the well-known depth-first search graph traversal for path finding. If the DFS lookup function detects the last vertex of a chain, then it concatenates the chains. If we do merely that the algorithm will run in $O(k_p * |E|)$ since we run k_p DFS functions. In our case, every DFS lookup function will take advantage of the previous DFS lookup functions' executions. DFS for path finding returns the path between the source vertex and the target vertex. In our case, the path between the first vertex of a chain and the last vertex of another chain. Hence, every execution goes through a set of vertices V_i that can be split into two vertex disjoint sets, R_i and P_i. In P_i belong the vertices of the path from the source vertex to the destination vertex. In R_i belong every vertex in $V_i - P_i$. If no path is found then $V_i = R_i$ and $P_i = \emptyset$.

Notice that every vertex in the set R_i is not the last vertex of a chain. If it was then it would belong to P_i and not to R_i. The same way, for every vertex in R_i, all its predecessors are in R_i too. Hence, if a forthcoming reverse DFS lookup function meets a vertex of R_i, there is no reason to proceed with its predecessors. All the above are basic DFS theory.

Algorithm 4 Concatenation

```
procedure CONCATENATION($G, D$)
INPUT: A DAG $G = (V, E)$, and a path decomposition D of G
OUTPUT: A chain decomposition of G
    for each path: $p_i \in D$ do
        $f_i \leftarrow$ first vertex of $p_i$
        $(R_i, P_i) \leftarrow$ reverse_DFS_lookup($G, f_i$)
        if $P_i \neq \emptyset$ then
            $l_i \leftarrow$ destination vertex of $P_i$ //Last vertex of a path
            Merge_Paths( $l_i$, $f_i$ )
        end if
        $G \leftarrow G \setminus R_i$
    end for
end procedure
```

Algorithm 4 shows our chain concatenation technique. As you see, the DFS lookup function is invoked for every starting vertex of a path. Every reverse DFS lookup function goes through the set R_i and the set P_i, examining the nodes and their incident edges. P_i is the path from the first vertex of a chain to the last vertex of another. The set R_i contains all of the vertices the function went through except the vertices of P_i.

Theorem 3.3.1. *The time complexity of Algorithm 4 is $O(|E| + (k_p - k_c) * l)$.*

Proof. Assume that we have k_p paths. We call k_p times the reverse_DFS_lookup function. Hence, we have (R_i, P_i) sets, $0 \leq i < k_p$. In every loop, we delete the vertices of R_i. Hence, $R_i \cap R_j = \emptyset$,$0 \leq i, j < k_p$ and $i \neq j$. We conclude that $\bigcup_{i=0}^{k_p-1} R_i \subseteq N$ and $\sum_{i=0}^{k_p-1} |R_i| \leq |N|$.

Path P_i, $0 \leq i < k_p$, is not empty if and only if concatenation has occurred. Hence, $\sum_{i=0}^{k_p-1} |P_i| \leq c * l$ where c is the number of concatenations and l is the longest path of the graph. Since every concatenation reduces the number of chains by one, we have $c = k_p - k_c$. □

3.3.2 Chain Decomposition Heuristic: A Better Approach

Previously, we described how to produce a chain decomposition applying a concatenation step after path decomposition. At this point, we will demonstrate an approach which not only runs in $O(|E| + c * l)$ time but it also finds a close to optimal chain decomposition.

We present Algorithm 5, which is a variation of Node Order Heuristic (Algorithm 3). It is like the Node Order heuristic but with two additions. The first is that when we visit a vertex with out-degree 1, we add its unique immediate successor to its path. The second is that we do not merely search for the first available immediate predecessor that is the last vertex of a path. Instead of the first available vertex, we choose an available vertex with the highest out-degree. Our aim using this heuristic is to create a chain construction in which more concatenations will occur. Algorithm 4 goes through all vertices. For every vertex, it examines all the outgoing (line 8) and all the incoming edges (line 19). Hence, the time complexity is linear.

Algorithm 6 illustrates our chain decomposition which is a combination of Algorithm 5 with chain concatenation. The only addition to Algorithm 4 is the if-statement of line 10 and its block. If we do not find an immediate predecessor, we search all predecessors using the reverse_DFS_lookup function. The differentiation of our concatenation is that it does not take part as a post-processing step. It is applied on time when the algorithm does not find an immediate predecessor that is the last vertex of a chain. We do it to avoid transitive edges that could lead to false matches.

3.3.3 Experiments

In this section, we present experiments on graphs created by NetworkX [40]. We used three different random graph generator models. Erdos-Renyi, Barabasi, and Watts-Strogatz model. Additionally, we use Path-Based DAG Model. For every model, we created 12 graphs. Six of 5000 nodes and six graphs of 10000 nodes and average degree 5,10,20,40,80, and 160. We examine the performance of heuristics in terms of the chains' number. We compute the minimum set of chains by using the Fulkerson's method [24]. Our aim is to reveal the behavior of the width and the behavior of heuristics used on graphs of these models. We noticed that the graphs generated by the same generator with the same parameters have insignificant width deviation (In three graphs created with the same parameters, the percentage of deviation on Erdos-Renyi and Path-Based model is about 5% and Barabasi model 10%. The Watts-Strogatz model deviation is higher, but that happens because the width has low values).

Fulkerson's method:

1. Construct transitive closure $G^*(V, E')$ of the graph, where $V = \{v_1, ..., v_n\}$.

Algorithm 5 Path Decomposition (H3)

```
1: procedure NODE-ORDER BASED VARIATION($G, T$)
   INPUT: A DAG $G = (V, E)$, and a topological sorting $T(v_1, ..., v_i, ..., v_N)$ of G
   OUTPUT: A path decomposition of G
2:     $K \leftarrow \emptyset$ //Set of paths
3:     for every vertex $v_i \in T$ in ascending topological order do
4:         Chain $C$
5:         if $u_i$ is assigned to a chain then
6:             $C \leftarrow u_i$'s chain
7:         else if $v_i$ is not assigned to a chain then
8:             $l_i \leftarrow$ choose the immediate predecessor with the lowest outdegree
9:                 that is the last vertex of a chain
10:            if $l_i \neq$ null then
11:                $C \leftarrow$ path indicated by $l_i$
12:                add $v_i$ to $C$
13:            else
14:                $C \leftarrow$ new Chain()
15:                add $v_i$ to $C$
16:            end if
17:            add $C$ to $K$
18:        end if
19:        if there is an immediate successor $s_i$ of $u_i$ with in-degree 1 then
20:            add $s_i$ to $C$
21:        end if
22:    end for
23: end procedure
```

2. Construct a bipartite graph B with bipartite (V_1, V_2), where $V1 = \{x_1, x_2, ..., x_n\}$, $V2 = \{y_1, y_2, ..., y_n\}$. An edge (x_i, y_j) is formed whenever $(v_i, v_j) \in E'$

3. Find a maximal matching M of B. The width of the graph is $n - |M|$. In order to construct the minimum set of chains, for any two edges $e_1, e_2 \in M$, if $e_1 = (x_i, y_t)$ and $e_2 = (x_t, y_j)$ then connect e_1 to e_2

Random Graph Generators:

- **Erdős-Rényi model** [27]: The generator returns a binomial graph. The generator's parameters are two, the number of nodes n and a probability p. Every edge in this model has a probability p to be formed.

- **Barabási–Albert** [9]: A graph of n nodes is grown by attaching new nodes each with m edges that are preferentially attached to existing nodes with high degree. The factors n and m are parameters to the algorithm.

Algorithm 6 Chain Decomposition (H3 conc.)

```
1: procedure NodeOrder based variation with concatenation($G, T$)
   INPUT: A DAG $G = (V, E)$, and a topological sorting $T(v_1, ..., v_i, ..., v_N)$ of G
   OUTPUT: A path decomposition of G
2:     $K \leftarrow \emptyset$ //Set of paths
3:     for every vertex $v_i \in T$ in ascending topological order do
4:         Chain $C$
5:         if $u_i$ is assigned to a chain then
6:             $C \leftarrow u_i$'s chain
7:         else if $v_i$ is not assigned to a chain then
8:             $l_i \leftarrow$ choose the immediate predecessor with the lowest outdegree
9:                 that is the last vertex of a chain
10:            if $l_i$ = null then
11:                $(R_i, P_i) \leftarrow$ reverse_DFS_lookup($G, u_i$)
12:                if $P_i \neq \emptyset$ then
13:                    $l_i \leftarrow$ destination vertex of $P_i$
14:                end if
15:                $G \leftarrow G \setminus R_i$
16:            end if
17:            if $l_i \neq$ null then
18:                $C \leftarrow$ path indicated by $l_i$
19:                add $v_i$ to $C$
20:            else
21:                $C \leftarrow$ new Chain()
22:                add $v_i$ to $C$
23:            end if
24:            add $C$ to $K$
25:        end if
26:        if there is an immediate successor $s_i$ of $u_i$ with in-degree 1 then
27:            add $s_i$ to $C$
28:        end if
29:    end for
30: end procedure
```

- **Watts–Strogatz** [76]: This model returns a Watts–Strogatz small-world graph. First it creates a ring over n nodes. Then each node in the ring is joined to its k nearest neighbors. Then shortcuts are created by replacing some edges as follows: for each edge (u,v) in the underlying "n-ring with k nearest neighbors" with probability b replace it with a new edge (u,w) with uniformly random choice of existing node w. The factors n,k, and b are the generator's parameters.

- **Path-Based DAG Model** [55]: In this model, graphs are randomly generated based on a number of predefined but randomly created paths.

To make the directed graphs acyclic, only edges from low to high ID are inserted. For more info about the generators see networkx documentation [40].

