On Graph Hierarchies
Chain Decomposition and Applications

Giorgos Kritikakis
georgecretek@gmail.com

November 2021

Abstract

In this paper, 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 a known 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. Furthermore, we show that E_{red} is bounded, $E_{red} <= width * |V|$, and we illustrate how to find a subset of E_{tr} (the set of transitive edges) without calculating transitive closure.

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.

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 2, we present path decomposition approaches, and in section 3, chain decomposition and path concatenation. Additionally, we show experiments and evaluate the performance of our heuristic. Next, we examine a few outcomes. In section

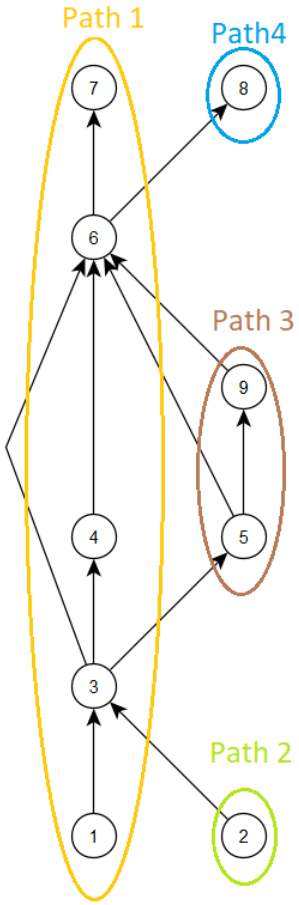
4, we prove that $E_{red} \leq width * |V|$, and see how we can in linear time, remove a subset of transitive edges and bound $E - E'_{tr}$ by $O(k * V)$ given a path/chain decomposition of size k. Finally, section 5 demonstrates how to build a known indexing scheme for transitive closure reporting experimental results.

In our research, we extensively used the Path-Based Framework. A new general-purpose hierarchical graph drawing framework that reveals critical aspects of graph hierarchies [15, 16] .

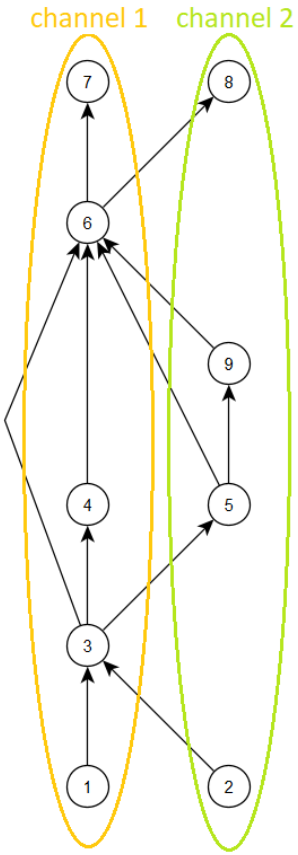
We conducted all the experiments using a laptop PC (Intel(R) Core(TM) i5-6200U CPU, 8 GB of main memory).

Definitions and Abbreviations

- **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. The vertices of a path/chain are in ascending topological order.
- **DAG:** Directed acyclic graph.
- **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 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 [6].
  - The number of chains in a minimal chain decomposition of a graph is equal to its width.
- **Transitive edge:** A 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 represent 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 [3] of G(V,E). The transitive reduction is unique for DAGs. It contains the minimum number of edges needed to form the same transitive relation with G(V, E).
- **Sink vertex**: A vertex with no outgoing edges.
- **Source vertex**: A vertex with no incoming edges.



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



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

Figure 1: On the left, there is a path decomposition of graph G. On the right, a chain decomposition of G.

2 Path Decomposition

Jagadish in [13] categorized path decomposition techniques into two categories. Chain Order Heuristics and Node Order Heuristics. The first construct the paths one by one, while the second creates the paths in parallel. He also described the benefits of topological sorting. More precisely, in [1], the author 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 precomputation 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 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 chain to the extent possible. The chain ends when no more unused immediate successors can be found. As you can see in Algorithm 1, the first for loop finds an unused vertex and creates a chain. The inner while loop extends the chain.

