Fast Reachability Using DAG Decomposition

Giorgos Kritikakis
Univeristy of Crete, Heraklion, Greece

Ioannis G. Tollis
Univeristy of Crete, Heraklion, Greece

Abstract

We present a fast and practical algorithm to compute the transitive closure (TC) of a directed graph. It is based on computing a reachability indexing scheme of a *directed acyclic graph* (DAG), $G = (V, E)$. Given any path/chain decomposition of G we show how to compute in parameterized linear time such a reachability scheme that can answer reachability queries in constant time. The experimental results reveal that our method is significantly faster in practice than the theoretical bounds imply, indicating that path/chain decomposition algorithms can be applied to obtain fast and practical solutions to the transitive closure (TC) problem. Furthermore, we show that the number of non-transitive edges of a DAG G is $\leq width * |V|$ and that we can find a substantially large subset of the transitive edges of G in linear time using a path/chain decomposition. Our extensive experimental results show the interplay between these concepts in various models of DAGs.

2012 ACM Subject Classification Theory of computation → Theory and algorithms for application domains; Theory of computation → Design and analysis of algorithms

Keywords and phrases graph algorithms, hierarchy, directed acyclic graphs (DAG), path/chain decomposition, transitive closure, transitive reduction, reachability, reachability indexing scheme

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.2

Related Version *Full Version*: `https://arxiv.org/abs/2212.03945`

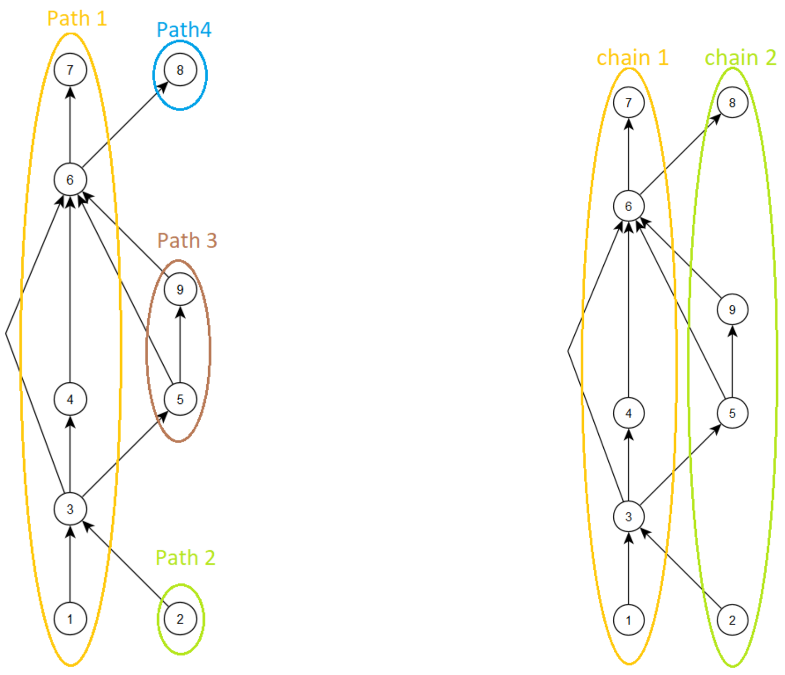
Supplementary Material *Software (Source Code)*: `https://github.com/GiorgosKritikakis/OnGraphHierarchies`

1 Introduction

The problem of computing reachability information or a transitive closure of a directed graph is fundamental in computer science and has a wealth of applications. Formally, given a directed graph $G = (V, E)$, the transitive closure of G, denoted as G^*, is a graph (V, E^*) such that E^* contains all 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 with cycles, we can find the strongly connected components (SCC) in linear time and collapse all vertices of a SCC into a supernode. Hence, any reachability query can be reduced to a query in the resulting *Directed Acyclic Graph* (DAG). 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 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.

The *width* of a DAG $G = (V, E)$ is the maximum number of mutually unreachable vertices of G [8]. An *optimum chain decomposition* of a DAG G contains the minimum number of chains, k, which is equal to the width of G. In Section 2 we present experimental results that show the behavior of the width of DAGs as they become larger and/or denser. Due to the multitude of applications there are several algorithms to find a chain decomposition of a DAG, see for example [16, 9, 7, 22, 4, 5, 18, 26]. 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. On the other hand, for several practical applications it is not necessary to compute an optimum chain decomposition.

