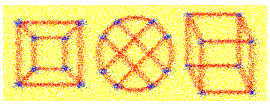

Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 30, no. 1, pp. 177–205 (2026)
DOI: 10.7155/jgaa.v30i1.3148

Parameterized Linear Time Transitive Closure

Giorgos Kritikakis[1] *Ioannis G. Tollis*[1]

[1]University of Crete, Heraklion, Greece

Submitted: Dec. 2025 Accepted: April 2026 Published: June 2026

Article type: Regular paper Communicated by: L. Georgiadis

Abstract. In this paper, we first study the problem of decomposing a *directed acyclic graph* (DAG), $G = (V, E)$ into vertex-disjoint chains and present a fast and practical chain decomposition technique. Our algorithm produces results that are very close to the optimum. The availability of fast and practical chain decomposition algorithms opens the way for novel solutions to fundamental problems. Building on our chain decomposition method, we present fast and practical techniques for computing the reachability information in a directed graph (i.e., its transitive closure). We focus on applicable solutions that enable constant-time reachability queries. Given any path/chain decomposition of G, our technique computes a reachability indexing scheme of a DAG, $G = (V, E)$ in parameterized linear time.

The experimental results reveal that our method is very fast in practice, indicating that chain decomposition algorithms can be applied to obtain fast and practical solutions to the transitive closure (TC) problem. Furthermore, we show that we can find a substantially large subset of the transitive edges of G in linear time using any chain decomposition. Our extensive experimental results show the interplay between these concepts in various models of DAGs. Additionally, we show the efficiency of this solution by speeding up Fulkerson's method for finding the minimum chain decomposition that corresponds exactly to the width of the graph.

1 Introduction

Computing the transitive closure or reachability information of a directed graph is fundamental in computer science and is an important step in many applications. Given a directed graph $G = (V, E)$, the transitive closure of G, denoted as $G^* = (V, E^*)$ such that E^* contains all the edges in E, and for any pair of vertices $u, v \in V$, if there exists a directed path from u to v in G, then there is a directed edge from u to v in E^*. An edge (v_1, v_2) of a DAG G is transitive if there is a path longer than one edge that connects v_1 and v_2. Given a directed graph G with

A preliminary version of part of this paper appeared in the 21st Symposium on Experimental Algorithms (SEA 2023).

E-mail addresses: georgecretek@gmail.com (Giorgos Kritikakis) tollis@csd.uoc.gr (Ioannis G. Tollis)


This work is licensed under the terms of the CC-BY license.

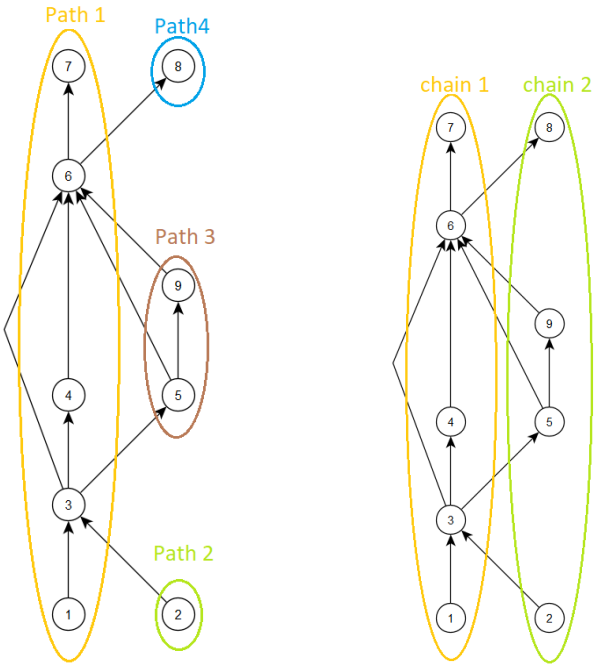
cycles, we can find the strongly connected components (SCC) of G in linear time and collapse all vertices of a SCC into a supernode. Clearly, any reachability query can be reduced to a query in the resulting *Directed Acyclic Graph* (DAG), or checking if the two vertices are in the same supernode. Additionally, DAGs are very important in many applications in several areas of research and business because they often represent hierarchical relationships between objects in a structure. Any DAG can be decomposed into vertex disjoint *paths* or *chains*. In a path, every vertex is connected to its successor by an edge, while in a chain any vertex is connected to its successor by a non-empty directed path, which may be an edge. A *path/chain decomposition* is a set of vertex disjoint paths/chains that cover all the vertices of a DAG (see Figure 1). Two chains (or paths) c_1 and c_2 can be merged into a new chain if the last vertex of c_1 can reach the first vertex of c_2, or if the last vertex of c_2 can reach the first vertex of c_1. The process of merging two or more paths or chains into a new chain is called path or chain *concatenation*.

The *width* of a DAG $G = (V, E)$ is the maximum number of mutually unreachable vertices of G [12]. An *optimum chain decomposition* of a DAG G contains the minimum number of chains, which is equal to the width of G. Due to the multitude of applications, there are several algorithms to find a chain decomposition of a DAG, see for example [24, 13, 15, 11, 39, 5, 6, 27, 47, 28, 8, 10]. Some of them find the optimum, and some are heuristics. Generally speaking, the algorithms that compute the optimum take more than linear time and use flow techniques, which are often heavy and complicated to implement. However, for several practical applications, it is not necessary to compute an optimum chain decomposition. The computation of an efficient chain decomposition of a DAG has many applications in several areas including bioinformatics [4, 19], evolutionary computation [25], databases [24], graph drawing [40, 41, 36], distributed systems [23, 45].

In Section 2, we review some basic path and chain decomposition approaches, and introduce a new efficient way to create a chain decomposition. Our approach creates a number of chains that is very close to the optimum. We analyze and utilize this algorithm to accelerate transitive closure solutions. Furthermore, we present experimental results that show that the width of a DAG drops as the graph becomes denser. In fact, we compute the width for four different random DAG models to obtain insights about its behavior as the size of the graphs grows. An interesting observation is that the width of DAGs created using the Erdős-Rényi (ER) random graph model is proportional to $\frac{\text{number of nodes}}{\text{average degree}}$. Next, in Section 3, we show that $|E_{red}| \leq width * |V|$, and describe how to remove a significant subset of transitive edges, E'_{tr}, in linear time, in order to bound $|E - E'_{tr}|$ by $O(k * V)$ given any path/chain decomposition of size k. Clearly, this can boost many known transitive closure solutions. Section 4 demonstrates how to build an indexing scheme that implicitly contains the transitive closure and we report experimental results that shed light on the interplay of width, E_{red}, and E_{tr}.

We consider reachability mainly for the static case, i.e., when the graph does not change. The question of whether an arbitrary vertex v can reach another arbitrary vertex u can be answered in linear time by running a breadth-first or depth-first search from v, or it can be answered in constant time after a reachability indexing scheme or transitive closure of the graph has been computed. The transitive closure of a graph with $|V|$ vertices and $|E|$ edges can be computed in $O(|V| * |E|)$ time by starting a breadth-first or depth-first search from each vertex. Alternatively, one can use the Floyd-Warshall algorithm [16] which runs in $O(|V|^3)$, or solutions based on matrix multiplication [43]. Currently, the best-known bound on the asymptotic complexity of a matrix multiplication algorithm is $O(|V|^{2.371339})$ time [2]. However, these and other similar improvements to Strassen's Algorithm are not used in practice because they are complicated and the constant coefficient hidden by the notation is extremely large. Here we focus on computing a reachability indexing scheme in parameterized linear time. Notice that we do not explicitly compute the

transitive closure matrix of a DAG. The matrix can be easily computed from the reachability indexing scheme in $O(|V|^2)$ time (i.e., constant time per entry).



(a) A path decomposition of a graph consisting of 4 paths.

(b) A chain decomposition of a graph consisting of 2 chains.

Figure 1: Path and chain decomposition of an example graph.

In this paper we present a practical algorithm to compute a reachability indexing scheme (or the transitive closure information) of a DAG $G = (V, E)$, utilizing any given path/chain decomposition. The scheme can be computed in parameterized linear time, where the parameter is the number of paths/chains, k_c, in the given decomposition and it can answer any reachability query in constant time. Let E_{tr}, $E_{tr} \subset E$, denote the set of transitive edges and E_{red}, $E_{red} = E - E_{tr}$, denote the set of non-transitive edges of G. We show that $|E_{red}| \leq width * |V|$ and that we can compute a substantially large subset of E_{tr} in linear time (see Section 3). This implies that any DAG can be reduced to a smaller DAG that has the same transitive closure (as it was originally observed in [37]) in linear time, bounding the total number of edges by $O(k_c * |V|)$ and the degree of each vertex by $O(k_c)$. Consequently, several reachability algorithms will run much faster in practice. The time complexity to produce the reachability indexing scheme is $O(|E_{tr}| + k_c * |E_{red}|) = O(|E_{tr}| + k_c * width * |V|) = O(|E_{tr}| + k_c^2 * |V|)$, and its space complexity is $O(k_c * |V|)$ (see Section 4). Our experimental results reveal the practical efficiency of this approach. In fact, the results show that our method is substantially better in practice than the theoretical bounds imply, indicating that path/chain decomposition algorithms can be used to solve the transitive closure (TC) problem and compute the TC matrix in $O(|V|^2)$ time. This paper extends and unifies ideas and experiments initiated in [30, 31, 32, 29].