Algorithm 1 Path Decomposition

```
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 ← ∅                                                    ▷ Set of paths
    Mark all nodes unused
    for every unused vertex v_i ∈ T in ascending topological order do
        current ← v_i
        C ← new Chain()
        Add current to C
        while there is an unused immediate successor s of the current node do
            add s to C
            current ← 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 chain. If there is no matching, then a new chain is created for the vertex. Algorithm 2 illustrates the node order heuristic.

Algorithm 2 Path Decomposition

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 Chain Decomposition

In this section, we present a path concatenation technique that takes as input a 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 must 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.

3.1 Path Concatenation

Our concatenation algorithm can work in combination with every path decomposition algorithm. 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. 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 $O(k_p * m)$ 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 3 Path 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 ∈ D do
        f_i ← first vertex of p_i
        (R_i, P_i) ← reverse_DFS_lookup(G, f_i)
        if P_i ≠ ∅ then
            l_i ← destination vertex of P_i          ▷ Last vertex of a path
            Merge_Paths( l_i, f_i )
        end if
        G ← G \ R_i
    end for
end procedure
```

Algorithm 2 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 3.1. *The time complexity of algorithm 3 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 $i \neq j$. R_i and R_j are vertex disjoint sets. We conclude that $\bigcup_{i=0}^{k_p-1} R_i \subseteq N$ and $\sum_{i=0}^{k_p-1} |R_i| \leq |N|$.

Every path on the graph cannot be longer than the longest path. P_i, $0 \leq i \leq k_p$, 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 reduce the number of chains by one, $c = k_p - k_c$. □

3.2 Chain Decomposition Heuristic: A Better Approach

Previously, we described how to produce a chain decomposition applying a post-processing path concatenation step. At this point, we will demonstrate an approach which not only runs in O(m+c*l) time. It also finds a close to optimal chain decomposition.

Algorithm 4 is our Node Order Heuristic variation. 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 the vertex with the

biggest 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, examines all the outgoing (line 8) and all the incoming edges (line 19). Hence, the time complexity is linear.

Algorithm 5 illustrates our path decomposition of Algorithm 4 in combination 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.

Algorithm 4 Path Decomposition

```
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 ← ∅                                                    ▷ Set of paths
3:      for every vertex v_i ∈ T in ascending topological order do
4:          Chain C
5:          if u_i is assigned to a chain then
6:              C ← u_i's chain
7:          else if v_i is not assigned to a chain then
8:              l_i ← choose the immediate predecessor with the lowest outdegree
9:                    that is the last vertex of a chain
10:             if l_i ≠ null then
11:                 C ← path indicated by l_i
12:                 add v_i to C
13:             else
14:                 C ← 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
```

Algorithm 5 Chain Decomposition

```
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 ← ∅                                                    ▷ Set of paths
3:     for every vertex v_i ∈ T in ascending topological order do
4:         Chain C
5:         if u_i is assigned to a chain then
6:             C ← u_i's chain
7:         else if v_i is not assigned to a chain then
8:             l_i ← 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) ← reverse_DFS_lookup(G, u_i)
12:                if P_i ≠ ∅ then
13:                    l_i ← destination vertex of P_i
14:                end if
15:                G ← G \ R_i
16:            end if
17:            if l_i ≠ null then
18:                C ← path indicated by l_i
19:                add v_i to C
20:            else
21:                C ← 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
```

3.3 Experiments

In this section, we present experiments on graphs created by NetworkX [12]. We used three different random graph generator models. Erdos-Renyi, Barabasi, and Watts-Strogatz 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 obtain a minimum set of chains by using the Fulkerson's method [7]. Our aim is to reveal the behavior of the width and the behavior of the heuristics in these models.

Fulkerson's method:

1. Construct transitive closure $G^*(V, E')$ of the graph, where $V = \{v_1, ..., v_n\}$.
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**: 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**: 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.
- **Watts–Strogatz**: This model returns a Watts–Strogatz small-world graph. It firstly 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.

In order to make the directed graphs acyclic, we remove the edges in which the target vertex has a bigger id than the source vertex. For more info about the generators see networkx documentation.

Table 1 shows the number of chains created by the heuristics for every graph of 5000 nodes. Table 2 does it respectively for the graphs of 10000 nodes. The tables' abbreviations are explained below:

- **CO**: Path decomposition using Chain Order Heuristic.
- **CO conc.**: Chain decomposition using Chain Order Heuristic and our concatenation technique (post-processing step)
- **NO**: Path decomposition using Node Order Heuristic.
- **NO conc.**: Chain decomposition using Node Order Heuristic and our concatenation technique (post-processing step)

- **H3**: Path decomposition using our Node Order Heuristic variation from section 3.2.
- **H3 conc.**: Chain decomposition using our technique from section 3.2.
- **Width**: The width of the graph (Fulkerson's method).

As you 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, and 5, 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 2a and 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 inverse function $width = \frac{\text{nodes}}{\text{average degree}}$.

In this section, we do not present runtime metrics for two reasons. The first is that we do it in the forthcoming section, see tables 4,5, and 3. The second is that all heuristics run in a few milliseconds, and there is no reason for comparison since all scale up on large graphs, much larger than those of 10000 nodes.