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 can be computed in $O(nm)$ time by starting a breadth-first or depth-first search from each vertex. Alternatively, one can use the Floyd-Warshall algorithm [12] which runs in $O(n^3)$, or solutions based on matrix multiplication [24]. Currently, the best known bound on the asymptotic complexity of a matrix multiplication algorithm $O(n^{2.3728596})$ time [2]. An algorithm with complexity $O(n^{2.37188})$ was very recently announced in a preprint [10]. However, this and similar improvements to Strassen's Algorithm are not used in practice because the constant coefficient hidden by the notation are extremely large. Here we focus on computing a reachability indexing scheme in almost 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 $O(n^2)$ time (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 a given path/chain decomposition (i.e., the DAG and a path/chain decomposition are given as input to the algorithm). The scheme can be computed in parameterized linear time, where the parameter is the number, k_c, of paths/chains in the given decomposition. The scheme 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 TC in linear time. Consequently, several hybrid reachability algorithms will run much faster in practice. The time complexity to produce the scheme is $O(|E_{tr}| + k_c * |E_{red}|)$, 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. Clearly, given the reachability indexing scheme the TC matrix can be computed in $O(|V|^2)$ time.

2 Width of a DAG and Decomposition into Paths/Chains

In this section, we briefly describe some categories of path and chain decomposition techniques and show experimental results for 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 [16]. 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. In addition to path-decomposition algorithm categorization, Jagadish in [16] describes chain decomposition heuristics. Those heuristics run in $O(n^2)$ time using a pre-computed transitive closure, which is not linear, and we will not discuss them further.

In [19], a chain decomposition technique was introduced that runs 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. This approach relies on path concatenation. We can concatenate two paths/chains into a single chain if there is a path between the last vertex of one chain and the first vertex of another chain. This algorithm produces decompositions that are very close to the optimum, and its worst-case time complexity is the same as the algorithms that construct simple path decomposition. The above techniques have been tested in practice, and we can utilize any of these approaches to build a chain decomposition in linear or almost linear time, see [19]. In the next sections, we describe how fast chain decomposition algorithms can enhance transitive closure solutions, and present in detail an indexing scheme.

In the rest of this section, we present results that reveal the behavior of the width as the graph density increases. We use three different random graph models implemented in networkx : Erdős-Rényi [11], Barabasi-Albert [3], and Watts-Strogatz [28] models. The generated graphs are made acyclic, by orienting all edges from low to high ID number, see the documentation of networkx [14] for more information about the generators. 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 in order to have results for various sizes and densities. 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 and were run on multiple copies of graphs. We observed that the graphs generated by the same generator with the same parameters

Table 1 The width of the graph in three different networkx models as the density increases for graphs of 5000 nodes.

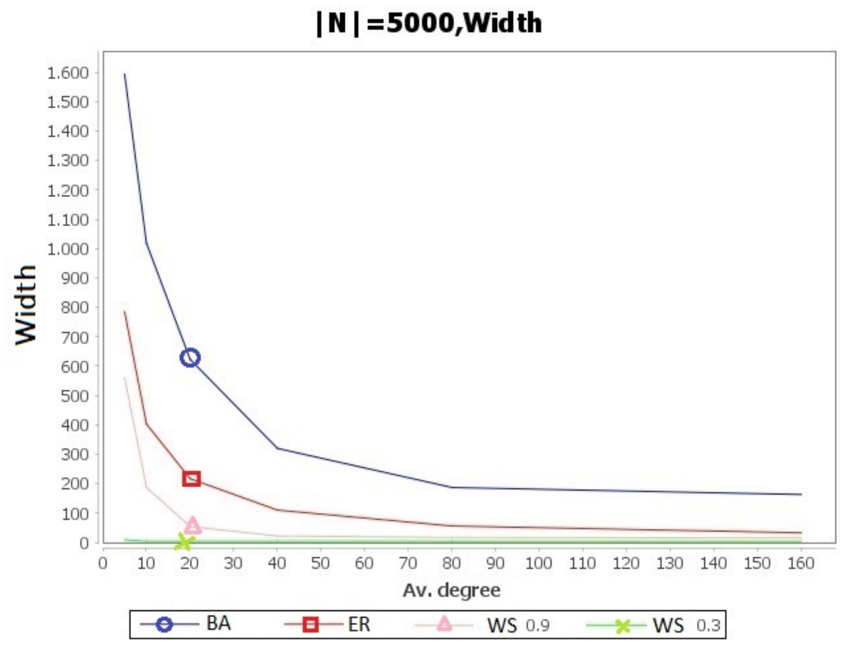
$\lvert V \rvert = 5000$						
Av. Degree	5	10	20	40	80	160
	BA					
Width	1593	1018	623	320	187	163
	ER					
Width	785	403	217	110	56	33
	WS, b=0.9					
Width	560	187	54	22	17	15
	WS, b=0.3					
Width	9	4	4	4	4	4