2 Width of a DAG and Decomposition into Paths/Chains

In this section, we briefly describe some categories of path and chain decomposition techniques, we introduce a new chain decomposition heuristic and show experimental results. The experiments explore the practical efficiency of our chain decomposition approach and the behavior of the width in different graph models. We focus on fast and practical path/chain decomposition heuristics. There are two categories of path decomposition algorithms, Node Order Heuristic, and Chain Order Heuristic, see [24]. The first constructs the paths one by one, while the second creates the paths in parallel. The chain-order heuristic starts from a vertex and extends the path to the extent possible. The path ends when no more unused immediate successors can be found. The node-order heuristic examines each vertex (node) and assigns it to an existing path. If no such path exists, then a new path is created for the vertex. We assume that the vertices are topologically ordered, since tracing the vertices in topological order is highly beneficial for creating compact decompositions. The importance of topological sorting on path decomposition has already been described in [24].

In addition to path-decomposition algorithm categorization, Jagadish in [24] describes some chain decomposition heuristics. Those heuristics run in $O(|V|^2)$ time using a pre-computed transitive closure, which is not linear, and we will not discuss them further. In this work, we do the opposite: we first compute an efficient chain decomposition very fast and use it to construct transitive closure solutions.

More precisely our algorithm decomposes a DAG into chains in $O(|E| + c * l)$ time, where c is the number of path concatenations, and l is the length of a longest path of the DAG (without requiring any precomputation of the transitive closure). One may argue that the worst-case time complexity of this technique is $O(|V|^2)$, since $c < |V|$ and $l < |V|$. However, we show theoretically and experimentally that the factor $c * l$ is rarely larger than the number of edges ($|E|$). Hence, it does not only have the tightest theoretical bound but also behaves purely like a linear time algorithm.

2.1 Path/Chain Concatenation

To reduce the number of chains of any given chain decomposition, we can find possible concatenations and merge the chains. Searching for a concatenation implies that we are searching for a path between two chains. We can start searching from the first vertex of a chain looking for the last vertex of another chain, or from the last vertex of a chain looking for the first vertex of another chain. A path decomposition can be constructed employing linear-time algorithms (like those presented in [24]), after which the resulting paths can be concatenated to produce a chain decomposition.

Given a DAG $G = (V, E)$ and a path decomposition D_p that contains k_p paths, we will build a chain decomposition of G that contains k_c chains in $O(|E| + (k_p - k_c) * l)$ time, where l is the length of a longest path of G. This is accomplished by performing path/chain concatenations. Since each path/chain concatenation reduces the number of chains by one, the total number of such concatenations is $(k_p - k_c)$. Since paths are by definition chains of a special structure, and we start by concatenating paths into chains, we use the more general term chain in our algorithms.

For every path in D_p we start a reversed DFS lookup function from the first vertex of a chain, looking for the last vertex of another chain traversing the edges backward. The reversed DFS lookup function is the well-known depth-first search graph traversal for path finding. If the reversed DFS lookup function detects the last vertex of a chain, then it concatenates the chains. If

we simply perform the above, as described, then the algorithm will run in $O(k_p * |E|)$ since we will run k_p reversed DFS searches. However, every reversed DFS search can benefit from the previous reversed DFS results. A reversed DFS for path finding returns the path between the source vertex and the target vertex, which in our case, is the path between the first vertex of a chain and the last vertex of another chain. Hence, every execution that goes through a set of vertices V_i it splits them into two vertex disjoint sets, R_i and P_i. The set P_i contains the vertices of the path from the source vertex to the destination vertex and R_i contains 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 set R_i is not the last vertex of a chain. If it were then it would belong to P_i and not to R_i. Similarly, for every vertex in R_i, all its predecessors are in R_i too. Hence, if a forthcoming reversed DFS lookup function meets a vertex of R_i, there is no reason to proceed with its predecessors.

Algorithm 1 Concatenation

```
1:  procedure CONCATENATION(G, D)
    INPUT: A DAG G = (V, E), and a path/chain decomposition D of G
    OUTPUT: A chain decomposition of G
2:      for each chain: p_i ∈ D do
3:          f_i ← first vertex of p_i
4:          (R_i, P_i) ← reversed_DFS_lookup(G, f_i)
5:          if P_i ≠ ∅ then
6:              l_i ← destination vertex of P_i          ▷ Last vertex of a chain
7:              Concatenate_Chains( l_i, f_i )
8:          end if
9:          G ← G \ R_i
10:     end for
11: end procedure
```

Algorithm 1 shows our path/chain concatenation technique. Observe that the reversed DFS lookup function is invoked for every starting vertex of a chain. Every reversed 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 chain. The set R_i contains all of the vertices the function went through except the vertices of P_i. Hence we have the following theorem:

Theorem 1. *The time complexity of Algorithm 1 is $O(|E| + (k_p - k_c) * l)$, where l is the length of a longest path in G.*

Proof: Assume that we have k_p paths. We call the reversed_DFS_lookup function k_p times. 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 $\bigcup_{i=0}^{k_p-1} R_i \subseteq V$ and $\sum_{i=0}^{k_p-1} |R_i| \leq |V|$.

A path/chain P_i, $0 \leq i < k_p$, is not empty if and only if a 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 paths/chains by one, we have that $c = k_p - k_c$. Evidently, the $O(|E|)$ factor arises because of the DFS exploration. □

Notice that according to the previous proof the actual time complexity of Algorithm 1 is

$O(|E| + \sum_{i=0}^{k_p-1} |P_i|)$ which in practice is expected to be significantly better than $O(|E| + c * l)$. This observation is further supported by our experimental results. In particular, we conducted experiments on Erdős–Rényi graphs with 10k, 20k, 40k, 80k, and 160k vertices and an average degree of 10. The corresponding running times were 9, 34, 99, 228, and 538 milliseconds, respectively, which demonstrates that the execution time scales linearly with the input size.

2.2 Efficient Chain decomposition

The concatenation technique we propose is fast in both theory and practice. However, it is also important that the results remain close to the optimum, which requires some additional steps. Traditionally, concatenation is applied after constructing a path decomposition. However, our experiments indicate that this approach can lead to incorrect matches because of concatenations that involve transitive edges. By incorrect match we refer to a concatenation of two chains that results in a one-unit increase in the final decomposition size, as it blocks a proper match with another chain. An edge (v_1, v_2) of a DAG G is said to be transitive if there exists a path of length greater than one connecting v_1 and v_2. If an edge cannot be immediately matched to extend a path, then a search for an available path to match should be initiated; otherwise, a potentially valid path might be incorrectly matched through a transitive edge of a subsequent vertex. Our results indicate that the common practice of treating the concatenation step as a separate postprocessing phase warrants reconsideration.

Algorithm 2 constructs the chains using our concatenation technique on the fly. As can be seen, the block of the if-statement of line 10 is run online instead of running Algorithm 1 as a post-processing step. In other words, if we do not find an immediate predecessor, we search all predecessors using the reversed_DFS_lookup function. The reversed_DFS_lookup function is applied in real-time if the algorithm does not find an immediate predecessor that is the last vertex of a chain.

Furthermore, Algorithm 2 adds two extra greedy choices: (a) when we visit a vertex of out-degree 1, we immediately add its unique immediate successor to its path/chain, and (b) instead of searching for the first available immediate predecessor (that is the last vertex of a path), we choose an available vertex with the lowest out-degree. These steps have a secondary effect on the decomposition outcome reducing its size by a few chains. Still, we keep them since they do not affect the theoretical or practical efficiency.

2.3 DAG Decomposition: Experimental Results

Our algorithm is the fastest in both theory and practice. Rather than focusing on runtime, which we report in Tables 2 and 3 in Section 4.4 , we emphasize how closely the output approaches the optimum. In this section, we present experimental results on graphs, most of which were created by NetworkX [20]. We use three different random graph generator models: Erdős-Rényi [14] (ER), Barabasi-Albert [3] (BA) and Watts-Strogatz [49] (WS) models.

The generated graphs are made acyclic, by orienting all edges from low to high ID numbers, see the documentation of networkx [20] for more information about the generators. Additionally, we use the Path-Based DAG Model (PB) that was introduced in [38] and is especially designed for DAGs with a predefined number of randomly created paths. For every model, we created 12 types of graphs: Six types of 5000 nodes and six types of 10000 nodes, both with average degrees 5, 10, 20, 40, 80, and 160. We used different average degrees to have results for various sizes and densities.