$|N|=5000$

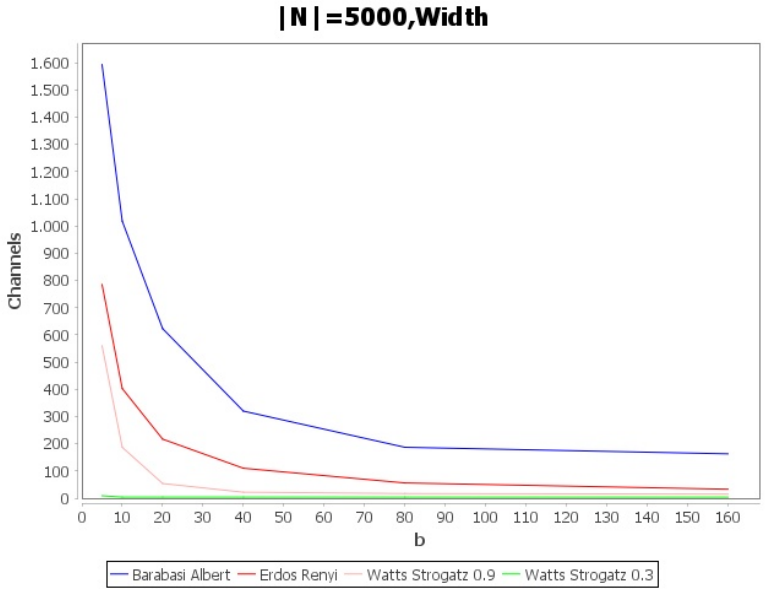
Av. Degree		5	10	20	40	80	160
		Barabasi Albert					
CO		1722	1178	801	471	296	189
CO conc.		1686	1127	747	411	252	164
NO		1792	1250	827	516	306	193
NO conc.		1743	1174	774	445	284	187
H3		1658	1102	720	424	256	165
H3 conc.		1630	1055	664	355	207	163
Width		1593	1018	623	320	187	163
		Erdos Renyi					
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

Table 1: Comparing path and chain decomposition algorithms on graphs with 5000 nodes.

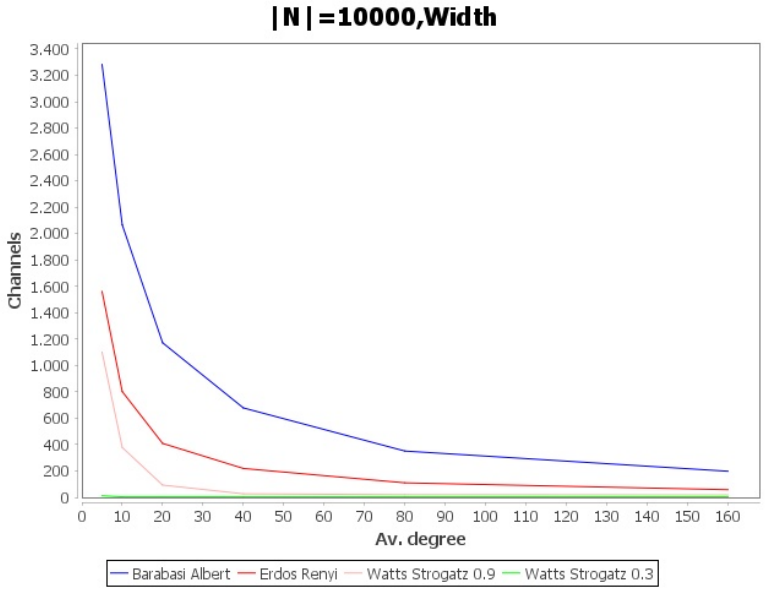
$|N|=10000$

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

Table 2: Comparing path and chain decomposition algorithms on graphs with 10000 nodes.