have small width deviation. For example, the percentage of deviation on ER is about 5% and for the BA model is less than 10%. The width deviation of the graphs in the WS model is a bit higher, but this is expected since the width of these graphs is significantly smaller. The aim of our experiments is to understand the behavior of the width of DAGs created in different models. Tables 1 and 2 show the width (computed by Fulkerson's method) for graphs of 5000 nodes and 10000 nodes, respectively.

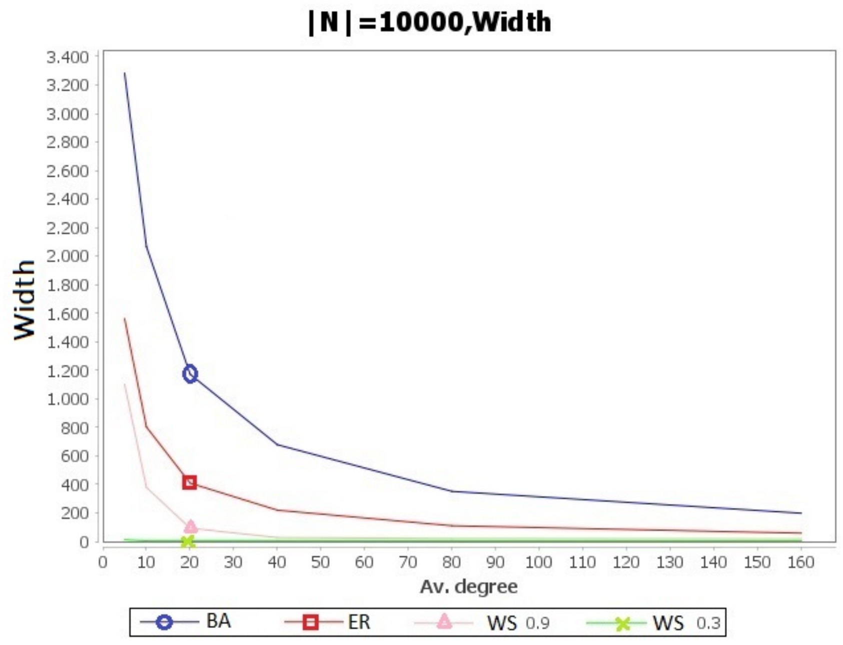
Random Graph Generators.

- **Erdős-Rényi (ER) model** [11]: 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 to the generator.
- **Watts–Strogatz (WS) model** [28]: 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.

Understanding the width in DAGS. In order to understand the behavior of the width of DAGs of these random graph 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 $b = 0.9$ and the second has $b = 0.3$. 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. That is the reason we choose a low and a high probability. Figures 2a and 2b, are derived from Tables 1 and 2, and 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 for WS with $b = 0.3$) the width of a DAG decreases fast as the density of the DAG increases. As a matter of fact, it is interesting to observe that the width of the ER model graphs is proportional to $\frac{\text{Number of nodes}}{\text{average degree}}$. The width of the BA model graphs is clearly higher, but it follows a similar trend.



(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.

Table 2 The width of the graph in three different networkx models as the density increases on graphs of 10000 nodes.