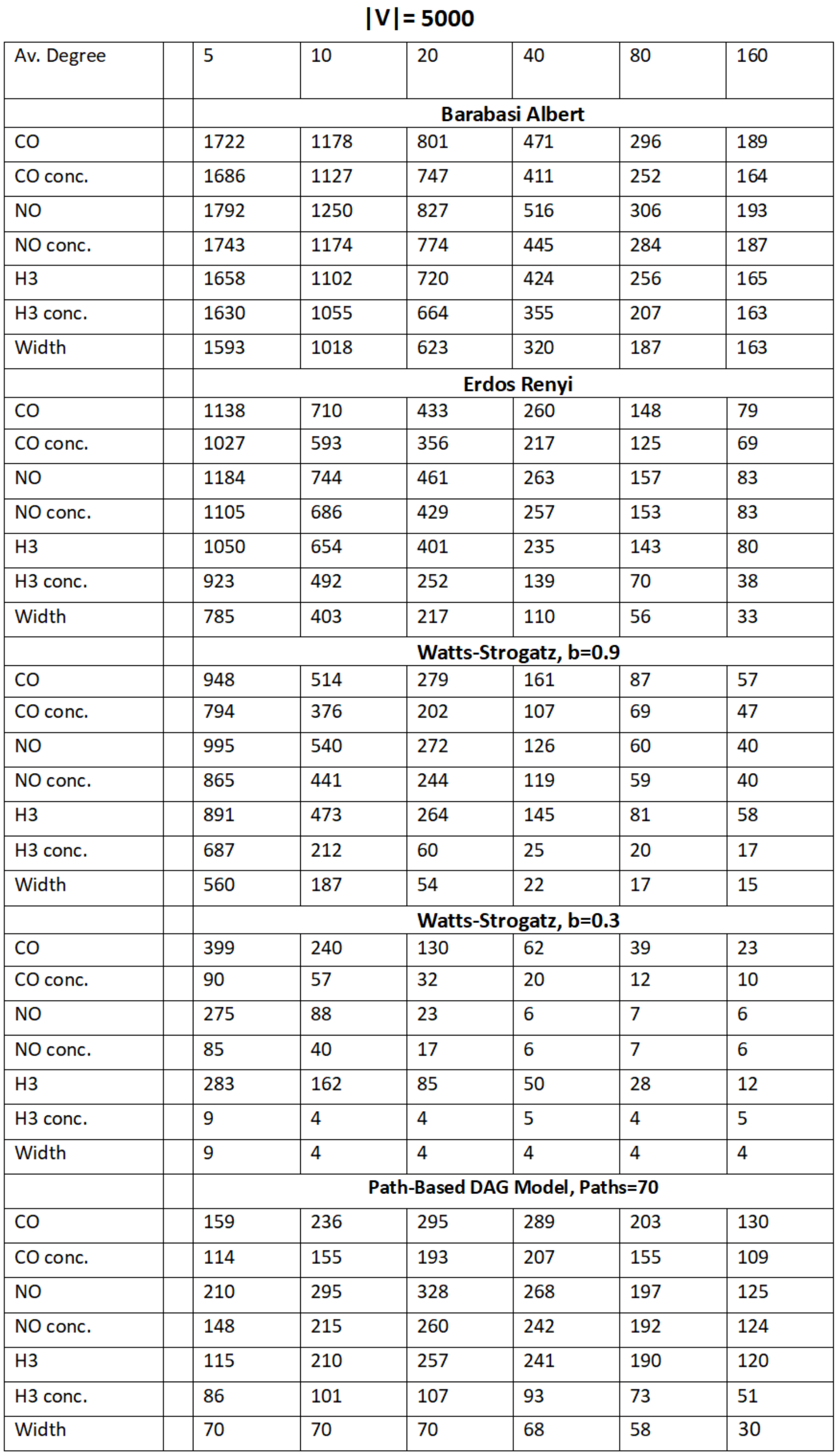
Table 3.1 shows the width and the number of chains created by the heuristics for every graph of 5000 nodes. Table 3.2 shows the same for graphs of 10000 nodes. The tables' abbreviations are explained below:

- **CO**: Path decomposition using Chain Order Heuristic. (Algorithm 2)
- **CO conc.**: Chain decomposition using Chain Order Heuristic and our concatenation technique. (Algorithm 2 followed by Algorithm 4)
- **NO**: Path decomposition using Node Order Heuristic. (Algorithm 3)
- **NO conc.**: Chain decomposition using Node Order Heuristic and our concatenation technique. (Algorithm 3 followed by Algorithm 4)
- **H3**: Path decomposition using our Node Order Heuristic variation from section 3.3.2. (Algorithm 5)
- **H3 conc.**: Chain decomposition using our technique from section (Algorithm 6)
- **Width**: The width of the graph (Fulkerson's method).

As we see, in both tables our chain decomposition (H3 conc.) performs better than the others since it produces fewer chains. To visualize how close is the outcome of our heuristic to the width, we made some charts. In Figures 3.4, 3.5, and 3.6, you can see how close is the blue line to the red one for Erdos Renyi, Barabsi Albert, and Watts Strogatz model. The red line indicates the width and the blue the chains produced by our technique.

Furthermore, we explore the behavior of the width on these models. Notice that the Barabasi Albert model produces graphs with a larger width than Erdos-Renyi. Respectively, the Erdos-Renyi model creates graphs with a larger width than Watts-Strogatz. For the Watts Strogatz model, we create two sets of graphs. The first has probability b equals 0.9 and the second 0.3. If the probability b of rewiring an edge is 0, the width would be one. That happens because the generator initially creates a path that goes through all vertices. As probability b grows, the width grows. That's the reason we choose a low and a high probability. Figure 3.2a and 3.2b demonstrates the behavior of the width for each model on the graphs of 5000 and 10000 nodes. Another interesting observation is that the width of the Erdos Renyi model follows the curve $width = \frac{\text{nodes}}{\text{average degree}}$.

All heuristics run in few milliseconds thus we do not elaborate on running time. In the following sections, we present partially run-time metrics in tables 3.4,3.5, and 3.3.



|V|= 5000

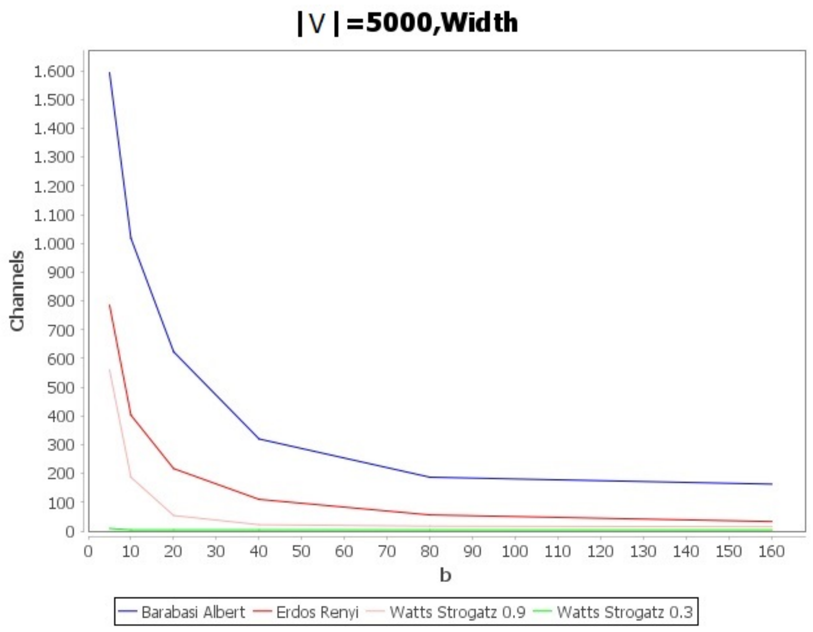
Av. Degree		5	10	20	40	80	160
		Barabasi Albert					
CO		1722	1178	801	471	296	189
CO conc.		1686	1127	747	411	252	164
NO		1792	1250	827	516	306	193
NO conc.		1743	1174	774	445	284	187
H3		1658	1102	720	424	256	165
H3 conc.		1630	1055	664	355	207	163
Width		1593	1018	623	320	187	163
		Erdos Renyi					
CO		1138	710	433	260	148	79
CO conc.		1027	593	356	217	125	69
NO		1184	744	461	263	157	83
NO conc.		1105	686	429	257	153	83
H3		1050	654	401	235	143	80
H3 conc.		923	492	252	139	70	38
Width		785	403	217	110	56	33
		Watts-Strogatz, b=0.9					
CO		948	514	279	161	87	57
CO conc.		794	376	202	107	69	47
NO		995	540	272	126	60	40
NO conc.		865	441	244	119	59	40
H3		891	473	264	145	81	58
H3 conc.		687	212	60	25	20	17
Width		560	187	54	22	17	15
		Watts-Strogatz, b=0.3					
CO		399	240	130	62	39	23
CO conc.		90	57	32	20	12	10
NO		275	88	23	6	7	6
NO conc.		85	40	17	6	7	6
H3		283	162	85	50	28	12
H3 conc.		9	4	4	5	4	5
Width		9	4	4	4	4	4
		Path-Based DAG Model, Paths=70					
CO		159	236	295	289	203	130
CO conc.		114	155	193	207	155	109
NO		210	295	328	268	197	125
NO conc.		148	215	260	242	192	124
H3		115	210	257	241	190	120
H3 conc.		86	101	107	93	73	51
Width		70	70	70	68	58	30

Table 3.1: Comparing path and chain decomposition algorithms on graphs with 5000 nodes.