(a) $y = x$



(b) $y = 3sinx$

Figure 2: Three simple graphs

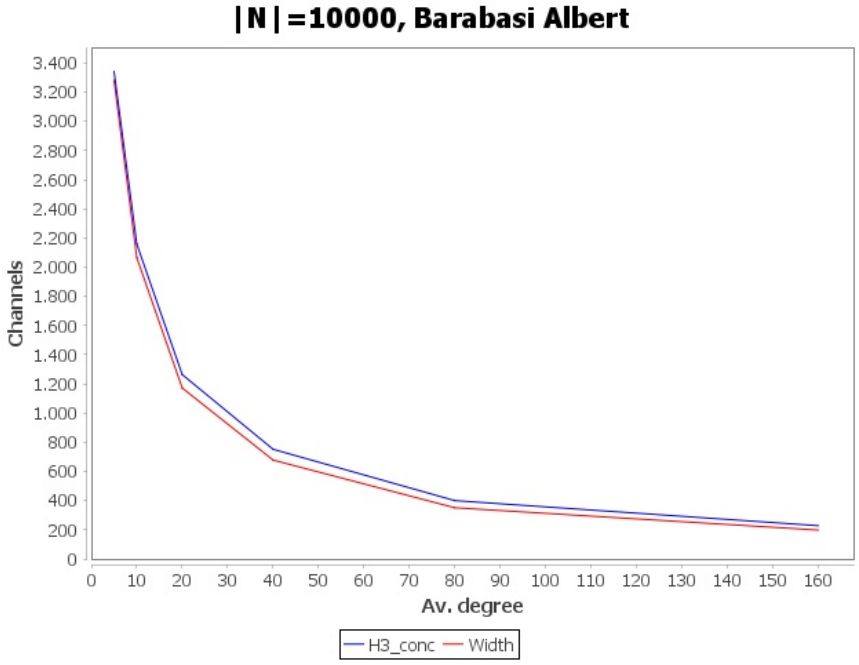


Figure 3: An example graph

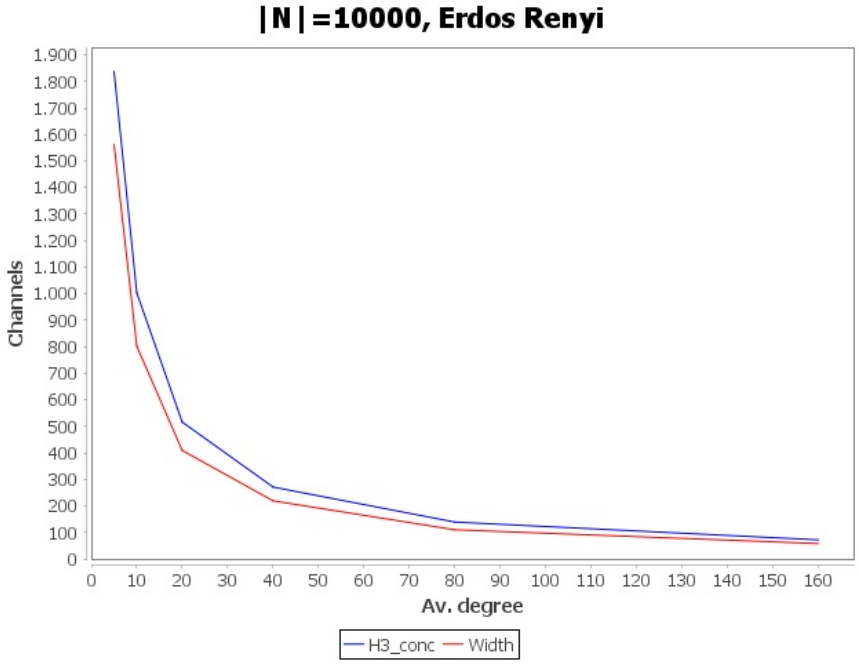


Figure 4: An example graph

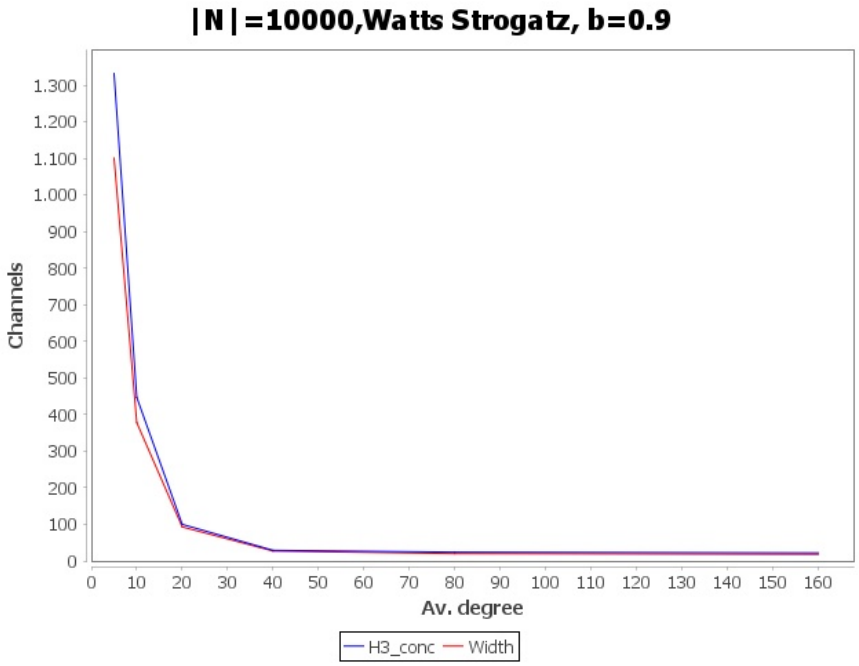


Figure 5: An example graph

4 Hierarchies and Transitivity

Proposition 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 of Proposition 1. 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 6a. □