Algorithm 2 Chain Decomposition (NH conc.)

```
1: procedure New Heuristic 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 chain decomposition of G
2:     K ← ∅                                                    ▷ Set of chains
3:     for every vertex v_i ∈ T in ascending topological order do
4:         Chain C                                              ▷ C is a pointer to a chain
5:         if v_i is assigned to a chain then
6:             C ← v_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, v_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 ← chain 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 v_i with in-degree 1 then
27:            add s_i to C
28:        end if
29:    end for
30:    return K
31: end procedure
```

We ran the heuristics on multiple copies of graphs and examined the performance of the heuristics in terms of the number of chains in the produced chain decompositions. All experiments were conducted on a simple laptop PC (Intel(R) Core(TM) i5-6200U CPU, with 8 GB of main memory). Our algorithms have been developed as stand-alone java programs. We observed that the graphs generated by the same generator with the same parameters have a small width deviation. For example, the percentage of deviation on the ER and Path-Based model is about 5% and for the BA model is less than 10%. The width deviation of the graphs in the WS model is slightly higher, but this is expected since the width of these graphs is significantly smaller.

Random Graph Generators:

- **Erdős-Rényi (ER) model** [14]: The generator returns a random graph $G_{n,p}$, where n is the number of nodes and every edge is formed with probability p.
- **Barabási–Albert (BA) model** [3]: Preferential attachment model: 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 of the generator.
- **Watts–Strogatz (WS) model** [49]: Small-world graphs: 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 an existing node w. The factors n, k, b are the parameters of the generator.
- **Path-Based DAG (PB) model** [38]: In this model, graphs are randomly generated based on a predefined number of randomly created paths.

We compute the minimum set of chains using the method of Fulkerson [17], which in brief consists of the following steps: 1) Construct the transitive closure $G^*(V, E')$ of G, where $V = \{v_1, ..., v_n\}$. 2) Construct a bipartite graph B with partitions (V_1, V_2), where $V_1 = \{x_1, x_2, ..., x_n\}$, $V_2 = \{y_1, y_2, ..., y_n\}$. 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.

The aim of our experiments is twofold: (a) to understand the behavior of the width of DAGs created in different models, and (b) to compare the behavior of our heuristic used on graphs of these models. Table 1 shows the width and the number of chains created by our algorithm for all graphs of 5000 and 10000 nodes. Observe that our chain decomposition heuristic, Algorithm 2, computes a chain decomposition that is very close to the optimum (width).

- **NH conc.**: Chain decomposition using Algorithm 2
- **Width**: The width of the graph (computed by Fulkerson's method).

Understanding the width in DAGS: In order to understand the behavior of the width on DAGs of these different models we observe: (i) the BA model produces graphs with a larger width than ER, and (ii) the ER model creates graphs with a larger width than WS. For the WS model, we created two sets of graphs: The first has probability b equals 0.9 and the second 0.3. Clearly, if the probability b of rewiring an edge is 0, the width would be one, since the generator initially creates a path that goes through all vertices. As the rewiring probability b grows, the width grows. This is the reason we choose a low and a high rewiring probability. Figures 2a and 2b demonstrate the behavior of the width for each model on the graphs of 5000 and 10000 nodes. Please notice that in almost all model graphs (except WS 0.3) the width of a DAG decreases fast as the density of the DAG increases. Additionally, it is interesting to observe here that the width of the ER model graphs is proportional to $\frac{\text{number of nodes}}{\text{average degree}}$.

Finally, we provide a comparison of the width of the ER and Path-Based model graphs, shown in Figure 4, for graphs of 10000 nodes and varying average degrees. We show this separately since the important details would not be visible in Figure 2b. It is interesting to note that for sparser PB graphs the width is very close to the number of predefined (but randomly created paths) whereas

	$\|V\| = 5000$						$\|V\| = 10000$					
Av. Degree	5	10	20	40	80	160	5	10	20	40	80	160
BA												
NH conc	1630	1055	664	355	207	163	3341	2159	1264	752	400	228
Width	1593	1018	623	320	187	163	3282	2066	1172	678	351	198
ER												
NH conc	923	492	252	139	70	38	1837	1003	516	271	139	72
Width	785	403	217	110	56	33	1561	802	409	219	110	58
WS, b=0.9												
NH conc	687	212	60	25	20	17	1332	447	100	29	24	22
Width	560	187	54	22	17	15	1101	378	93	27	20	18
WS, b=0.3												
NH conc	9	4	4	5	4	5	12	4	4	4	4	4
Width	9	4	4	4	4	4	12	4	4	4	4	4
PB, Paths=100												
NH conc	86	101	107	93	73	51	125	141	153	142	120	89
Width	70	70	70	68	58	30	100	100	100	99	90	47

Table 1: The number of chains produced by Algorithm 2, compared to the optimum, for graphs with 5000 and 10000 nodes.

the width of the ER graphs is very high. However, as the graphs become denser, the width for both models seems to eventually converge.

Number of chains versus width in DAGS: In order to compare the number of chains produced by our approach on the graphs of these models, besides the aforementioned tables, we also created several figures. Namely, we visualize how close is the number of chains produced by our heuristics to the width. In particular, Figure 3 and Figures 8, 9, and 10 in the Appendix show how close is the number of chains produced by our technique (i.e., the blue line) to the width (i.e., red line) for ER, BA, WS, and PB graph models.

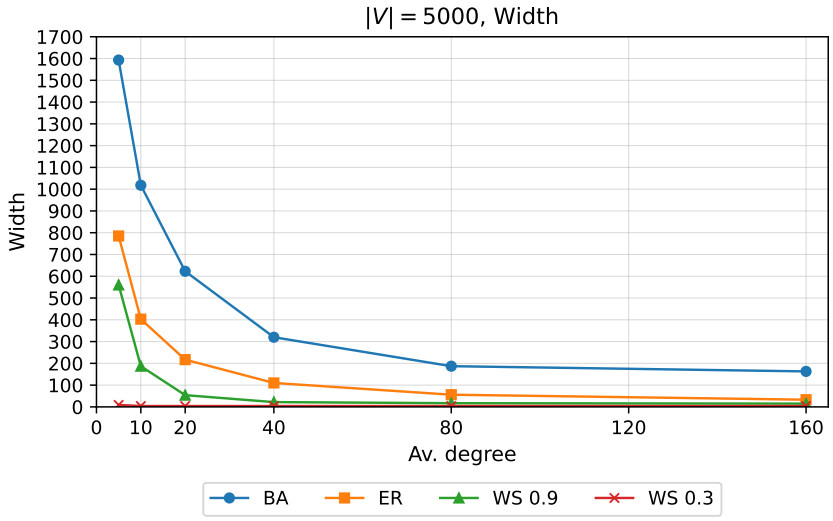
Our heuristics run very fast, as expected, in just a few milliseconds, see Tables 2 and 3 in Section 4.4. Thus, it is not interesting to elaborate further on their running time in this subsection.

3 DAG Reduction for Faster Transitivity

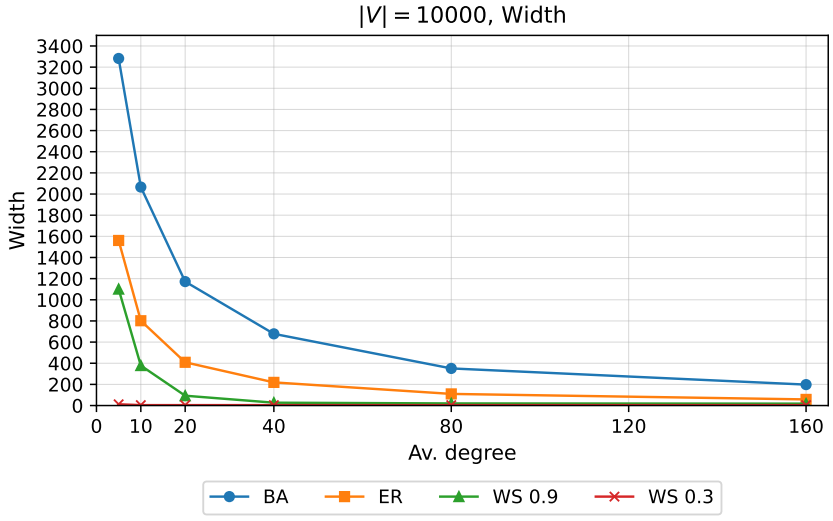
In this section, we show how to produce a linear time algorithm for the reduction of transitive edges, see Algorithm 3, which in combination with our fast chain decomposition, Algorithm 2, materialize the idea of a general-purpose reduction technique based on chain decomposition. This concept of reduction or abstraction of a DAG is useful in several applications beyond transitive closure. In [37] it was used to visualize (reduced) hierarchical graphs while displaying full reachability information. Therefore, we state the following useful lemmas and Theorem 2:

Lemma 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 that the vertex v has two outgoing edges, (v, t_1) and (v, t_2), and both t_1 and t_2 are in chain



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



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

Figure 2: The width curve on graphs of 5000 and 10000 nodes using three different models.