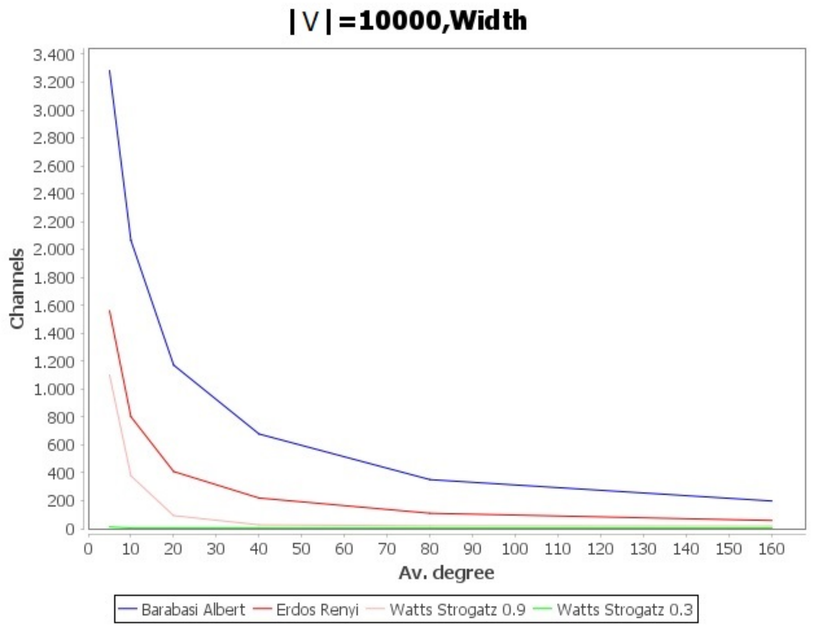
|V|=10000

Av. Degree		5	10	20	40	80	160
		Barabasi Albert					
CO		3501	2401	1537	985	586	357
CO conc.		3441	2301	1415	865	500	294
NO		3635	2519	1645	1033	625	387
NO conc.		3549	2413	1515	959	563	345
H3		3385	2257	1411	911	535	321
H3 conc.		3341	2159	1264	752	400	228
Width		3282	2066	1172	678	351	198
		Erdos Renyi					
CO		2283	1432	871	513	294	165
CO conc.		2015	1213	730	428	251	145
NO		2369	1517	891	531	294	165
NO conc.		2172	1383	833	507	290	163
H3		2135	1325	804	482	272	166
H3 conc.		1837	1003	516	271	139	72
Width		1561	802	409	219	110	58
		Watts-Strogatz, b=0.9					
CO		1869	1064	566	306	170	92
CO conc.		1575	771	381	218	119	72
NO		1975	1083	528	238	101	56
NO conc.		1717	894	455	218	92	56
H3		1748	975	524	269	150	95
H3 conc.		1332	447	100	29	24	22
Width		1101	378	93	27	20	18
		Watts-Strogatz, b=0.3					
CO		816	434	242	133	78	37
CO conc.		184	122	57	38	24	17
NO		565	171	37	10	7	7
NO conc.		165	72	24	9	7	7
H3		534	299	180	96	34	34
H3 conc.		12	4	4	4	4	4
Width		12	4	4	4	4	4
		Path-Based DAG Model, Paths=100					
CO		234	389	507	482	371	250
CO conc.		161	254	304	323	281	207
NO		305	504	550	512	370	238
NO conc.		205	343	440	448	343	227
H3		168	316	443	427	337	232
H3 conc.		125	141	153	142	120	89
Width		100	100	100	99	90	47

Table 3.2: Comparing path and chain decomposition algorithms on graphs with 10000 nodes.



(a) The width curve on graphs of 5000 nodes.



(b) The width curve on graphs of 10000 nodes.

Figure 3.2: The width curve on graphs of 5000 and 10000 nodes using three different models.

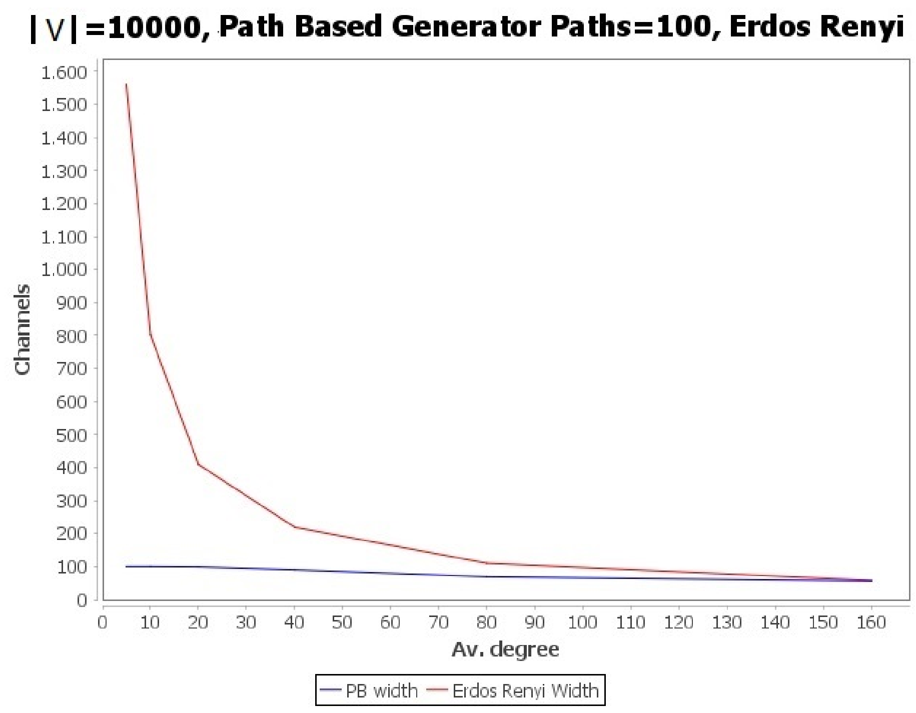


Figure 3.3: A comparison between Erdos-Renyi model and Path Based model.

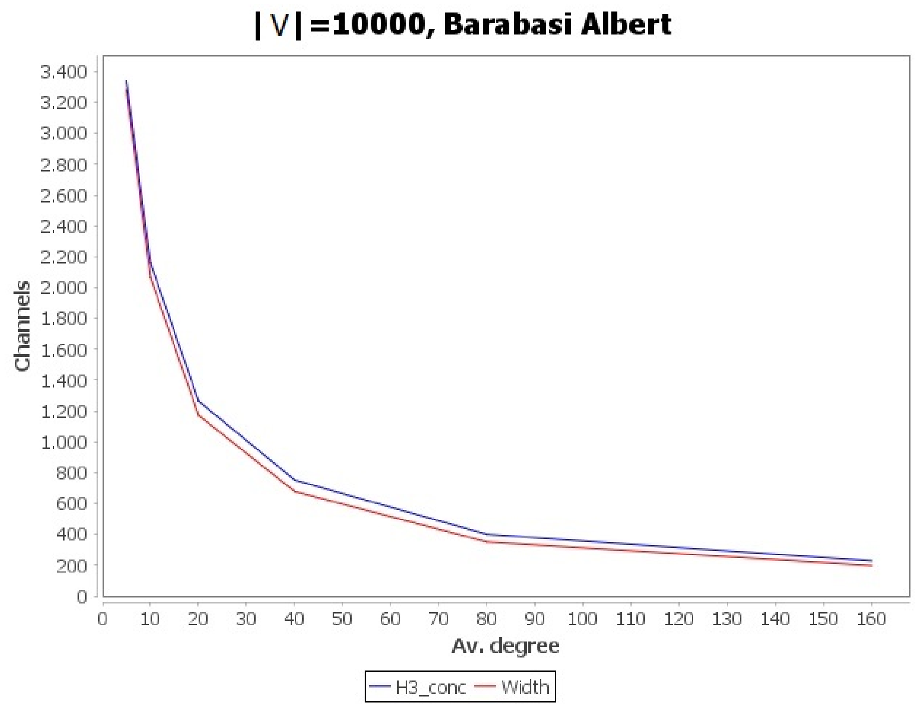


Figure 3.4: The efficiency of our chain decomposition algorithm in Barabasi Albert model.

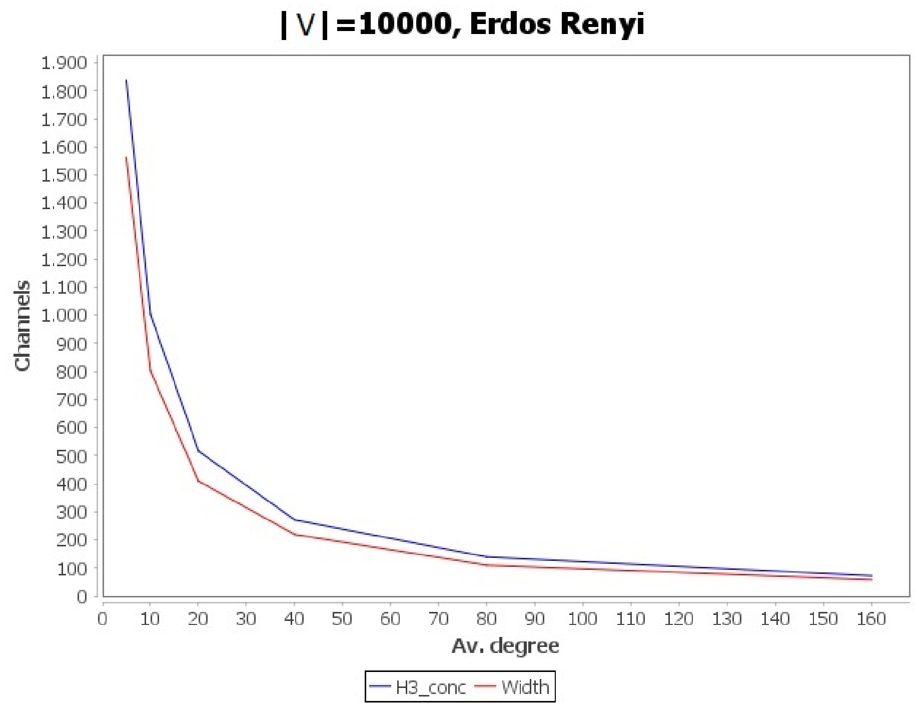


Figure 3.5: The efficiency of our chain decomposition algorithm in Erdos Renyi model.