Proposition 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 of Proposition 2. Similar to the proof of proposition 1. See figure 6b. □

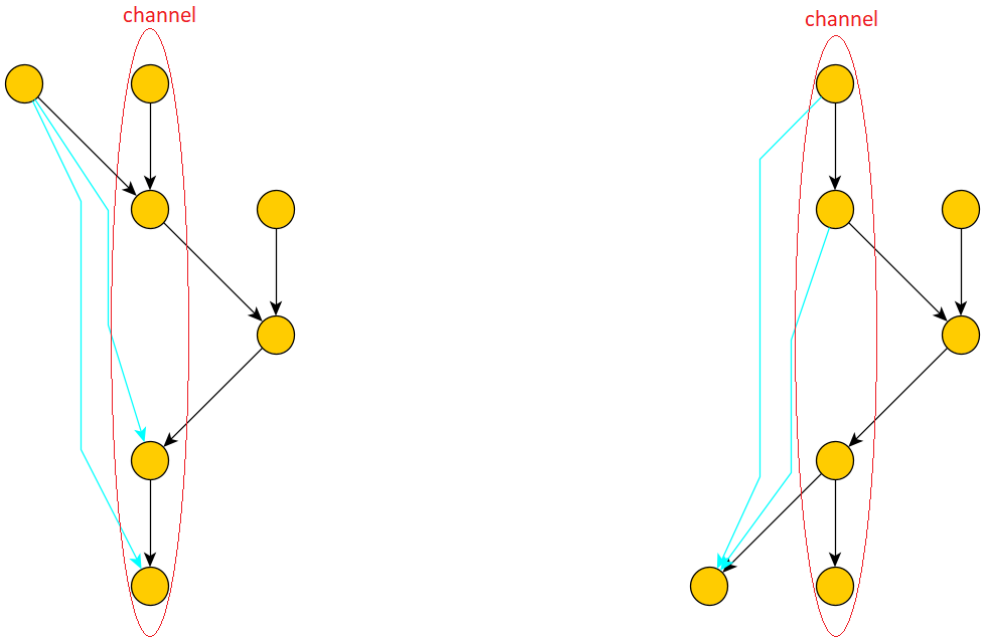


Figure 6: The blue edges are transitive. (a) shows the outgoing transitive edges that end to the same chain. (b) shows the incoming transitive edges that start from the same chain.

Theorem 4.1. *The non-transitive edges of a DAG $G(V, E)$ are less or equal to $width * |V|$, $E_{red} = E - E_{tr} \leq width * |V|$.*

Proof of Theorem 4.1. Given any DAG G and its width w, there is a chain decomposition of G with w number of chains. From Proposition 1, every vertex of G could

have only one outgoing, non-transitive edge per chain, thus its non-transitive outgoing edges can not be more than $w * |V|$. Notice that the same stands for the incoming edges, according to Proposition 2. □

According to theorem 4.1, the time complexity of Algorithm 5 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 many shorter chains. Hence, for most of the graphs is not even possible $E_{red} = width * |V|$, E_{red} usually is much less than that. We present experiments in table 4 and 5.

Also, an essential application of proposition 1 and 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 arcs 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 edges, and the indegree and outdegree of every vertex by k_c. For example, algorithms based on tree cover, see [2, 5, 20, 21], are practical on sparse graphs and can be enhanced further with a preprocessing step that removes transitive edges.

5 Indexing Scheme

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

Jagadish described that indexing scheme in 1990. As we told, 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 the precomputed transitive closure, and the outcome is very close to the optimal.

Simon, see [18], built that indexing scheme too. He calculates a path decomposition, boosting the method presented in [11]. The linear time heuristic he presented is Chain Order Heuristic. Our technique follows the same steps for the creation of the indexing scheme. The differentiation is that we do not merely do path decomposition, we do chain decomposition using our algorithm. Simon's algorithm needs $O(k_p * |E_{red}|)$ time and $O(k_p * n)$ space (k_p is the number of paths).