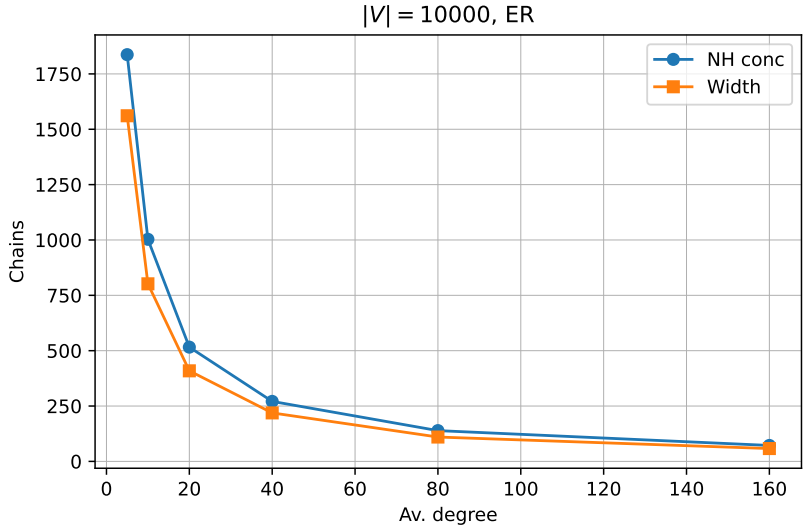


Figure 3: The efficiency of our chain decomposition algorithm in the ER model.

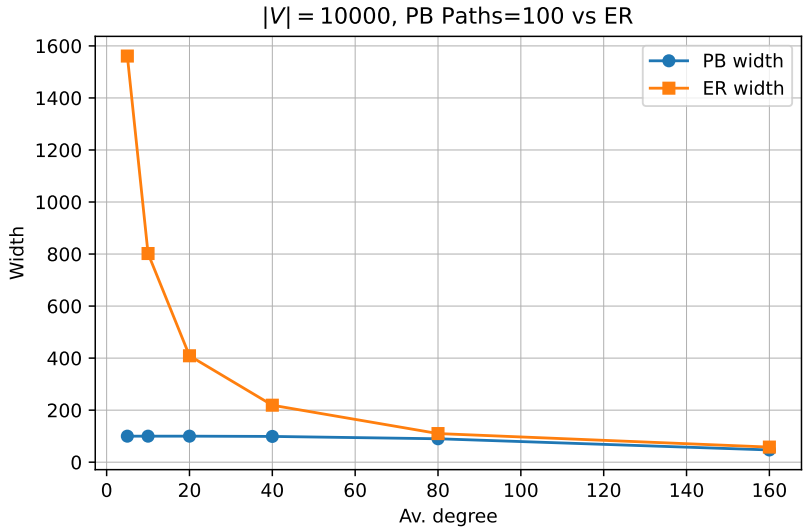


Figure 4: A comparison of width between ER model and PB model.

C_i. The vertices are in ascending topological order in the chain by definition. Assume t_1 has a lower topological rank than t_2. 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 5a. □

Lemma 2. *Given any 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 Lemma 1, see Figure 5b. □

Theorem 2. *Let $G = (V, E)$ be a DAG with width w. The non-transitive edges of G are less than or equal to $w * |V|$, in other words $|E_{red}| = |E| - |E_{tr}| \leq w * |V|$.*

Proof: Given any DAG G and its width w, there is a chain decomposition of G with w number of chains. By Lemma 1, every vertex of G could have only one outgoing, non-transitive edge per chain. The same holds for the incoming edges, according to Lemma 2. Thus the non-transitive edges of G are bounded by $w * |V|$. □

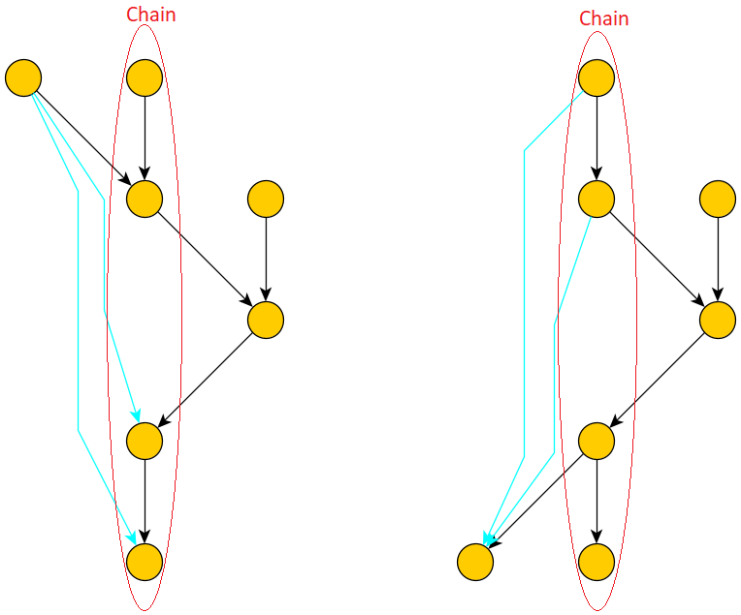


Figure 5: The light blue edges are transitive. (a) shows the outgoing transitive edges that end in the same chain. (b) shows the incoming transitive edges that start from the same chain.

An interesting application of the above is that we can find a significantly large subset of E_{tr} in linear time. Given any chain (or 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 (i.e., the vertex with the highest topological rank) of each chain.

Algorithm 3, is a linear time algorithm that reduces the outgoing transitive edges. The algorithm traverses all the vertices (outer loop) and utilizes a k_c-size array. For each vertex, it initializes the k_c-size array within the first inner loop. Then, it populates the array with the edges having the lowest rank for every chain during the second inner loop. Finally, it retrieves the reduced edge list from the k_c-size array in the third inner loop. This algorithm is simple and efficient but is not the only nor the most advanced way to reduce the graph in linear time using a chain decomposition. For example, we could examine the vertices in descending topological order, chain by chain, while maintaining an additional chain-value indicator in the k-sized array for each edge. By doing so, we

would be able to remove additional transitive edges, such as (u,t), if there exists an edge (u',t') where u' is a predecessor of u in the chain and t' is a successor of t in their chain. This can be done in total linear time. To maintain our concise description approach to transitive closure, we refrain from elaborating on these details.

In this fashion we find a superset of E_{red}, call it E'_{red}, in linear time. Equivalently, we can find $E'_{tr} = E - E'_{red}$. E'_{tr} is a significantly large subset of E_{tr} since $|E - E'_{tr}| = |E'_{red}| \leq k_c * |V|$. Clearly, this approach can be used as a linear-time preprocessing step in order to substantially reduce the size of any DAG while keeping the same transitive closure as the original DAG G. Consequently, this will speed up every transitive closure algorithm bounding the number of edges of any input graph, and the indegree and outdegree of every vertex by k_c. For example, algorithms based on tree cover, see [1, 9, 46, 48], are practical on sparse graphs and can be enhanced further with such a preprocessing step that removes transitive edges. Additionally, this approach may have practical applications in dynamic or hybrid transitive closure techniques: If one chooses to answer queries online by using graph traversal for every query, one could reduce the size of the graph with a fast (linear-time) preprocessing step that utilizes chains. Also, in the case of insertion/deletion of edges one could quickly decide if the edges to add or remove are transitive. Transitive edges do not affect the transitive closure, hence no updates are required. This could be practically useful in the dynamic insertion/deletion of edges.

Algorithm 3 Reduction of outgoing edges.

```
1:  procedure REDUCTION(G, D)
    INPUT: A DAG G = (V, E), and the decomposition D of size k of G.
2:      incidentEdges[] ← new array of size k that holds edges.
3:      for each vertex: v ∈ G do
4:          for every outgoing edge e(v, t)  do
5:              chain ← The chain number of vertex t.
6:              incidentEdges[chain] ← e
7:          end for
8:          for every outgoing edge e(v, t)  do
9:              chain ← The chain number of vertex t.
10:             e'(v, t') ← incidentEdges[chain]
11:             if  t' succeeds t in the chain then
12:                 incidentEdges[chain] ← e
13:             end if
14:         end for
15:         reducedAdjList ← new empty list.
16:         for every outgoing edge e(v, t)  do
17:             chain ← The chain number of vertex t.
18:             e'(v, t') ← incidentEdges[chain]
19:             if t' == t then
20:                 reducedAdjList.add(t)
21:             end if
22:         end for
23:         v.adjTargetList ← reducedAdjList
24:     end for
25: end procedure
```

4 Reachability Indexing Scheme

In this section, we present an important application that uses a chain decomposition of a DAG. Namely, we solve the transitive closure problem by creating a reachability indexing scheme that is based on a chain decomposition and we evaluate it by running extensive experiments. Our experiments shed light on the interplay of various important factors as the density of the graphs increases.