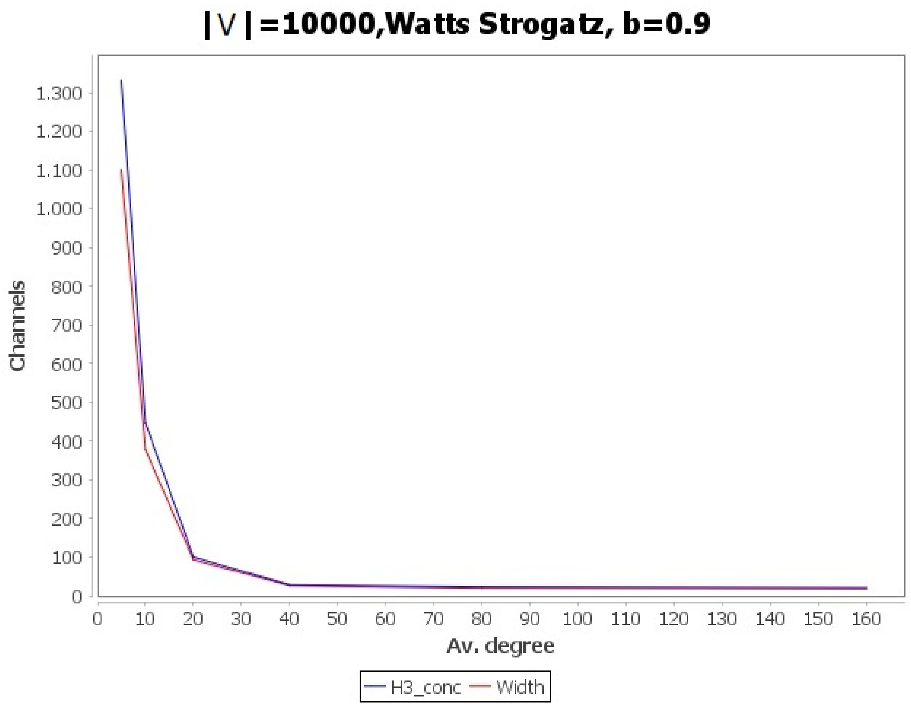


Figure 3.6: The efficiency of our chain decomposition algorithm in Watts-Strogatz model.

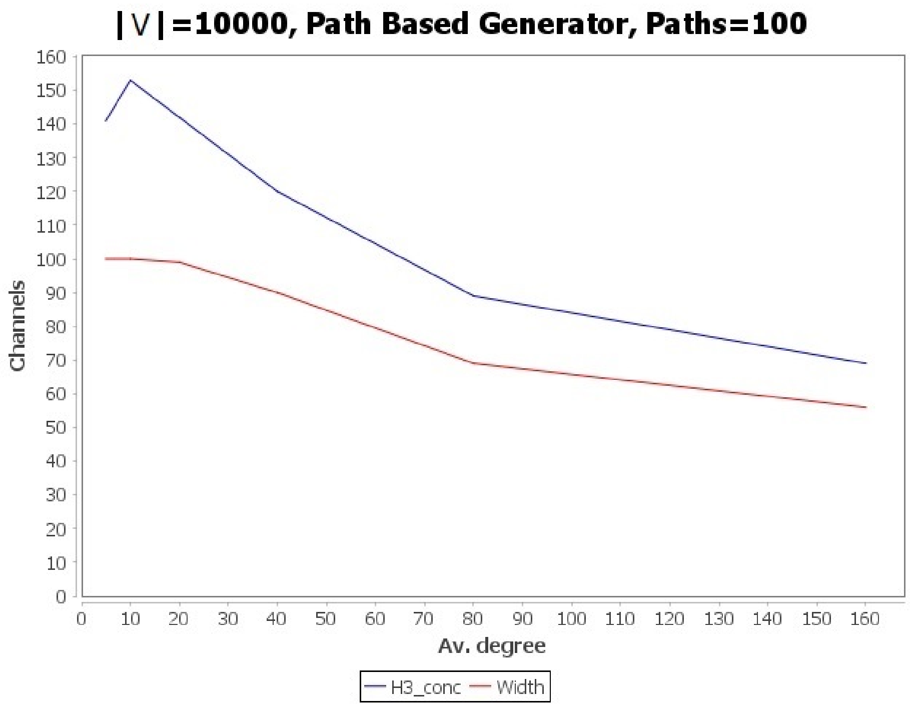


Figure 3.7: The efficiency of our chain decomposition algorithm in Path Based model.

3.4 Hierarchies and Transitivity

Lemma 3.4.1. *Given a chain decomposition D of a DAG $G = (V, E)$, each vertex $v_i \in V$, $0 \leq i < |V|$, can have at most one outgoing non-transitive edge per chain.*

Proof. Given a graph $G(V, E)$, a decomposition $D(C_1, C_2, ..., C_{k_c})$ of G, and a vertex $v \in V$, assume vertex v has two outgoing edges, (v, t_1) and (v, t_2), and both t_1 and t_2 are in chain C_i. The vertices are in ascending topological order in the chain by definition. Assume t_1 has a lower topological rank than t2. Thus, there is a path from t_1 to t_2, and accordingly a path from v to t_2 through t_1. Hence, the edge (v, t_2) is transitive. See Figure 3.8a. □

Lemma 3.4.2. *Given a chain decomposition D of a DAG $G = (V, E)$, each vertex $v_i \in V$, $0 \leq i < |V|$, can have at most one incoming non-transitive edge per chain.*

Proof. Similar to the proof of proposition 3.4.1. See figure 3.8b. □

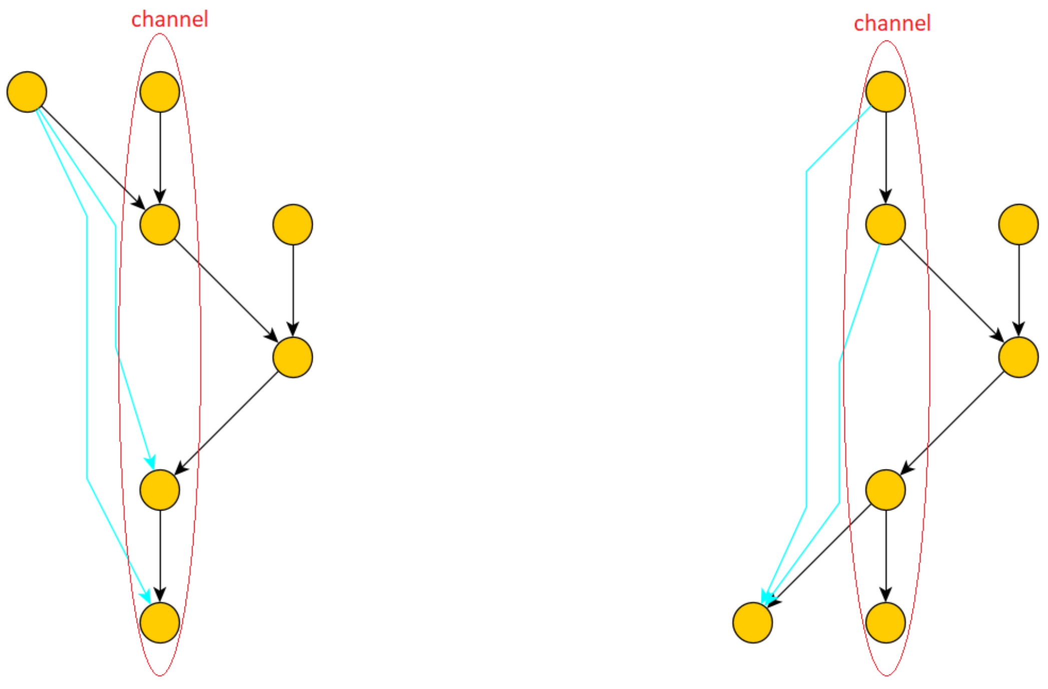


Figure 3.8: Example for the proof of Lemma 3.4.2. The blue edges are transitive. (a) shows the outgoing transitive edges that end to the same chain. (b) shows the incoming transitive edges that start from the same chain.

Theorem 3.4.3. *Let $G = (V, E)$ be a DAG with width w. The non-transitive edges of G are less than or equal to $width * |V|$, in other words $E_{red} = E - E_{tr} \leq width * |V|$.*

Proof. Given any DAG G and its width w, there is a chain decomposition of G with w number of chains. From Lemma 3.4.1, every vertex of G could have only one outgoing, non-transitive edge per chain, thus its non-transitive outgoing edges cannot be more than $w * |V|$. Notice that the same stands for the incoming edges, according to Lemma 3.4.2. □

According to Theorem 3.4.3, the time complexity of Algorithm 6 can be expressed as $O(k_c * |E_{red}|) = O(k_c * width * |V|)$ since $|E_{red}| \leq width * |V|$. Additionally, the chains rarely have the same length. Usually, the decomposition consists of a few long chains and several shorter chains. Hence, for most of the graphs it is not even possible $|E_{red}| = width * |V|$, $|E_{red}|$ it usually is much less than that. We present experimental results that confirm this in table 3.4 and 3.5.

Also, an essential application of Lemma 3.4.1 and 3.4.2 is that we can find a subset of E_{tr} in linear time. Given a chain decomposition or a path decomposition with k_c chains, we can trace the vertices and their outgoing edges and keep the edges that point to the lowest point of each chain, rejecting the rest as transitive. We do the same for the incoming edges keeping the edges that come from the highest point (vertex with highest topological rank) of each chain. This way, we find a subset $E'_{tr} \subseteq E_{tr}$. Hence, $|E - E'_{tr}| \leq k_c * |V|$. This linear time preprocessing can facilitate every transitive closure technique bounding the input graph edges, and the indegree and outdegree of every vertex by k_c. For example, algorithms based on tree cover, see [5, 15, 71, 75], are practical on sparse graphs and can be enhanced further with a preprocessing step that removes transitive edges.

3.5 Indexing Scheme

In this section, we present an important application of our chain decomposition technique. We solve the transitive closure problem by creating an indexing scheme that is based on chain decomposition.

Jagadish described a compressed transitive closure technique in 1990 [44] applying the indexing scheme and path/chain decomposition. As we discussed, Jagadish's heuristic for chain decomposition runs in $O(n^2)$ using the pre-computed transitive closure. Our technique outperforms that. It runs in almost linear time without using a pre-computed transitive closure, and the outcome is close to the optimal. Furthermore, his method focuses on compression and does not answer queries in constant time.

Simon, see [66], describes that indexing scheme too. He calculates a path decomposition, boosting the method presented in [38]. The linear time heuristic he presented is Chain Order Heuristic. In the following sections, we show that using our channel decomposition technique outperforms finding the indexing scheme using merely a path decomposition.

We build our solution in $O(k_c * |E_{red}|)$ time, where using our solution, we can answer queries in constant time. k_c is the number of chains and $|E_{red}|$ is the number of non-transitive edges. Additionally, we will show that $|E_{red}| \leq width * |V|$. The space complexity of our algorithm is $O(k_c * |V|)$. Furthermore, we present extensive experimental work, and we show both in theory and practice the efficiency of our approach.