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

By finding the strongly connected components, we can make any graph acyclic. All vertices of a SCC will form a supernode since any vertex is reachable from any other vertex in the same component. That 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 $O(m+c*l)$. Simon performs path decomposition that runs in $O(n+m)$. In step 2, we sort the adjacency lists in O(n+m) time. Lastly, we create the indexing scheme in $O(k_c * |E_{red}|)$ time and $O(k_c * n)$ space. If we had done merely path decomposition, the time complexity would be $O(k_p * |E_{red}|)$ and $O(k_p * n)$ space. Probably, you have already noticed the relation between step 1 and step 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 with k_c size for every vertex. You can see the example in figure 7. 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 7 has $(1, 1)$. The node belongs to the $1st$ chain, and it is the $1st$ element in it. Given the chain decomposition, we can easily construct the pairs in O(n) time with a chain decomposition traversal. 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 [1,2,3]. 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 $2nd$ 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 $3rd$ 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 source 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 or equal to T's position, else it cannot.

5.2 Sorting Adjacency lists

Algorithm 6 sorts the adjacency list of every vertex. More precisely, it sorts the adjacency lists of immediate successors in ascending topological order. 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, $1 \leq i \leq n$, pushes v_i in the stacks of all immediate predecessors. This step could take part even before the chain decomposition as a preprocessing step. We present it in this section to emphasize its crucial role in indexing scheme creation. If the adjacency list is not sorted the time complexity of the algorithm would be $O(k_c * |E|)$ and not $O(k_c * |E_{red}|)$.

Proposition 3. *Algorithm 6 sorts the adjacency lists of immediate successors in ascending topological order.*

Proof of Proposition 6. 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

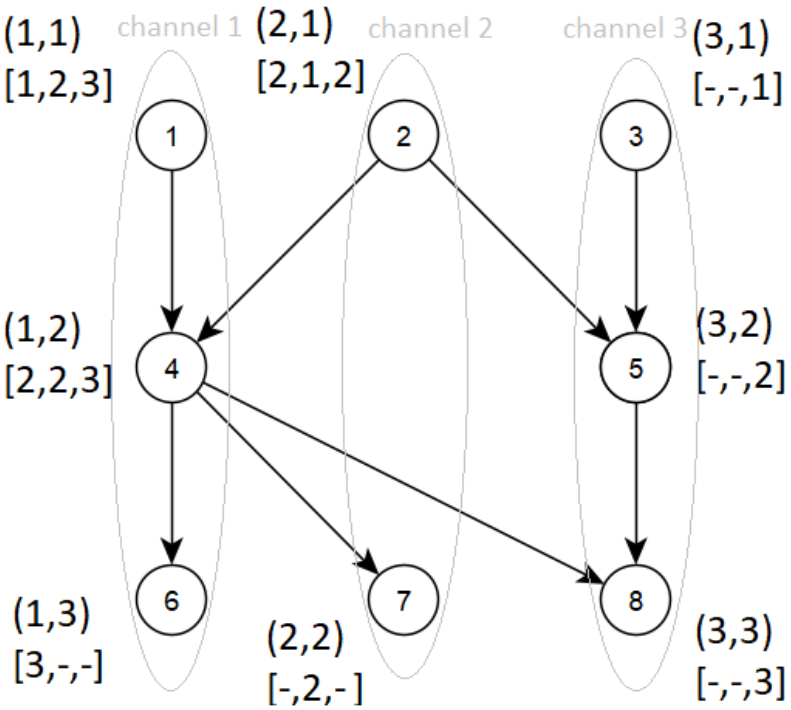


Figure 7: An example of an indexing scheme.