$\|V\| = 10000$						
Av. Degree	5	10	20	40	80	160
	BA					
Width	3282	2066	1172	678	351	198
	ER					
Width	1561	802	409	219	110	58
	WS, b=0.9					
Width	1101	378	93	27	20	18
	WS, b=0.3					
Width	12	4	4	4	4	4

3 DAG Reduction for Faster Transitivity

The importance of removing transitive edges in order to create an abstract graph utilizing paths and chains was first described in [20]. Their focus was on graph visualization techniques, while in this paper we apply a similar abstraction to solve the transitive closure problem. This concept of abstraction or reduction of a DAG may be useful in several applications beyond transitive closure or reachability. Therefore we state the following useful lemmas and Theorem 3:

▶ **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 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 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 3a. ◀

▶ **Lemma 2.** *Given a chain decomposition D of a DAG $G = (V, E)$, each vertex $v_i \in V$, $0 \leq i < |V|$, can have at most one incoming non-transitive edge per chain.*

Proof. Similar to the proof of Lemma 1, see Figure 3b. ◀

▶ **Theorem 3.** *Let $G = (V, E)$ be a DAG with width w. The non-transitive edges of G are less than or equal to width $* |V|$, in other words $|E_{red}| = |E| - |E_{tr}| \leq width * |V|$.*

Proof. Given any DAG G and its width w, there is a chain decomposition of G with w number of chains. 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 $width * |V|$. ◀

An interesting application of the above is that we can find a significantly large subset of E_{tr} in linear time as follows: 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. In this fashion we find a superset of E_{red}, call it E'_{red}, in

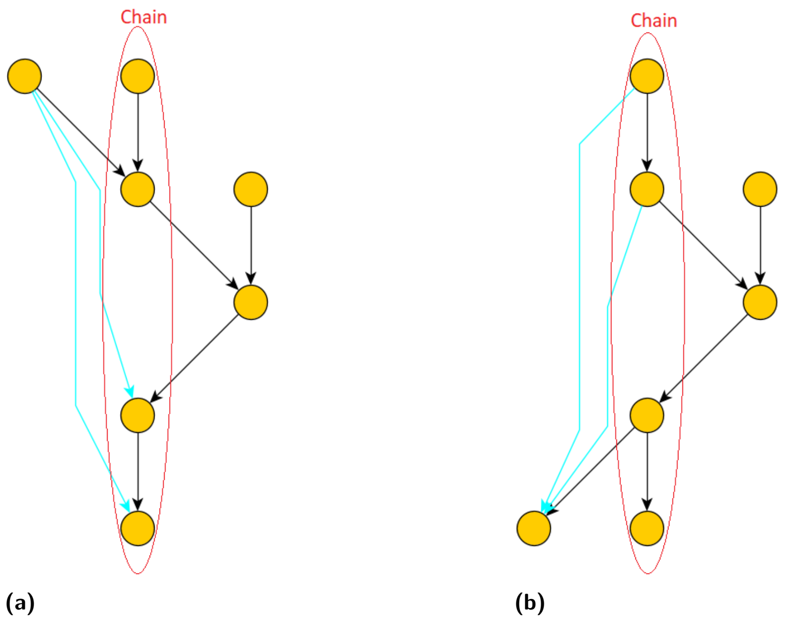


Figure 3 The light blue edges are transitive. (a) shows the outgoing transitive edges that end in the same chain. (b) shows the incoming transitive edges that start from the same chain.

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, 6, 25, 27], 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 dynamic insertion/deletion of edges.

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 [16] 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. Thus his scheme does not answer queries in constant time. Simon [23], describes a technique similar to [16]. His technique is based on computing a path decomposition, thus boosting the method presented in [13]. The linear time heuristic used by Simon is similar to the Chain Order Heuristic of [16]. A different approach is a graph structure referred to as path-tree cover introduced in [17], 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 [23] 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 the chain decomposition approach of [19], which produces smaller decompositions than previous heuristic techniques, without any considerable run-time overhead. Additionally, this heuristic, called NH_conc, will perform better than any path decomposition algorithms as will be explained next. Thus the indexing scheme is more efficient both in terms of 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 3 and 4 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 a 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 to 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.

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 of indexes. A small example is depicted in Figure 4. 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 4 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.

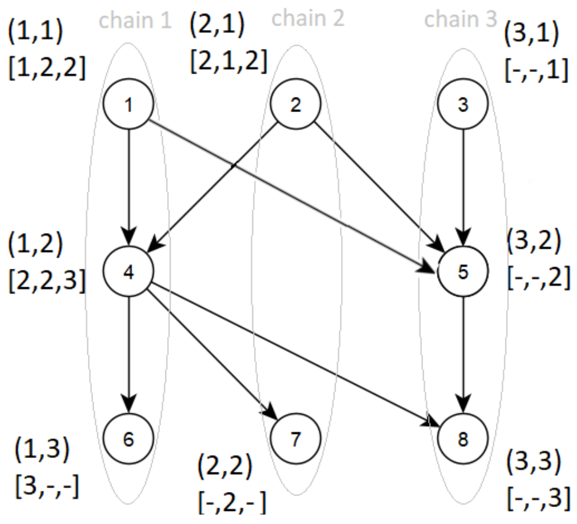


Figure 4 An example of an indexing scheme.