By finding the strongly connected components, we can make any directed graph acyclic. All vertices of a SCC will form a supernode since any vertex is reachable from any other vertex in the same component. This is a well-known step, so we assume that the input of our method is a DAG. The steps given a DAG are:

1. Perform Chain decomposition
2. Sort Adjacency lists
3. Create Indexing Scheme

In step 1, we use our chain decomposition technique that runs in $O(|E| + c * l)$. In step 2, we sort the adjacency lists in $O(|V| + |E|)$ time. Finally, we create the indexing scheme in $O(k_c * |E_{red}|)$ time and $O(k_c * |V|)$ space. If we had done merely path decomposition, the time complexity would be $O(k_p * |E_{red}|)$ and $O(k_p * |V|)$ space. Probably, you have already noticed the relation between step 1 and step 3. The fewer chains the first step gives, the more efficient becomes the third.

3.5.1 The Indexing Scheme

Assume there is a chain decomposition of a DAG G with size k_c. Its indexing scheme includes a pair and an array of indexes of size k_c for every vertex. See for example Figure 3.9. The first integer of the pair indicates the node's chain and the second its position in the chain. For example, vertex 1 of Figure 3.9 has $(1, 1)$. The node belongs to the $1st$ chain, and it is the $1st$ element in it. Given a chain decomposition, we can easily construct the pairs in $O(|V|)$ time with a traversal of the chains. Every cell of the k_c size array represents a chain. The i-th cell represents the i-th chain. The entry in the i-th cell corresponds to the lowest point of the i-th chain the vertex can reach. For example, the array of vertex 1 is $[1, 3, 2]$. The first cell of the array indicates that vertex 1 can reach the $1st$ vertex of the first chain (can reach itself, reflexive property). The second cell of the array indicates that vertex 1 can reach the $3nd$ vertex of the second chain (There is a path from vertex 1 to vertex 7). Finally, the third cell of the array indicates that vertex 1 can reach the $2rd$ vertex of the third chain.

Notice that we do not need the second integer of any pair. If we know the chain a vertex belongs in, we can conclude its position using the array. We present it like that to make it easier to understand.

The process of answering a reachability query is simple. Assume, there is a vertex s and a target vertex t. To find if the vertex t is reachable from the s, we get t's chain, and we use it as an index in s's array. Hence, we know the lowest point of t's chain vertex s can reach. s can reach t if that point is less than or equal to t's position, else it cannot.

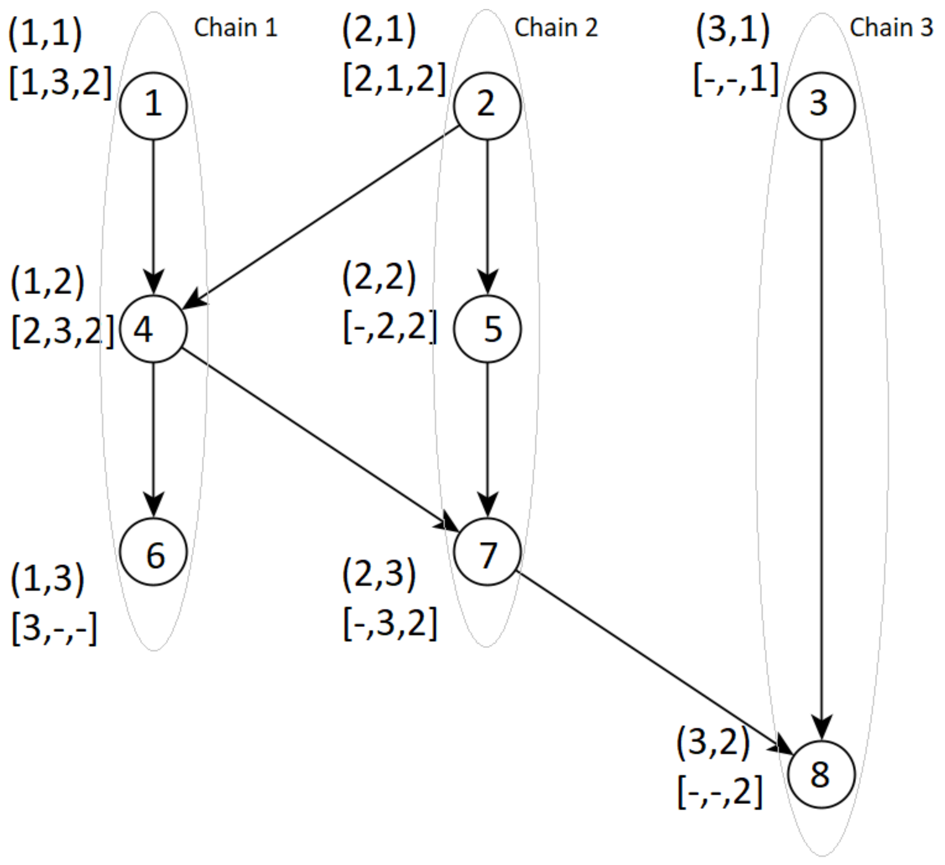


Figure 3.9: An example of an indexing scheme.

3.5.2 Sorting Adjacency lists

Algorithm 7 sorts the adjacency list of every vertex. More precisely, it sorts the adjacency lists of immediate successors in ascending topological order in linear time. The variable stack indicates the sorted adjacency list. The algorithm traverses the vertices in reverse topological order $(v_n, ..., v_1)$. For every vertex v_i, $1 \leq i \leq n$, it pushes v_i in the stacks of all immediate predecessors. This step could be performed even before the chain decomposition as a preprocessing step. We present it in this section to emphasize its crucial role in the indexing scheme creation. If the adjacency list is not sorted the time complexity of the algorithm would be $O(k_c * |E|)$ instead of $O(k_c * |E_{red}|)$.

Algorithm 7 Sorting Adjacency lists

```
procedure SORT($G, t$)
INPUT: A DAG $G = (V, E)$ and a topological sorting t of G
    for each vertex: $v_i \in G$ do
        $v_i$.stack ← new stack()
    end for
    for each vertex $v_i$ in reverse topological order do
        for every incoming edge $e(s_j, v_i)$ do
            $s_j$.stack.push($v_i$)
        end for
    end for
end procedure
```

Lemma 3.5.1. *Algorithm 7 sorts the adjacency lists of immediate successors in ascending topological order.*

Proof of Lemma 3.5.1. Assume that there is a stack $(u_1, ..., u_n)$, u_1 is the top of the stack. Assume that there is a pair (u_j, u_k) in the stack, where u_j has a bigger topological rank than u_k and u_j precedes u_k. That means that the for-loop examined u_j before u_k since it goes through the vertices in reverse topological order. This is a contradiction. The vertex u_j cannot precede u_k if it was examined first by the for-loop. $\square$

3.5.3 Creating the Indexing Scheme.

Algorithm 8 constructs the indexing scheme. The first for-loop initializes the array of indexes. For every vertex, it initializes the cell that corresponds to its chain. The rest of the cells are initialized to infinite. The indexing scheme initialization is illustrated in figure 3.10. The dashes represent the infinite. Notice that after the initialization, the indexes of all sink vertices have been calculated. Since a sink has no successors, the only vertex it can reach is itself.

The second for-loop builds the indexing scheme. It goes through vertices in descending topological order. For each vertex, it visits its immediate successors

Algorithm 8 Indexing Scheme

```
1: procedure CREATE INDEXING SCHEME($G, T, D$)
   INPUT: A DAG $G = (V, E)$, a topological sorting T of G, and the decomposition D of G.
2:     for each vertex: $v_i \in G$ do
3:         $v_i$.indexes $\leftarrow$ new table[size of D]
4:         $v_i$.indexes.fill($\infty$)
5:         $ch\_no \leftarrow v_i$'s chain index
6:         $pos \leftarrow v_i$'s chain position
7:         $v_i$.indexes[ $ch\_no$ ] $\leftarrow pos$
8:     end for
9:     for each vertex $v_i$ in reverse topological order do
10:        while $v_i$.stack $\neq \emptyset$ do
11:            $target \leftarrow v_j$.stack.pop()
12:            $t\_ch \leftarrow target$'s chain index
13:            $t\_pos \leftarrow target$'s chain position
14:            if $t\_pos < v_i$.indexes[$t\_ch$] then // $(v_i, target)$ is not transitive
15:                $v_i$.updateIndexes($target.indexes$)
16:            end if
17:        end while
18:    end for
19: end procedure
```

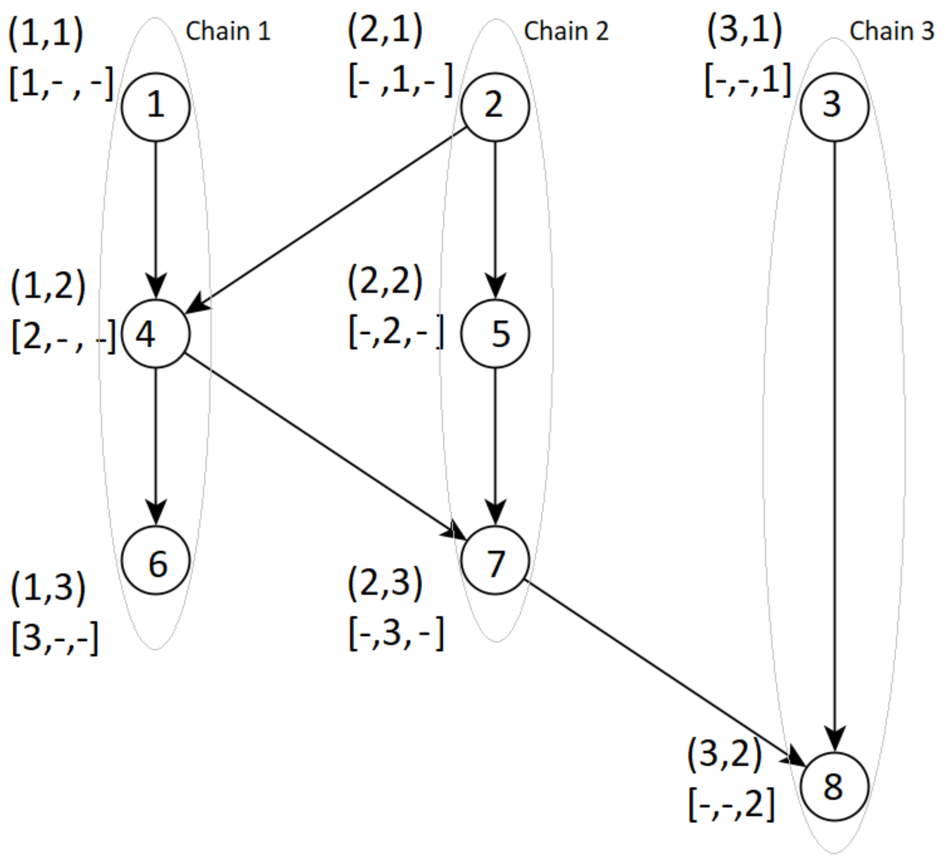


Figure 3.10: Initialization of indexes.