Algorithm 6 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 ∈ 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.add(v_i)
        end for
    end for
end procedure
```

topological rank than u_k and u_j precedes u_k. That means 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. □

5.3 Creating the Indexing Scheme.

Algorithm 7 constructs the indexing scheme. The first for-loop initializes the array of indexes. For every vertex, initializes the cell that corresponds to its chain. The rest of the cells initializes with infinite. The indexing scheme initialization illustrated in figure 8. The dashes represent the infinite. Notice that after the initialization, the indexes of all sink vertices have been calculated. Since they have no successors, the only vertex they can reach is themselves.

The second for-loop builds the indexing scheme. It goes through vertices in descending topological order. For each vertex, it visits its immediate successors (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), $0 \leq i \leq kc$, the new value of a_i is $\min\{a_i\ ,\ b_i\}$. This process needs k_c steps.

Proposition 4. *Given a vertex v and the calculated indexes of its successors, the while-loop of algorithm 7 (lines 10-17) calculates the indexes of v by updating its array with its non-transitive outgoing edges' successors.*

Proof of Proposition 4. Updating the indexes of vertex v with all its immediate successors will make v aware of all its descendants. The while-loop of Algorithm 7 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 it is transitive, we know by definition that exists a path from v to t with a length of more than 1. Suppose that the path is (v,v1,..,t). The vertex v1 is a predecessor of t and

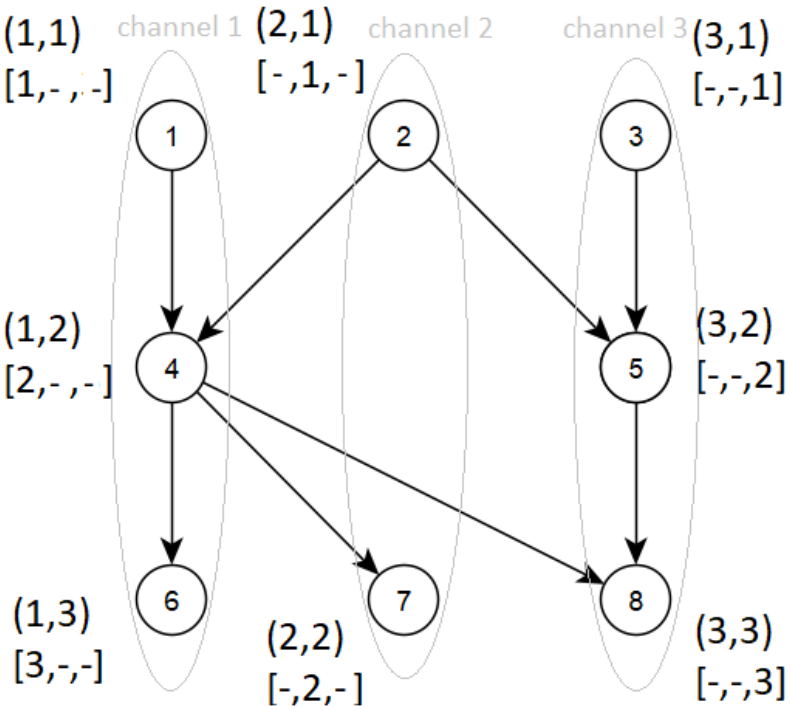


Figure 8: Initialization of indexes.

Algorithm 7 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 ∈ G do
3:         v_i.indexes ← new table[size of D]
4:         v_i.indexes.fill(∞)
5:         ch_no ← v_i's chain index
6:         pos ← v_i's chain position
7:         v_i.indexes[ ch_no ] ← pos
8:     end for
9:     for each vertex v_i in reverse topological order do
10:        while v_i.stack ≠ ∅ do
11:            target ← v_j.stack.pop()
12:            t_ch ← target's chain index
13:            t_pos ← 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
```

immediate successor of v. Hence it has a lower topological rank than t. The while-loop examines the incident vertices in ascending topological order. Hence, vertex t gets visited after vertex v1. The opposite leads to a contradiction. Consequently, for every incident transitive edge of v, the loop firstly visits a vertex v1 which is a predecessor of t. Thus vertex v will be updated by v1 and aware that the edge (v,t) is transitive. There is no reason to update vertex v indexes since the indexes of t will be greater or equal □

Theorem 5.1. *Algorithm 7 computes the indexing scheme in* $O(k_c * |E_{red}|)$ *time.*

Proof of Theorem 5.1. In the initialization step, the indexes of all sink vertices have been computed as we described before. 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 proposition 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 proposition 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. □

5.4 Experiments

We used the same graph of 5000 and 10000 nodes we described in 3.3 produced by three different models of the Networkx. We performed chain decomposition using our approach (H3_conc, Alg. 5), and created the indexing scheme (Alg. 7). Assume the

sorting of the adjacency list is a preprocessing step (Alg. 6). We recorded our results in Tables 4 and 5. Table 4 holds the results of graphs with 5000 nodes, and Table 5 the results of graphs with 10000 nodes. Next, we explain tables' columns.

- **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 9 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. 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 4 and 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 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 1 and 2 include metrics of creating the indexing scheme using different decomposition techniques on Erdos Reyni graphs of 10000 nodes. In table 1, we have

Av. Degree		Channels	CO Time (ms)	Indexing Scheme Time(ms)	Total
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
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 5 for chain decomposition.

Table 3: The tables present the run time of indexing scheme using path and chain decomposition.

created the indexing scheme using the chain order heuristic(path decomposition), while in table 2, we use our chain decomposition algorithm.

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	0.14	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	0.76	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

Table 4: Metrics on graphs of 5000 nodes.

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

Table 5: Metrics on graphs of 10000 nodes.