Jagadish described a compressed transitive closure technique in 1990 [24] by applying an indexing scheme and simple path/chain decomposition techniques. His method uses successor lists and focuses on the compression of the transitive closure. Simon [42], describes a technique similar to [24]. His technique is based on computing a path decomposition, thus boosting the method presented in [18]. The linear time heuristic used by Simon is similar to the Chain Order Heuristic of [24]. A different approach is a graph structure referred to as path-tree cover introduced in [26], similarly, the authors utilize a path decomposition algorithm to build their labeling.

In the following subsections, we describe how to compute an indexing scheme in $O(|E_{tr}| + k_c * |E_{red}|)$ time, where k_c is the number of chains (in any given chain decomposition) and $|E_{red}|$ is the number of non-transitive edges. Following the observations of Section 3, the time complexity of the scheme can be expressed as $O(|E_{tr}| + k_c * |E_{red}|) = O(|E_{tr}| + k_c * width * |V|)$ since $|E_{red}| \leq width * |V|$. Using an approach similar to Simon's [42] our scheme creates arrays of indices to answer queries in constant time. The space complexity is $O(k_c * |V|)$.

For our experiments, we utilize algorithm 2 of Section 2.2. This heuristic performs better than any path decomposition algorithm, as will be explained next. Thus, the indexing scheme is more efficient in terms of both time and space requirements. Furthermore, the experimental work shows that, as expected, the chains rarely have the same length. Usually, a decomposition consists of a few long chains and several short chains. Hence, for most graphs, it is not even possible to have $|E_{red}| = width * |V|$, which assumes the worst case for the length of each chain. In fact, $|E_{red}|$ is usually much lower than that, and the experimental results presented in Tables 2 and 3 confirm this observation in practice.

Given a directed graph with cycles, we can find the strongly connected components (SCC) in linear time. Since any vertex is reachable from any other vertex in the same SCC (they form an equivalence class), all vertices in an SCC can be collapsed into a supernode. Hence, any reachability query can be reduced to a query in the resulting directed acyclic graph (DAG). This is a well-known step that has been widely used in many applications. Therefore, without loss of generality, we assume that the input graph of our method is a DAG. The following general steps describe how to compute the reachability indexing scheme:

1. Compute a Chain decomposition
2. Sort all Adjacency Lists
3. Create an Indexing Scheme

In Step 1, we use our chain decomposition technique that runs in $O(|E| + c * l)$ time. In Step 2, we sort all the adjacency lists in $O(|V| + |E|)$ time. Finally, we create an indexing scheme in $O(|E_{tr}| + k_c * |E_{red}|)$ time and $O(k_c * |V|)$ space. Clearly, if the algorithm of Step 1 computes fewer chains, then Step 3 becomes more efficient in terms of time and space.

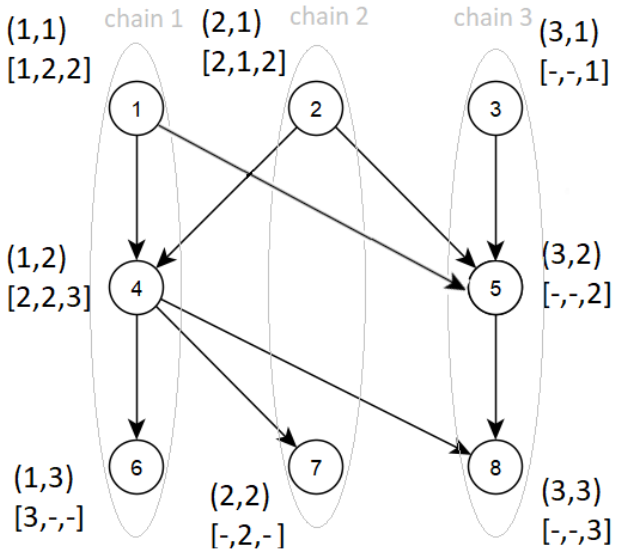


Figure 6: An example of an indexing scheme.

4.1 The Indexing Scheme

Given any chain decomposition of a DAG G with size k_c, an indexing scheme will be computed for every vertex that includes a pair of integers and an array of size k_c indexes. A small example is shown in Figure 6. 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 6 has a pair $(1, 1)$. This means that vertex 1 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 using a simple traversal of the chains. Every entry 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 that the vertex can reach. For example, the array of vertex 1 is $[1, 2, 2]$. The first cell of the array indicates that vertex 1 can reach the first vertex of the first chain (can reach itself, reflexive property). The second cell of the array indicates that vertex 1 can reach the second 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 second vertex of the third chain.

Notice that we do not need the second integer of all pairs. If we know the chain a vertex belongs to, we can conclude its position using the array. We use this presentation to simplify the understanding of the reader.

The process of answering a reachability query is simple. Assume, there is a source vertex s and a target vertex t. To find if vertex t is reachable from s, we first find the chain of t, and we use it as an index in the array of s. 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.

4.2 Sorting the Adjacency lists

Next, we use a linear time algorithm to sort all the adjacency lists of immediate successors in ascending topological order. See Algorithm 4. The algorithm maintains a stack for every vertex that indicates the sorted adjacency list. Then it 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 into all immediate predecessors' stacks. This step can be performed as a preprocessing step, even before receiving the chain decomposition. To emphasize its crucial role in the efficient creation of the indexing scheme, if the lists are not sorted then the second part of the time complexity would be $O(k_c * |E|)$ instead of $O(k_c * |E_{red}|)$.

Algorithm 4 Sorting Adjacency lists

```
1:  procedure SORT(G, t)
    INPUT: A DAG G = (V, E)
2:      for each vertex: v_i ∈ G do
3:          v_i.stack ← new stack()
4:      end for
5:      for each vertex v_i in reverse topological order do
6:          for every incoming edge e(s_j, v_i) do
7:              s_j.stack.push(v_i)
8:          end for
9:      end for
10: end procedure
```

Lemma 3. *Algorithm 4 sorts the adjacency lists of immediate successors in ascending topological order, in linear time.*

Proof: Assume that there is a stack $(u_1, ..., u_n)$, u_1 is at 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. This means that the for-loop examined u_j before u_k. Since the algorithm processes the vertices in reverse topological order, this is a contradiction. Vertex u_j cannot precede u_k if it were examined first by the for-loop. The algorithm traces all the incoming edges of every vertex. Therefore, it runs in linear time. □

4.3 Creating the Indexing Scheme

Now we present Algorithm 5 that 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 infinity. The indexing scheme initialization is illustrated in Figure 7. The dashes represent the infinite values. 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 (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 an 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 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), $0 \leq i < k_c$, the new value of a_i will be $\min\{a_i, b_i\}$. This process needs k_c steps.

Lemma 4. *Given a vertex v and the calculated indexes of its successors, the for-loop of Algorithm 5 (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 5 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 $(v, v_1, .., t)$. 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. Hence there is no reason to update the indexes of vertex v with those of vertex t since the indexes of t will be greater than or equal to those of v. □