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 users.

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 Adjacency lists

Next, we use a linear time algorithm to sort all the adjacency lists of immediate successors in ascending topological order. See Algorithm 2 in Appendix A.2. 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 $O(k_c * |E_{red}|)$.

4.3 Creating the Indexing Scheme

Now we present Algorithm 1 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 5. 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

Algorithm 1 Indexing Scheme.

```
1: procedure CREATE INDEXING SCHEME($G, T, D$)
   INPUT: A DAG $G = (V, E)$, a topological sorting T of G, and the decomposition D of G.
2:     for each vertex: $v_i \in G$ do
3:         $v_i$.indexes $\leftarrow$ new table[size of D]
4:         $v_i$.indexes.fill($\infty$)
5:         $ch\_no \leftarrow v_i$'s chain index
6:         $pos \leftarrow v_i$'s chain position
7:         $v_i$.indexes[ $ch\_no$ ] $\leftarrow pos$
8:     end for
9:     for each vertex $v_i$ in reverse topological order do
10:        for each adjacent target vertex $t$ of $v_i$ in ascending topological order do
11:            $t\_ch \leftarrow$ chain index of $t$
12:            $t\_pos \leftarrow$ 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
```

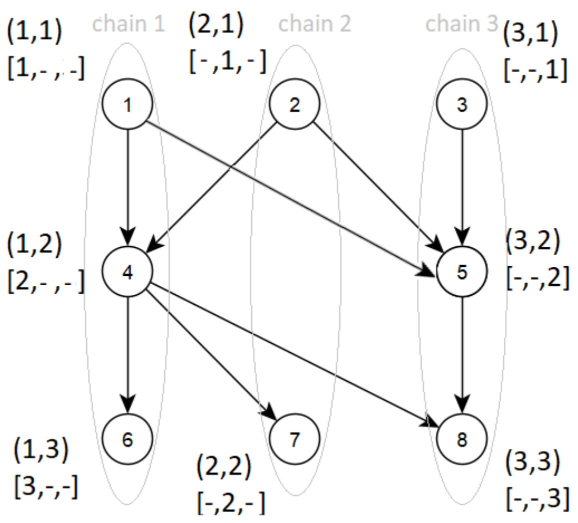


Figure 5 Initialization of indexes.

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), $0 \leq i < kc$, 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 while-loop of Algorithm 1 (lines 10-17) calculates the indexes of v by updating its array with its non-transitive outgoing edges' successors. (Proof in Appendix A.1).*

Combining the previous algorithms and results we conclude this section with the following:

▶ **Theorem 5.** *Let $G = (V, E)$ be a DAG. Algorithm 1 computes an indexing scheme for G in $O(|E_{tr}| + k_c * |E_{red}|)$ time. (Proof in Appendix A.1).*

As described in the introduction, a parameterized linear-time algorithm for computing the minimum number of chains was recently presented in [5]. 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 1 it computes an indexing scheme for G in parameterized linear time. This implies that the transitive closure of G can be computed in parameterized linear time. Hence we have the following:

▶ **Corollary 6.** *Let $G = (V, E)$ be a DAG. Algorithm 1 can be used to compute an indexing scheme for G in parameterized linear time. Hence the transitive closure of G can be computed in parameterized linear time.*

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 [14] and the Path-Based model of [21]. We computed a chain decomposition using the algorithm introduced in [19], called NH_conc, and created an indexing scheme using Algorithm 1. For simplicity, we assume that the adjacency lists of the input graph are sorted, using Algorithm 2, as a preprocessing step. We report our experimental results in Tables 3 and 4 for graphs with 5000 nodes and graphs with 10000 nodes, respectively.

In theory, the phase of the indexing scheme creation needs $O(|E_{tr}| + k_c * |E_{red}|)$ 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 decrease. 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 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 6 (see Appendix A). The flat (blue line) represents the run time to compute the indexing scheme, and the curve (red line) the run time of the DFS-based algorithm for computing the transitive closure (TC). Clearly, the time of the DFS-based algorithm increases as the average degree increases, while the time of the indexing scheme is a straight line almost parallel to the x-axis. All models of Tables 3 and 4 follow this pattern.