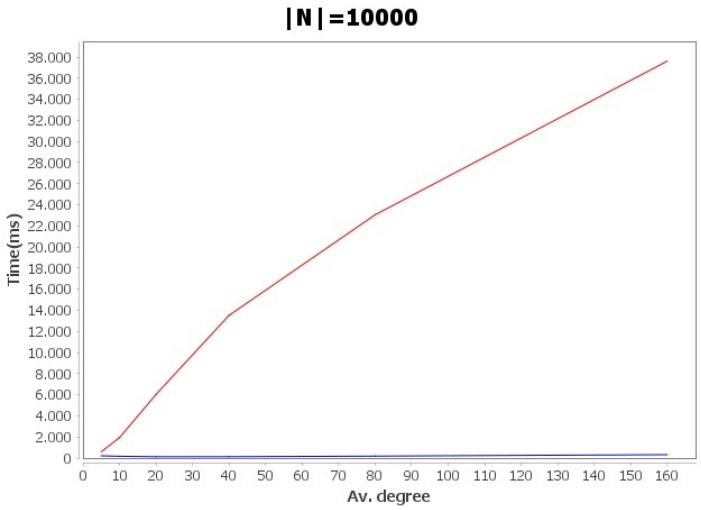


Figure 9: Run time comparison between the Indexing Scheme (blue line) and TC (red line) for Erdos-Renyi model on graphs of 10000 nodes. See table 5.

6 Conclusions

In this work, we show not only that finding a chain decomposition can be done in almost linear time but also the number of chains can be very close to the minimum. Our 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 a wide area of applications. We applied them to the problem of transitive closure. We built with the most efficient way a known indexing scheme to answer queries in constant time. The time complexity is $O(k_c*|E_{red}|)$, and the space complexity is $O(k_c*|V|)$. Similarly, we ran experimental work revealing the practical efficiency of this approach, especially for dense graphs.

Acknowledgement

Many thanks to Mr. Ioannis Tollis since he was the person I could share my thoughts with throughout my research. His advice and our discussions always were constructive and influential.

References

[1] R. Agrawal. "Alpha: an extension of relational algebra to express a class of recursive queries". In: *IEEE Transactions on Software Engineering* 14.7 (1988), pp. 879–885. DOI: 10.1109/32.42731.

[2] Rakesh Agrawal, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. "Efficient management of transitive relationships in large data and knowledge bases". In: *ACM SIGMOD Record* 18.2 (1989), pp. 253–262.

[3] A. V. Aho, M. R. Garey, and J. D. Ullman. "The transitive reduction of a directed graph". In: *SIAM Journal on Computing* 1.2 (1972), pp. 131–137.

[4] Albert-László Barabási and Réka Albert. "Emergence of scaling in random networks". In: *science* 286.5439 (1999), pp. 509–512.

[5] 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*. Citeseer. 2005, pp. 493–504.

[6] R. P. DILWORTH. "A decomposition theorem for partially ordered sets". In: *Ann. Math.* 52 (1950), pp. 161–166.

[7] Fulkerson DR. "Note on Dilworth's embedding theorem for partially ordered sets". In: *Proc. Amer. Math. Soc.* 52.7 (1956), pp. 701–702.

[8] P Erdős. "RÉNYI, A.:" On random graphs". In: *I". Publicationes Mathematicae (Debre* (1959).

[9] Delbert Ray Fulkerson. "Note on Dilworth's decomposition theorem for partially ordered sets". In: *Proc. Amer. Math. Soc.* Vol. 7. 4. 1956, pp. 701–702.

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

[11] Alla Goralčıková and Václav Koubek. "A reduct-and-closure algorithm for graphs". In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 1979, pp. 301–307.

[12] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[13] H. V. JAGADISH. "A Compression Technique to Materialize Transitive Closure". In: *ACM Trans. Database Systems* 15.4 (1990), pp. 558–598.

[14] *JFree*. URL: www.jfree.org.

[15] 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*. 2019, pp. 593–595.

[16] Giacomo Ortali and Ioannis G. Tollis. "A New Framework for Hierarchical Drawings". In: *Journal of Graph Algorithms and Applications* 23.3 (2019), pp. 553–578. DOI: 10.7155/jgaa.00502.

[17] Micha A Perles. "A proof of Dilworth's decomposition theorem for partially ordered sets". In: *Israel Journal of Mathematics* 1.2 (1963), pp. 105–107.

[18] K. SIMON. “An improved algorithm for transitive closure on acyclic digraphs”. In: *Theor. Comput. Sci.* 58.1-3 (1988), pp. 325–346.

[19] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 1971, pp. 114–121. DOI: 10.1109/SWAT.1971.10.

[20] 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*. 2007, pp. 845–856.

[21] Haixun Wang et al. “Dual labeling: Answering graph reachability queries in constant time”. In: *22nd International Conference on Data Engineering (ICDE'06)*. IEEE. 2006, pp. 75–75.

[22] Duncan J Watts and Steven H Strogatz. “Collective dynamics of 'small-world' networks”. In: *nature* 393.6684 (1998), pp. 440–442.

[23] *yWorks*. URL: www.yworks.com.