Algorithm 5 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:         for each adjacent target vertex t of v_i in ascending topological order do
11:             t_ch ← chain index of t
12:             t_pos ← chain position of t
13:             if t_pos < v_i.indexes[t_ch] then          ▷ (v_i, t) is not transitive
14:                 v_i.updateIndexes(t.indexes)
15:             end if
16:         end for
17:     end for
18: end procedure
```

Combining the previous algorithms and results we have the following:

Theorem 3. *Let $G = (V, E)$ be a DAG. Algorithm 5 computes an indexing scheme for G in $O(|E_{tr}| + k_c * |E_{red}|)$ time.*

Proof: 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

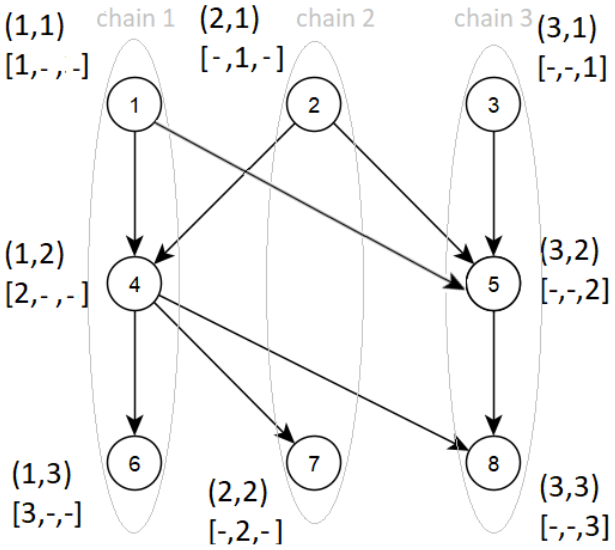


Figure 7: Initialization of indexes.

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 4, we can calculate its indexes, ignoring the transitive edges. Assume the for-loop has reached vertex v_i in the ith iteration, and the indexes of its successors are calculated. Following Lemma 4, we can calculate its indexes. Hence, by induction, we can calculate the indexes of all vertices, ignoring all $|E_{tr}|$ transitive edges in $O(|E_{tr}|+k_c*|E_{red}|)$ time. □

Since $O(|E_{tr}| + k_c * |E_{red}|) = O(|E_{tr}| + k_c * w * |V|) \leq O(|E_{tr}| + k_c^2 * |V|)$, we conclude that the transitive closure (indexing scheme) of G can be computed in parameterized linear time given any chain decomposition.

Corollary 1. *Let $G = (V, E)$ be a DAG. Algorithm 5, can be used to compute an indexing scheme (transitive closure) of G in parameterized linear time given any chain decomposition with k_c chains.*

Apparently, there is a trade-off to consider when building an indexing scheme deploying our chain decomposition technique, Algorithm 3. The heuristic performs concatenations between paths. For every successful concatenation, the extra runtime overhead is $O(l)$, where l is the longest path between the two concatenated paths. The unsuccessful concatenations do not cause any overhead. Assume that we have a path decomposition and then we perform chain concatenation. If there is no concatenation between two paths, the concatenation algorithm will run in linear time.

On the other hand, if there are concatenations, for each one of them, the cost is $O(l)$ time, but the savings in the indexing scheme creation is $\Theta(|V|)$ in space requirements and $\Theta(|E_{red}|)$ in time, since every concatenation reduces the required index size for every vertex by one. Hence, instead of computing a simple path decomposition, the use of our chain concatenation procedure to create a more compact indexing scheme faster is always preferred.

As described in the introduction, a parameterized linear-time algorithm for computing the minimum number of chains was recently presented in [6]. Its time complexity is $O(k^3|V| + |E|)$

where k is the minimum number of chains, which is equal to the width of G. If we use this chain decomposition as input to Algorithm 5 we have the following:

Corollary 2. *Let $G = (V, E)$ be a DAG. Algorithm 5 can compute an indexing scheme (transitive closure) of G in parameterized linear time in terms of width.*

4.4 Experimental Results

We conducted experiments using the same graphs of 5000 and 10000 nodes as we described in Section 2 that were produced by the four different models of Networkx [20] and the Path-Based model of [38]. We computed a chain decomposition using the Algorithm 2, and created an indexing scheme using Algorithm 5. For simplicity, we assume that the adjacency lists of the input graph are sorted, using Algorithm 4, as a preprocessing step. We report our experimental results in Tables 2 and 3 for graphs with 5000 nodes and graphs with 10000 nodes, respectively.

In theory, the phase of the indexing scheme creation requires $O(|E_{tr}|+k_c*|E_{red}|)$ time. However, the experimental results shown in the tables reveal some interesting (and expected) findings in practice: As the average degree increases and the graph becomes denser, (a) the cardinality of E_{red} remains almost stable; and (b) the number of chains decreases. The observation that the number of non-transitive edges, E_{red}, does not vary significantly as the average degree increases, implies that the number of transitive edges, $|E_{tr}|$, increases proportionally to the increase in the number of edges, since ($E_{tr} = E - E_{red}$). Since the algorithm merely traces in linear time the transitive edges, the growth of $|E_{tr}|$ affects the run time only linearly. As a result, the run time of our technique does not increase significantly as the size (number of edges) of the input graph increases. In order to demonstrate this fact visually, we show the curves of the running time for the graphs of 10000 nodes produced by the ER model in Figure 11 of the Appendix. The x-axis shows the average degree of the nodes, i.e., the increasing density of the graphs. The flat (blue line) represents the run time to compute the indexing scheme, and the curve (red line) represents the run time of the DFS-based algorithm for computing the transitive closure (TC) that utilizes a two-dimensional adjacency matrix, and has time complexity $O(|V| * |E|)$. Clearly, the time of the DFS-based algorithm increases as the average degree (density) increases, while the time of the indexing scheme is an almost straight line parallel to the x-axis, i.e., remains almost constant with respect to the density. All models, as shown in Tables 2 and 3, follow this pattern.

Another interesting and to some extent surprising observation that comes from the results of Tables 2 and 3 is that the transitive edges for almost all models of the graphs of 5000 and 10000 nodes with average degree above 20 are above 85%, i.e., $|E_{tr}|/|E| \geq 85\%$, see the appropriate columns in both tables. In some cases where the graphs are a bit denser, the percentage grows above 95%. This observation has important implications in designing practical algorithms for faster transitive closure computation in both the static and the dynamic case.

Additional experiments on real-world citation graphs are reported in Appendix B.

$\|V\| = 5000$								
Average Degree	Number of Chains	$\|E_{tr}\|$	$\|E_{red}\|$	$\|E_{tr}\|/\|E\|$	NH_conc Time (ms)	Indexing Scheme Time (ms)	Total time (ms)	TC
	BA							
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	335	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
	ER							
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
	WS, 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	460	9173
	WS, 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
	PB, 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 2: Experimental results for the indexing scheme for graphs of 5000 nodes.

$\|V\| = 10000$								
Average Degree	Number of Chains	$\|E_{tr}\|$	$\|E_{red}\|$	$\|E_{tr}\|/\|E\|$	NH_conc Time (ms)	Indexing Scheme Time (ms)	Total time (ms)	TC
	BA							
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
	ER							
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
	WS, 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
	WS, 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
	PB, 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: Experimental results for the indexing scheme for graphs of 10000 nodes.

4.5 Improving the Method of Fulkerson