(outgoing edges) in ascending topological order and updates the indexes. Suppose we have the edge (v, s), and we have calculated the indexes of vertex s (s is immediate successor of v). The process of updating the indexes of v with its immediate successor s means that s will pass all its information to the vertex v. Hence, vertex v will be aware that it can reach s and all its successors. Assume the array of indexes of v is $[a_1, a_2, ..., a_{k_c}]$ and the array of s is $[b_1, b_2, ..., b_{k_c}]$. To update the indexes of v using s, we merely trace the arrays and keep the smallest values. For every pair of indexes (a_i, b_i), $0 \leq i < kc$, the new value of a_i will be $\min\{a_i\ , b_i\}$. This process needs k_c steps.

Lemma 3.5.2. *Given a vertex v and the calculated indexes of its successors, the while-loop of algorithm 8 (lines 10-17) calculates the indexes of v by updating its array with its non-transitive outgoing edges' successors.*

Proof. Updating the indexes of vertex v with all its immediate successors will make v aware of all its descendants. The while-loop of Algorithm 8 does not perform the update function for every direct successor. It skips all the transitive edges. Assume there is such a descendant t and the transitive edge (v, t). Since the edge is transitive, we know by definition that there exists a path from v to t with a length of more than 1. Suppose that the path is $(v, v_1, .., t)$. with a traversal of the chains. Vertex v_1 is a predecessor of t and immediate successor of v. Hence it has a lower topological rank than t. Since, while-loop examines the incident vertices in ascending topological order, then vertex t will be visited after vertex v_1. The opposite leads to a contradiction. Consequently, for every incident transitive edge of v, the loop firstly visits a vertex v_1 which is a predecessor of t. Thus vertex v will be firstly updated by v_1 and it will record the edge (v, t) as transitive. There is no reason to update vertex v indexes with those of vertex t since the indexes of t will be greater or equal. □

Theorem 3.5.3. *Let $G = (V, E)$ be a DAG. Algorithm 8 computes the indexing scheme in $O(k_c * |E_{red}|)$ time.*

Proof of Theorem 3.5.3. In the initialization step, the indexes of all sink vertices have been computed as we described above. Taking vertices in reverse topological order, the first vertex we meet is a sink vertex. When the for-loop of line 9 visits the first non-sink vertex, the indexes of its successors are computed (all its successors are sink vertices). According to Lemma 1, we can calculate its indexes, ignoring the transitive edges. Assume the for-loop has reached the vertex v_i in the $i - th$ iteration, and the indexes of its successors are calculated. Similarly, from Lemma 1, we can calculate its indexes. By induction, we can calculate the indices of all vertices, ignoring all transitive edges in $O(|E_{red}| * k_c)$ time. □

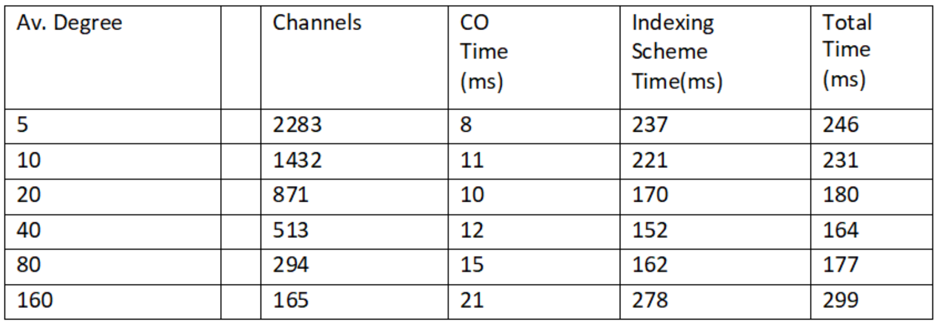
3.5.4 Experiments

We used the same graphs of 5000 and 10000 nodes as we described in Section 3.3.3 produced by three different models of the Networkx. We performed chain decomposition using our approach (Alg. 6, H3_conc), and created the indexing scheme using Algorithm 8). Assume the sorting of the adjacency list is a preprocessing step (Alg. 7) and the input graph has sorted adjacency lists. We recorded our results in Tables 3.4 and 3.5. Table 3.4 holds the results of graphs with 5000 nodes, and Table 3.5 the results of graphs with 10000 nodes. Next, we explain the columns of the tables.

- **Av. Degree**: The average degree of the graph
- **Chains**: Number of chains computed by our heuristic (H3_conc).
- $|\mathbf{E_{tr}}|$: Number of transitive edges.
- $|\mathbf{E_{red}}|$: Number of non-transitive edges.
- $|\mathbf{E_{tr}}|/|\mathbf{E}|$: The percentage of transitive edges.
- **H3_conc Time (ms)**: The time, in milliseconds, of the chain decomposition step.
- **Indexing Scheme Time (ms)**: The time, in milliseconds, of the indexing scheme creation step.
- **Total**: The total time(ms) needed to decompose the graph and create the indexing scheme. It is the sum of the two preceding cells.
- **TC**: The time needed by a known algorithm for transitive closure with time complexity $O(|V| * |E|)$. The algorithm performs a DFS function for every vertex to mark reachable vertices. It stores the results in a 2-D adjacency matrix.

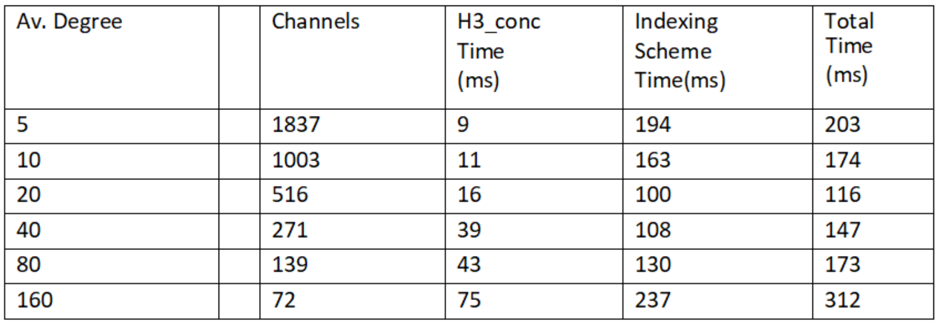
The phase of indexing scheme creation needs $k_c * |E_{red}| + |E_{tr}|$ steps. The numbers on the tables are interesting. As the average degree increases and the graph becomes denser, the cardinality of E_{red} remains almost stable, and the chains decrease. Of course, since the E_{red} does not vary as the average degree increases, the cardinality of E_{tr} increases ($E_{tr} = E - E_{red}$). The algorithm merely traces in linear time the transitive edges. Consequently, the growth of E_{tr} does not affect the run time considerably. As a result, the run time of our technique does not increase as the input graph increases. To demonstrate it clearly, we drew the line chart of figure 3.11 for the graphs of 10000 nodes produced by the Erdos-Renyi model. The blue line represents the run time of the indexing scheme, and the red line the run time of the algorithm based on DFS (TC). The time of the algorithm based on DFS increases as the average degree increases, while the time of the indexing scheme is a straight line parallel to the x-axis. All models follow this pattern. See Tables 3.4 and 3.5.

We decompose the graph into chains with our algorithm since it is the most efficient. A chain decomposition is preferable to a path decomposition if we create



Av. Degree		Channels	CO Time (ms)	Indexing Scheme Time(ms)	Total Time (ms)
5		2283	8	237	246
10		1432	11	221	231
20		871	10	170	180
40		513	12	152	164
80		294	15	162	177
160		165	21	278	299

(a) Metrics: Creating the indexing scheme in combination with the chain order heuristic.



Av. Degree		Channels	H3_conc Time (ms)	Indexing Scheme Time(ms)	Total Time (ms)
5		1837	9	194	203
10		1003	11	163	174
20		516	16	100	116
40		271	39	108	147
80		139	43	130	173
160		72	75	237	312

(b) Metrics: Creating the indexing scheme in combination with algorithm 6 for chain decomposition.

Table 3.3: The tables present the run time of indexing scheme using path and chain decomposition.

the indexing scheme. Assume that we have a path decomposition, and we perform chain concatenation. If there is no concatenation between two paths, the concatenation algorithm will run in linear time, which is an acceptable cost. On the other hand, if there are concatenations, for each one of them, the cost is $O(l)$ time units but the gain in the following step of scheme creation is $|V|$ units of space and $|E_{red}|$ units of time. That stands because every concatenation reduces the indexes we need for every vertex by one. Hence, applying path concatenation, we create faster a more compact indexing scheme.

Tables 3.3a and 3.3b include metrics of creating the indexing scheme using different decomposition techniques on Erdos Reyni graphs of 10000 nodes. In table 3.3a, we have created the indexing scheme using the chain order heuristic(path decomposition), while in table 3.3b, we use our chain decomposition algorithm.