In Section 2.3 we compute the minimum set of chains using the method of Fulkerson [13]. As described, this method consists of three steps. The first step is the computation of the transitive closure, the second step involves the construction of a bipartite graph, and the third entails the computation of a maximal matching. It has been a general belief for decades that the method of Fulkerson is not efficient (and thus not practically efficient) because of its dependency on the transitive closure problem. In this section, we show that the use of the indexing scheme (instead of the traditional transitive closure computation) in Fulkerson's method improves the computation of the width, as can be concluded by reviewing Tables 3 and 2, and Figure 11 in the Appendix.

Specifically, the indexing scheme needs $O(k_c^2 * |V| + |E_{tr}|)$ time, the construction of the bipartite graph has worst-case time complexity $O(|V|^2)$. The last step, maximal matching, requires $O(|E| * \sqrt{|V|})$ time, using the algorithm by Hopcroft–Karp [22]. Hence, the implementation of the method requires $O(k_c^2 * |V| + |V|^2 + |E| * \sqrt{|V|})$ time. In order to explore the practical significance of this we ran several experiments using ER-model graphs. Table 4, shows the runtime of each step of the Fulkerson method, and the final column shows the total runtime. Notice, that our transitive closure solution (indexing scheme), is not the most time-consuming step of this method, as was believed in the past. In fact it is the fastest step, and performs significantly better than the second step which is theoretically bounded by $O(|V|^2)$.

This intriguing result can be explained by our earlier experimental results. As shown in Tables 2 and 3 the vast majority of edges are transitive. In fact, the number of non-transitive edges seem to be linear with respect to the number of nodes, (often only a few times the number of nodes). More precisely, our experiments show that the non-transitive edges are always less than six times the number of nodes. Hence, assuming that the number of non-transitive edges is a small constant times the number of nodes, then $|E_{red}|$ is $O(|V|)$. Therefore, the expected runtime of the indexing scheme would be $O(k_c * |V| + |E_{tr}|)$.

Clearly, transitive closure solutions based on matrix multiplication cannot be faster than $O(|V|^2)$ since they perform computations on a two-dimensional adjacency matrix. The purpose of this sub-section is to show one application that directly benefits from the practical efficiency of our algorithms. By applying them to improve the running time of Fulkerson's method, we challenge a prevailing belief about its effectiveness. It is not our intention to delve deeper into algorithms for minimum chain decomposition. Experimental results on optimal chain decomposition algorithms can be found in [7].

5 Conclusions and Extensions

In this paper we describe a fast, simple, and practical technique to compute a chain decomposition of a DAG G that is not merely a path decomposition. Such decompositions are used to compute a transitive closure of G when needed. In particular, we focus on an indexing scheme that enables us to answer queries in constant time. We compute this scheme in parameterized linear time and space. Furthermore, we show how to reduce the graph in linear time given any path/chain decomposition removing transitive edges and how it can bolster transitive closure solutions. Our work explores techniques that attempt to provide fast and practical solutions. It is both theoretical and experimental, revealing crucial aspects of these problems. Our algorithms are applicable to real-world scenarios and can change the way we think about certain problems, as demonstrated by our experimental work on the method of Fulkerson, described in Section 4.5.

Although our techniques were not developed for the dynamic case, where edges and nodes

Fulkerson Method				
Average Degree	Indexing Scheme Time (ms)	Bipartite Graph Time (ms)	Hopcroft–Karp Maximal Matching Time (ms)	Total Time (ms)
	$\|V\| = 5000$			
5	146	538	566	1250
10	98	1072	3228	4398
20	108	1223	5267	6598
40	42	916	8137	9095
80	165	1177	12265	13607
160	268	1129	10219	11616
	$\|V\| = 10000$			
5	466	1129	10219	11814
10	207	2794	10613	13614
20	241	3580	28918	32739
40	177	3925	48249	52351
80	321	5053	55315	60689
160	220	6107	69958	76285

Table 4: Runtime evaluation of the Fulkerson Method utilizing the indexing scheme for ER graphs.

are added or deleted dynamically, the picture that emerges is very interesting. According to our experimental results, shown in Tables 2 and 3, the overwhelming majority of edges in a DAG are transitive. The insertion or deletion of a transitive edge clearly requires a constant time update since it does not affect transitivity, and can be detected in constant time. On the other hand, the insertion or removal of a non-transitive edge may require a minor or major recomputation to reestablish a correct chain decomposition. Similarly, since the nodes of the DAG are topologically ordered, the insertion of an edge that goes from a high node to a low node signifies that the SCCs of the graph will change, perhaps locally. However, even if the insertion/deletion of new nodes/edges causes significant changes in the reachability index (transitive closure) one can simply recompute a chain decomposition in linear or almost linear time, and then recompute the reachability scheme in parameterized linear time, $O(|E_{tr}| + k_c * |E_{red}|)$, and $O(k_c * |V|)$ space, which is still very efficient in practice. For a very recent comparison of practical fully dynamic transitive closure techniques see [21]. We plan to work on the problems that arise in the computation of dynamic path/chain decomposition and reachability indexes in the future.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD Record*, 18(2):253–262, 1989. doi:10.1145/66926.66950.

[2] J. Alman, R. Duan, V. V. Williams, Y. Xu, Z. Xu, and R. Zhou. More asymmetry yields faster matrix multiplication. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2005–2039. SIAM, 2025. doi:10.1137/1.9781611978322.63.

[3] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999. doi:10.1515/9781400841356.349.

[4] P. Bonizzoni. A linear-time algorithm for the perfect phylogeny haplotype problem. *Algorithmica*, 48(3):267–285, 2007. doi:10.1007/s00453-007-0094-3.

[5] M. Cáceres, M. Cairo, B. Mumey, R. Rizzi, and A. 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. doi:10.1007/978-3-030-86838-3_20.

[6] M. Cáceres, M. Cairo, B. Mumey, R. Rizzi, and A. I. Tomescu. Sparsifying, shrinking and splicing for minimum path cover in parameterized linear time. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 359–376. SIAM, 2022. doi:10.1137/1.9781611977073.18.

[7] M. Cáceres, B. Mumey, S. Toivonen, and A. I. Tomescu. Practical minimum path cover. In *International Symposium on Experimental Algorithms*, pages 1–19. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2024.

[8] M. A. Caceres Reyes. Minimum chain cover in almost linear time. *arXiv e-prints*, 2023.

[9] L. Chen, A. Gupta, and M. E. 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.

[10] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and S. Sachdeva. Almost-linear-time algorithms for maximum flow and minimum-cost flow. *Communications of the ACM*, 66(12):85–92, 2023. doi:10.1145/3610940.

[11] Y. Chen and Y. 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. doi:10.9734/bjmcs/2015/19380.

[12] R. P. DILWORTH. A decomposition theorem for partially ordered sets. *Ann. Math.*, 52:161–166, 1950. doi:10.1007/978-1-4899-3558-8_1.

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

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

[15] S. Felsner, V. Raghavan, and J. Spinrad. Recognition algorithms for orders of small width and graphs of small dilworth number. *Order*, 20(4):351–364, 2003. doi:10.1023/b:orde.0000034609.99940.fb.

[16] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962. doi:10.1145/367766.368168.

[17] D. R. Fulkerson. Note on dilworth's decomposition theorem for partially ordered sets. In *Proc. Amer. Math. Soc*, volume 7, pages 701–702, 1956. doi:10.1090/s0002-9939-1956-0078334-6.

[18] A. Goralčíková and V. Koubek. A reduct-and-closure algorithm for graphs. In *International Symposium on Mathematical Foundations of Computer Science*, pages 301–307. Springer, 1979. doi:10.1007/3-540-09526-8_27.

[19] J. Gramm, T. Nierhoff, R. Sharan, and T. Tantau. Haplotyping with missing data via perfect path phylogenies. *Discrete Applied Mathematics*, 155(6-7):788–805, 2007. doi:10.1016/j.dam.2005.09.020.

[20] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008. doi:10.25080/tcwv9851.

[21] K. Hanauer, M. Henzinger, and C. Schulz. Faster fully dynamic transitive closure in practice. *CoRR*, abs/2002.00813, 2020. URL: https://arxiv.org/abs/2002.00813, arXiv:2002.00813.

[22] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973. doi:10.1137/0202019.

[23] S. Ikiz and V. 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.

[24] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, Dec. 1990. URL: http://doi.acm.org/10.1145/99935.99944, doi:10.1145/99935.99944.

[25] W. Jaśkowski and K. Krawiec. Formal analysis, hardness, and algorithms for extracting internal structure of test-based problems. *Evolutionary computation*, 19(4):639–671, 2011. doi:10.1162/evco_a_00046.

[26] R. Jin, Y. Xiang, N. Ruan, and H. 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. doi:10.1145/1376616.1376677.

[27] S. Kogan and M. Parter. Beating matrix multiplication for nˆ{1/3}-directed shortcuts. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[28] S. Kogan and M. Parter. Faster and unified algorithms for diameter reducing shortcuts and minimum chain covers. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 212–239. SIAM, 2023. doi:10.1137/1.9781611977554.ch9.

[29] G. Kritikakis. Ongraphhierarchies: Source code and experimental implementation. https://github.com/GiorgosKritikakis/OnGraphHierarchies.

[30] G. Kritikakis. Analysis and visualization of hierarchical graphs. Master's thesis, University of Crete, February 2022.

[31] G. Kritikakis and I. G. Tollis. Fast and practical dag decomposition with reachability applications. *arXiv e-prints*, pages arXiv–2212, 2022.

[32] G. Kritikakis and I. G. Tollis. Fast reachability using dag decomposition. In *21st International Symposium on Experimental Algorithms (SEA 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

[33] J. Kunegis. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*, pages 1343–1350, 2013. URL: http://dl.acm.org/citation.cfm?id=2488173, doi:10.1145/2487788.2488173.

[34] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[35] M. Ley. The DBLP computer science bibliography: Evolution, research issues, perspectives. In *Proc. Int. Symposium on String Process. and Inf. Retr.*, pages 1–10, 2002. doi:10.1007/3-540-45735-6_1.

[36] P. Lionakis, G. Kritikakis, and I. G. Tollis. Algorithms and experiments comparing two hierarchical drawing frameworks. *arXiv preprint arXiv:2011.12155*, 2020.

[37] P. Lionakis, G. Ortali, and I. 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.

[38] P. Lionakis, G. Ortali, and I. G. Tollis. Constant-time reachability in dags using multi-dimensional dominance drawings. *SN Computer Science*, 2(4):1–14, 2021. doi:10.1007/s42979-021-00713-6.

[39] V. Mäkinen, A. I. Tomescu, A. Kuosmanen, T. Paavilainen, T. Gagie, and R. Chikhi. Sparse dynamic programming on dags with small width. *ACM Transactions on Algorithms (TALG)*, 15(2):1–21, 2019. doi:10.1145/3301312.

[40] G. Ortali and I. G. Tollis. Algorithms and bounds for drawing directed graphs. In *International Symposium on Graph Drawing and Network Visualization*, pages 579–592. Springer, 2018. doi:10.1007/978-3-030-04414-5_41.

[41] G. Ortali and I. G. Tollis. A new framework for hierarchical drawings. *Journal of Graph Algorithms and Applications*, 23(3):553–578, 2019. doi:10.7155/jgaa.00502.

[42] K. SIMON. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58(1-3):325–346, 1988. doi:10.1016/0304-3975(88)90032-1.

[43] V. Strassen et al. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969. doi:10.1007/bf02165411.

[44] L. Šubelj and M. Bajec. Model of complex networks based on citation dynamics. In *Proc. of the WWW Workshop on Large Scale Network Analysis*, pages 527–530, 2013. doi:10.1145/2487788.2487987.

[45] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. *Journal of Parallel and Distributed Computing*, 41(2):173–189, 1997. doi:10.1006/jpdc.1996.1298.

[46] S. Trißl and U. 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. doi:10.1145/1247480.1247573.

[47] J. Van Den Brand, Y. T. Lee, Y. P. Liu, T. Saranurak, A. Sidford, Z. Song, and D. Wang. Minimum cost flows, mdps, and $\ell 1$-regression in nearly linear time for dense instances. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 859–869, 2021.

[48] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 75–75. IEEE, 2006. doi:10.1109/icde.2006.53.

[49] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *nature*, 393(6684):440–442, 1998. doi:10.1515/9781400841356.301.

A Figures

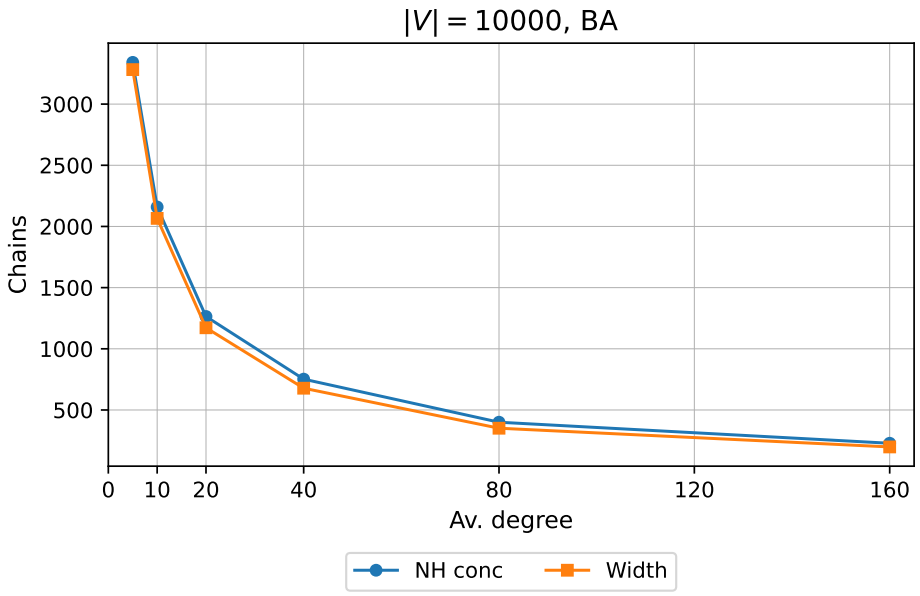


Figure 8: The efficiency of the chain decomposition algorithm in the BA model.

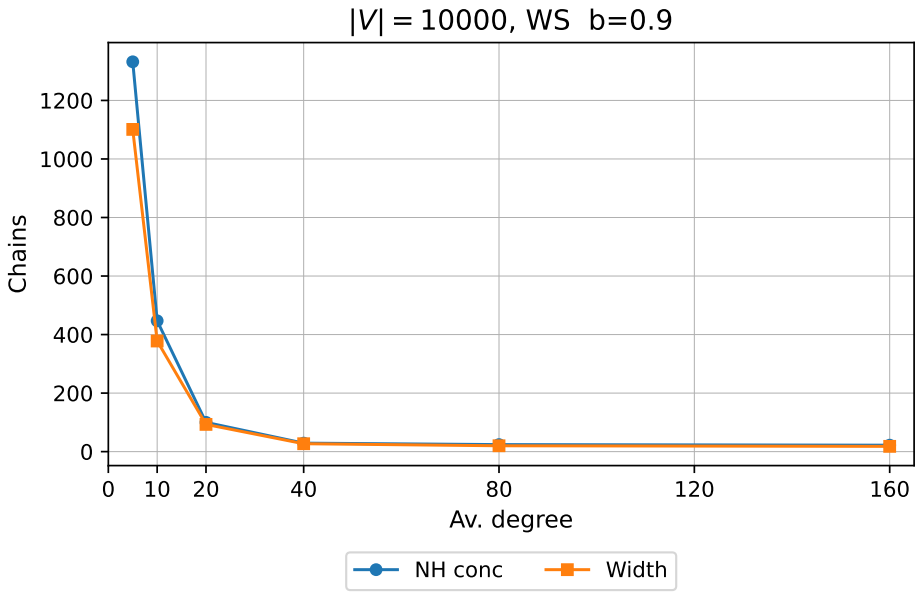


Figure 9: The efficiency of our chain decomposition algorithm in WS model.

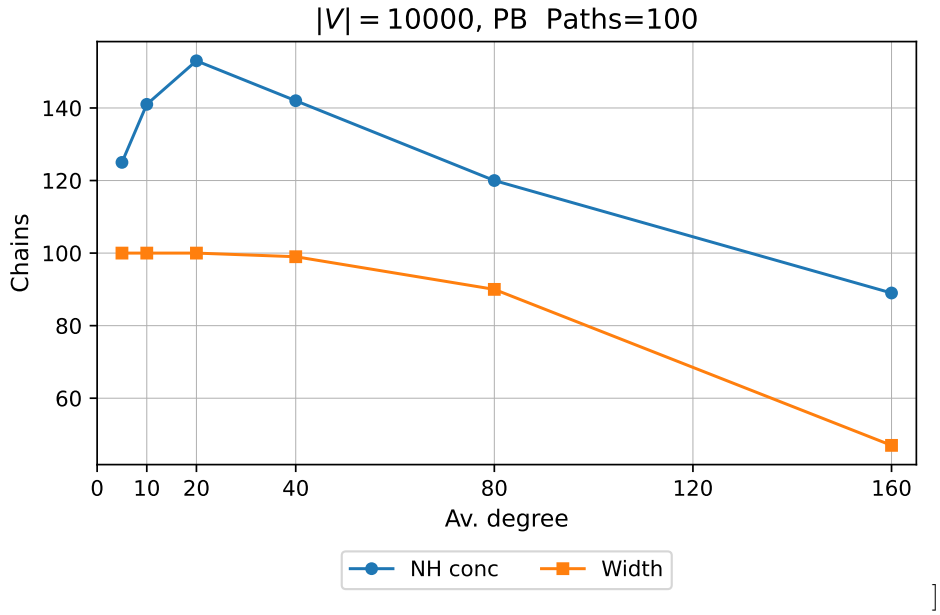


Figure 10: The efficiency of our chain decomposition algorithm in the PB model.

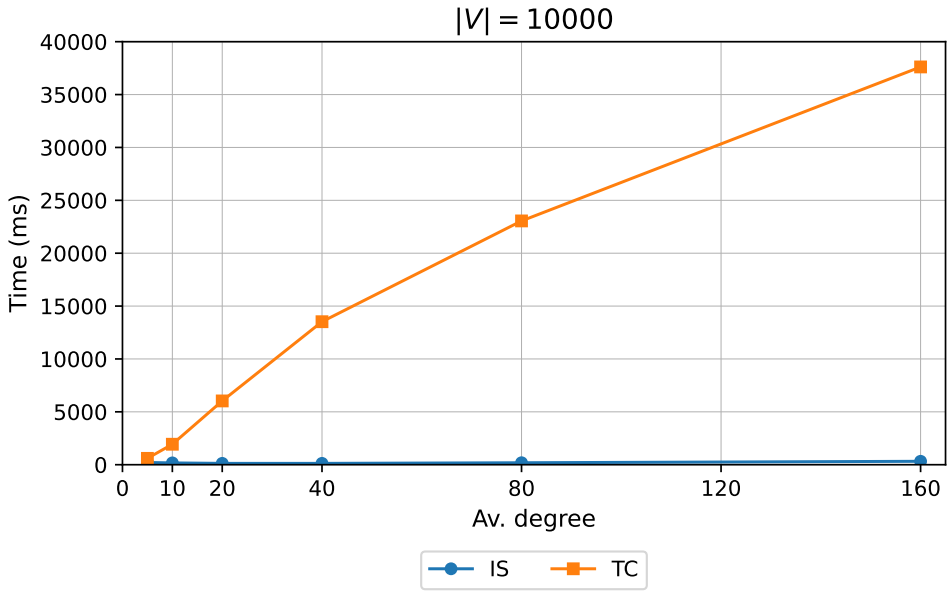


Figure 11: Run time comparison between the Indexing Scheme (blue line) and TC (red line) for ER model on graphs of 10000 nodes, see Table 3.

B Indexing Scheme Experiments

In this section, we present experimental results on real-world citation networks. We consider the citation networks hep-ph and hep-th as well as the DBLP citation network [35] and the Cora dataset [44]. All datasets were obtained from the KONECT repository [33] and the Stanford SNAP repository [34]. For each of these graphs, we created a supernode for each Strongly Connected Component (SCC), resulting in a DAG.

Chain decomposition is calculated by Algorithm 2, the sixth column (NH_conc Time) is the run time of Algorithm 2, the seventh column (TC Indexing Scheme) is the run time of Algorithm 5 given the chain decomposition. The column Width represents the optimum number of chains. The last column is the run time of the DFS-based algorithm for computing the transitive closure (TC Adj. Matrix) where it maintains a two dimensional matrix and runs in $O(|V| * |E|)$. The run times are measured in milliseconds.

We run these experiments on a laptop with an AMD Ryzen 9 HX PRO CPU and 64 GB of main memory. Notice that the results are similar to those of Section 4.4.

Graph	$\|V\|$	$\|E\|$	Number of Chains	Width	NH_conc Time (ms)	TC Indexing Scheme Time (ms)	TC Adj. Matrix Time (ms)
hep-ph	21608	116805	7354	7236	61	244	5787
hep-th	20085	130469	5551	5495	10	195	5745
DBLP	12285	43868	9440	9440	3	140	304
cora	18061	44214	10032	10006	8	310	677

Table 5: Experimental results on real-world citation graphs.