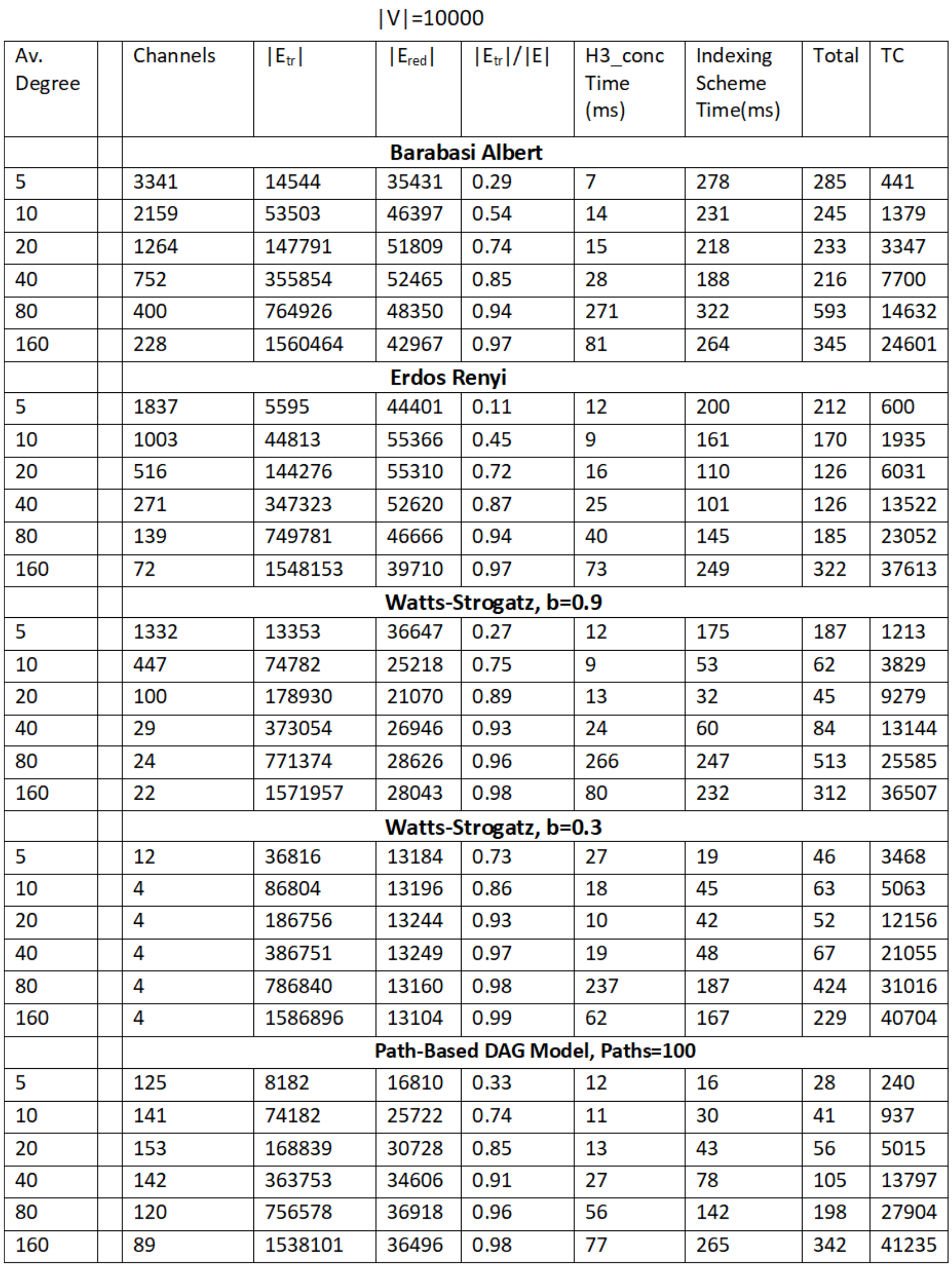
3.6 Conclusions

In this work, we present heuristics that find a chain decomposition in almost linear time and such that the number of chains can be very close to the minimum. Our

|V|=5000

Av. Degree		Channels	$\|E_{tr}\|$	$\|E_{red}\|$	$\|E_{tr}\|/\|E\|$	H3_conc Time (ms)	Indexing Scheme Time(ms)	Total	TC
		Barabasi Albert							
5		1630	8054	18921	0.32	3	101	104	137
10		1055	28230	21670	0.57	12	79	91	333
20		664	75801	23799	0.76	6	54	60	638
40		355	180815	22504	0.89	10	48	58	1418
80		207	382422	20854	0.95	122	118	240	3018
160		163	770771	17660	0.98	25	107	132	5464
		Erdos Renyi							
5		923	3440	21466	0.138	6	67	73	172
10		492	24761	25425	0.49	10	51	61	487
20		252	75312	24646	0.75	5	26	31	1079
40		139	175809	22634	0.89	46	51	97	2896
80		70	378015	19435	0.95	16	50	66	5260
160		38	769919	16843	0.98	98	138	236	8609
		Watts-Strogatz, b=0.9							
5		687	7742	17258	0.30	13	71	84	393
10		212	37992	12008	0.75984	11	18	29	817
20		60	89272	10728	0.89	23	22	45	1530
40		25	186486	13514	0.93	47	45	92	3704
80		20	386294	13706	0.97	115	103	218	6172
160		17	787066	12934	0.98	253	207	406	9173
		Watts-Strogatz, b=0.3							
5		9	18421	6579	0.74	11	8	19	910
10		4	43505	6495	0.87	8	11	19	1107
20		4	93490	6510	0.93	18	18	36	2176
40		5	193416	6584	0.97	17	18	35	4753
80		4	393348	6652	0.98	98	82	180	7949
160		5	793430	6570	0.99	250	166	416	11757
		Path-Based DAG Model, Paths=70							
5		86	14155	10809	0.57	8	7	15	206
10		101	36801	13102	0.74	7	12	19	313
20		107	84168	15419	0.85	7	15	22	890
40		93	181388	16988	0.91	49	216	265	2584
80		73	376220	17303	0.96	128	163	291	4603
160		51	758207	16566	0.98	55	141	196	9358

Table 3.4: Indexing scheme analysis on graphs of 5000 nodes.



|V|=10000

Av. Degree		Channels	$\|E_{tr}\|$	$\|E_{red}\|$	$\|E_{tr}\|/\|E\|$	H3_conc Time (ms)	Indexing Scheme Time(ms)	Total	TC
		Barabasi Albert							
5		3341	14544	35431	0.29	7	278	285	441
10		2159	53503	46397	0.54	14	231	245	1379
20		1264	147791	51809	0.74	15	218	233	3347
40		752	355854	52465	0.85	28	188	216	7700
80		400	764926	48350	0.94	271	322	593	14632
160		228	1560464	42967	0.97	81	264	345	24601
		Erdos Renyi							
5		1837	5595	44401	0.11	12	200	212	600
10		1003	44813	55366	0.45	9	161	170	1935
20		516	144276	55310	0.72	16	110	126	6031
40		271	347323	52620	0.87	25	101	126	13522
80		139	749781	46666	0.94	40	145	185	23052
160		72	1548153	39710	0.97	73	249	322	37613
		Watts-Strogatz, b=0.9							
5		1332	13353	36647	0.27	12	175	187	1213
10		447	74782	25218	0.75	9	53	62	3829
20		100	178930	21070	0.89	13	32	45	9279
40		29	373054	26946	0.93	24	60	84	13144
80		24	771374	28626	0.96	266	247	513	25585
160		22	1571957	28043	0.98	80	232	312	36507
		Watts-Strogatz, b=0.3							
5		12	36816	13184	0.73	27	19	46	3468
10		4	86804	13196	0.86	18	45	63	5063
20		4	186756	13244	0.93	10	42	52	12156
40		4	386751	13249	0.97	19	48	67	21055
80		4	786840	13160	0.98	237	187	424	31016
160		4	1586896	13104	0.99	62	167	229	40704
		Path-Based DAG Model, Paths=100							
5		125	8182	16810	0.33	12	16	28	240
10		141	74182	25722	0.74	11	30	41	937
20		153	168839	30728	0.85	13	43	56	5015
40		142	363753	34606	0.91	27	78	105	13797
80		120	756578	36918	0.96	56	142	198	27904
160		89	1538101	36496	0.98	77	265	342	41235

Table 3.5: Indexing scheme analysis on graphs of 10000 nodes.

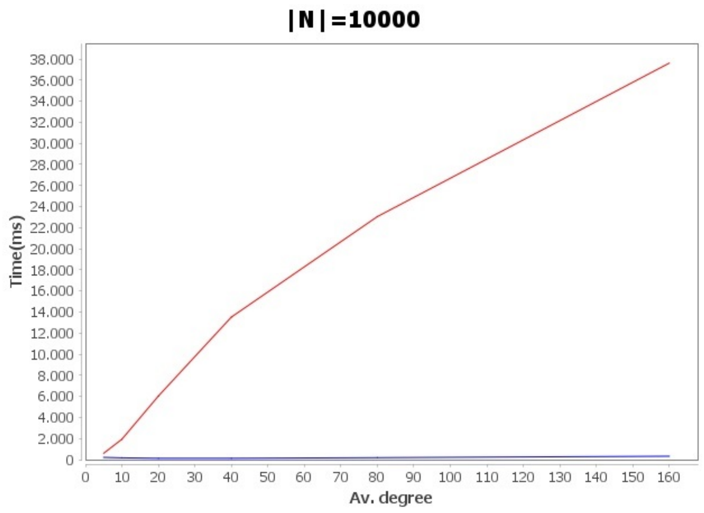


Figure 3.11: Run time comparison between the Indexing Scheme (blue line) and TC (red line) for Erdos-Renyi model on graphs of 10000 nodes. See table 3.5.

experiments expose the behavior of the width as the density grows, along with the efficiency of our heuristics. We bound the set E_{red} by $width * |V|$ and illustrate how to find a subset of E_{tr} in linear time given a path/chain decomposition. Our approach and theory have applications in many areas. We applied them to the problem of transitive closure. We built in $O(width * k_c * |V|)$ time and $O(k_c * |V|)$ space an indexing scheme that allows us to answer reachability queries in constant time. The time complexity is $O(k_c * |E_{red}|)$, and the space complexity is $O(k_c * |V|)$. Additionally, our experimental work reveals the practical efficiency of this approach, especially for very large, and medium to dense graphs.

Bibliography

[1] Jfree.

[2] Tom Sawyer Software.

[3] yWorks.

[4] R. Agrawal. Alpha: an extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, 1988.

[5] Rakesh Agrawal, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD Record*, 18(2):253–262, 1989.

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

[7] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of usability studies*, 4(3):114–123, 2009.

[8] Michael J. Bannister, David A. Brown, and David Eppstein. Confluent orthogonal drawings of syntax diagrams. In Emilio Di Giacomo and Anna Lubiw, editors, *Graph Drawing and Network Visualization - 23rd International Symposium, GD 2015, Los Angeles, CA, USA, September 24-26, 2015, Revised Selected Papers*, Lecture Notes in Computer Science, pages 260–271, 2015.

[9] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[10] Paola Bonizzoni. A linear-time algorithm for the perfect phylogeny haplotype problem. *Algorithmica*, 48(3):267–285, 2007.

[11] Nicolas Boria, Gianpiero Cabodi, Paolo Camurati, Marco Palena, Paolo Pasini, and Stefano Quer. A greedy approach to answer reachability queries on dags. *arXiv preprint arXiv:1611.02506*, 2016.

[12] Ulrik Brandes and Boris Köpf. Fast and simple horizontal coordinate assignment. In *Graph Drawing, 9th International Symposium, GD 2001 Vienna, Austria, September 23-26, 2001, Revised Papers*, pages 31–44, 2001.

[13] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. A fast layout algorithm for k-level graphs. In *Graph Drawing, 8th International Symposium, GD 2000, Colonial Williamsburg, VA, USA, September 20-23, 2000, Proceedings*, pages 229–240, 2000.

[14] Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I Tomescu. A linear-time parameterized algorithm for computing the width of a dag. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 257–269. Springer, 2021.

[15] Li Chen, Amarnath Gupta, and M Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31st international conference on Very large data bases*, pages 493–504. Citeseer, 2005.

[16] Yangjun Chen and Yibin Chen. On the dag decomposition. *British Journal of Mathematics and Computer Science*, 2014. 10(6): 1-27, 2015, Article no.BJMCS.19380, ISSN: 2231-0851.

[17] Yangjun Chen and Yibin Chen. On the graph decomposition. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, pages 777–784. IEEE, 2014.

[18] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (OGDF). In *Handbook on Graph Drawing and Visualization.*, pages 543–569. 2013.

[19] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice-Hall, 1999.

[20] Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Armando Parise, Roberto Tamassia, Emanuele Tassinari, Francesco Vargiu, and Luca Vismara. Drawing directed acyclic graphs: An experimental study. In Stephen C. North, editor, *Graph Drawing, Symposium on Graph Drawing, GD '96, Berkeley, California, USA, September 18-20, Proceedings*, volume 1190 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 1996.

[21] Giuseppe Di Battista, E Pietrosanti, Roberto Tamassia, and Ioannis G Tollis. Automatic layout of pert diagrams with x-pert. In *[Proceedings] 1989 IEEE Workshop on Visual Languages*, pages 171–176. IEEE, 1989.

[22] Emilio Di Giacomo, Walter Didimo, Giuseppe Liotta, Fabrizio Montecchiani, and Ioannis G Tollis. Exploring complex drawings via edge stratification. In *International Symposium on Graph Drawing*, pages 304–315. Springer, 2013.

[23] R. P. DILWORTH. A decomposition theorem for partially ordered sets. *Ann. Math.*, 52:161–166, 1950.

[24] Fulkerson DR. Note on dilworth's embedding theorem for partially ordered sets. *Proc. Amer. Math. Soc.*, 52(7):701–702, 1956.

[25] Peter Eades and Nicholas C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.

[26] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of Sugiyama's algorithm for layered graph drawing. In János Pach, editor, *Graph Drawing*, pages 155–166, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[27] P Erdős. Rényi, a.:" on random graphs. *I". Publicationes Mathematicae (Debre*, 1959.

[28] Donald L Fisher and William M Goldstein. Stochastic pert networks as models of cognition: Derivation of the mean, variance, and distribution of reaction time using order-of-processing (op) diagrams. 1983.

[29] Michael Fröhlich and Mattias Werner. Demonstration of the interactive graph-visualization system *da Vinci*. In *Graph Drawing, DIMACS International Workshop, GD '94, Princeton, New Jersey, USA, October 10-12, 1994, Proceedings*, pages 266–269, 1994.

[30] Delbert Ray Fulkerson. Note on dilworth's decomposition theorem for partially ordered sets. In *Proc. Amer. Math. Soc*, volume 7, pages 701–702, 1956.

[31] Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and K-P Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[32] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Trans. Software Eng.*, 19(3):214–230, 1993.

[33] Emden R. Gansner, Eleftherios E. Koutsofios, and Stephen C. North. Drawing graphs with dot. 2015.

[34] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw., Pract. Exper.*, 30(11):1203–1233, 2000.

[35] Mohammad Ghoniem, J-D Fekete, and Philippe Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In *IEEE symposium on information visualization*, pages 17–24. Ieee, 2004.

[36] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design and Applications*. Wiley Publishing, 1st edition, 2014.

[37] Michael T Goodrich and Roberto Tamassia. *Algorithm design and applications*. Wiley Hoboken, 2015.

[38] Alla Goralčíková and Václav Koubek. A reduct-and-closure algorithm for graphs. In *International Symposium on Mathematical Foundations of Computer Science*, pages 301–307. Springer, 1979.

[39] Jens Gramm, Till Nierhoff, Roded Sharan, and Till Tantau. Haplotyping with missing data via perfect path phylogenies. *Discrete Applied Mathematics*, 155(6-7):788–805, 2007.

[40] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[41] Michael Himsolt. Graphlet: design and implementation of a graph editor. *Softw., Pract. Exper.*, 30(11):1303–1324, 2000.

[42] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[43] Selma Ikiz and Vijay K Garg. Efficient incremental optimal chain partition of distributed program traces. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, pages 18–18. IEEE, 2006.

[44] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, December 1990.

[45] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[46] Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Xu Yu. SCARAB: scaling reachability computation on large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 169–180, 2012.

[47] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 595–608, 2008.

[48] Michael Jünger, Petra Mutzel, and Christiane Spisla. A flow formulation for horizontal coordinate assignment with prescribed width. In *Graph Drawing and Network Visualization - 26th International Symposium, GD 2018, Barcelona, Spain, September 26-28, 2018, Proceedings*, pages 187–199, 2018.

[49] Michael Kaufmann and Dorothea Wagner. Drawing graphs: Methods and models. *LNCS vol. 2025*, 2001.

[50] Evgenios M. Kornaropoulos and Ioannis G. Tollis. Algorithms and bounds for overloaded orthogonal drawings. *Journal of Graph Algorithms and Applications*, 20(2):217–246, 2016.

[51] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[52] Anna Kuosmanen, Topi Paavilainen, Travis Gagie, Rayan Chikhi, Alexandru I. Tomescu, and Veli Mäkinen. Using minimum path cover to boost dynamic programming on dags: Co-linear chaining extended. In *Research in Computational Molecular Biology - 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings*, pages 105–121, 2018.

[53] Lei Li, Wen Hua, and Xiaofang Zhou. HD-GDD: high dimensional graph dominance drawing approach for reachability query. *World Wide Web*, 20(4):677–696, 2017.

[54] Panagiotis Lionakis, Giacomo Ortali, and Ioannis Tollis. Adventures in abstraction: Reachability in hierarchical drawings. In *Graph Drawing and Network Visualization: 27th International Symposium, GD 2019, Prague, Czech Republic, September 17–20, 2019, Proceedings*, pages 593–595, 2019.

[55] Panagiotis Lionakis, Giacomo Ortali, and Ioannis G Tollis. Constant-time reachability in dags using multidimensional dominance drawings. *SN Computer Science*, 2(4):1–14, 2021.

[56] Nikola S. Nikolov and Patrick Healy. *Hierarchical Drawing Algorithms, in Handbook of Graph Drawing and Visualization, ed. Roberto Tamassia.* CRC Press, 2014. pp. 409-453.

[57] James B. Orlin. Max flows in O(nm) time, or better. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 765–774, 2013.

[58] Giacomo Ortali and Ioannis G Tollis. Algorithms and bounds for drawing directed graphs. In *International Symposium on Graph Drawing and Network Visualization*, pages 579–592. Springer, 2018.

[59] Giacomo Ortali and Ioannis G. Tollis. A new framework for hierarchical drawings. *Journal of Graph Algorithms and Applications*, 23(3):553–578, 2019.

[60] Frances Newbery Paulisch and Walter F. Tichy. EDGE: an extendible graph editor. *Softw., Pract. Exper.*, 20(S1):S1, 1990.

[61] Micha A Perles. A proof of dilworth's decomposition theorem for partially ordered sets. *Israel Journal of Mathematics*, 1(2):105–107, 1963.

[62] Sergey Pupyrev, Lev Nachmanson, and Michael Kaufmann. Improving layered graph layouts with edge bundling. In Ulrik Brandes and Sabine Cornelsen, editors, *Graph Drawing - 18th International Symposium, GD 2010, Konstanz, Germany, September 21-24, 2010. Revised Selected Papers*, Lecture Notes in Computer Science, pages 329–340, 2010.

[63] Helen C Purchase, John Hamer, Martin Nöllenburg, and Stephen G Kobourov. On the usability of lombardi graph drawings. In *International symposium on graph drawing*, pages 451–462. Springer, 2012.

[64] Georg Sander. Layout of compound directed graphs. Technical report, Universität des Saarlandes, 1996.

[65] Claus-Peter Schnorr. An algorithm for transitive closure with linear expected time. *SIAM J. Comput.*, 7(2):127–133, 1978.

[66] K. SIMON. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58(1-3):325–346, 1988.

[67] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[68] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[69] Robert Tarjan. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121, 1971.

[70] Alexander I Tomlinson and Vijay K Garg. Monitoring functions on global states of distributed programs. *Journal of Parallel and Distributed Computing*, 41(2):173–189, 1997.

[71] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 845–856, 2007.

[72] Edward R Tufte. *The visual display of quantitative information*, volume 2. Graphics press Cheshire, CT, 2001.

[73] Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 913–924, 2011.

[74] Renê Rodrigues Veloso, Loïc Cerf, Wagner Meira Jr., and Mohammed J. Zaki. Reachability queries in very large graphs: A fast refined online search approach. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 511–522, 2014.

[75] Haixun Wang, Hao He, Jun Yang, Philip S Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 75–75. IEEE, 2006.

[76] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *nature*, 393(6684):440–442, 1998.

[77] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *VLDB J.*, 21(4):509–534, 2012.