Table 3 Experimental results for the indexing scheme for graphs of 5000 nodes.

$\lvert V\rvert = 5000$								
Average Degree	Number of Chains	$\lvert E_{tr}\rvert$	$\lvert E_{red}\rvert$	$\lvert E_{tr}\rvert/\lvert E\rvert$	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 4 Experimental results for the indexing scheme for graphs of 10000 nodes.

$\lvert V\rvert = 10000$								
Average Degree	Number of Chains	$\lvert E_{tr}\rvert$	$\lvert E_{red}\rvert$	$\lvert E_{tr}\rvert / \lvert E\rvert$	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

Apparently, there is a trade-off to consider when building an indexing scheme deploying the technique of [19]. 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, then 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 needed index size for every vertex by one. Hence, instead of computing a simple path decomposition (in linear time) the use of a path concatenation procedure in order to create a more compact indexing scheme faster is preferred for almost all applications. Another interesting and to some extent surprising observation that comes from the results of Tables 3 and 4 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.

5 Conclusions and Extensions

Our extensive experiments expose the practical behavior of (1) the *width*, (2) E_{red}, and (3) E_{tr} as the density and size of graphs grow. Furthermore, we show that the set E_{red} is bounded by $width * |V|$ and show how to find a substantially large subset of E_{tr} in linear time given any path/chain decomposition. These facts have important practical implications to the reachability problem and show the potential applications of these techniques in a dynamic setting where edges and nodes are inserted and deleted from a (very large) graph. Although our techniques were not developed for the dynamic case, the picture that emerges is very interesting.

According to our experimental results, see Tables 3 and 4, 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 in order 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 have changed, 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, see [15] for a very recent comparison of practical fully dynamic transitive closure techniques. 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 Rakesh Agrawal, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD Record*, 18(2):253–262, 1989.

2 Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021.

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

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

5 Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru 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.

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

7 Yangjun Chen and Yibin Chen. On the dag decomposition. *British Journal of Mathematics and Computer Science*, 2014. 10(6): 1-27, 2015, Article no.BJMCS.19380, ISSN: 2231-0851. URL: `https://www.researchgate.net/publication/285591312_Pre-Publication_Draft_2015_BJMCS_19380`.

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

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

10 Ran Duan, Hongxun Wu, and Renfei Zhou. Faster matrix multiplication via asymmetric hashing, 2023. `arXiv:2210.10173`.

11 P Erdös and A Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 1959.

12 Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

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

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

15 Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Faster fully dynamic transitive closure in practice. *CoRR*, abs/2002.00813, 2020. `arXiv:2002.00813`.

16 H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, December 1990. `doi:10.1145/99935.99944`.

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

18 Shimon Kogan and Merav Parter. Beating matrix multiplication for n^{1/3}-directed shortcuts. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

19 Giorgos Kritikakis and Ioannis G Tollis. Fast and practical dag decomposition with reachability applications. *arXiv e-prints*, 2022. `arXiv:2212.03945`.

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

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

22 Veli Mäkinen, Alexandru I Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. Sparse dynamic programming on dags with small width. *ACM Transactions on Algorithms (TALG)*, 15(2):1–21, 2019.

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

24 Volker Strassen et al. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

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

26 Jan Van Den Brand, Yin Tat Lee, Yang P Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and ℓ1-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.

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

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

A Appendix

A.1 Proofs

Proof of Lemma 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 1 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. ◀

Proof of Theorem 5. In the initialization step, the indexes of all sink vertices have been computed as we described above. Taking vertices in reverse topological order, the first vertex we meet is a sink vertex. When the for-loop of line 9 visits the first non-sink vertex, the indexes of its successors are computed (all its successors are sink vertices). According to Lemma 5.1, 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 5.1, 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. ◀

Algorithm 2 Sorting Adjacency lists.

```
1: procedure $\textsc{Sort}(G, t)$
   INPUT: A DAG $G = (V, E)$
2:     for each vertex: $v_i \in G$ 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
```

A.2 Sorting Adjacency lists Algorithm

▶ **Lemma 7.** *Algorithm 2 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. ◀

A.3 Figures

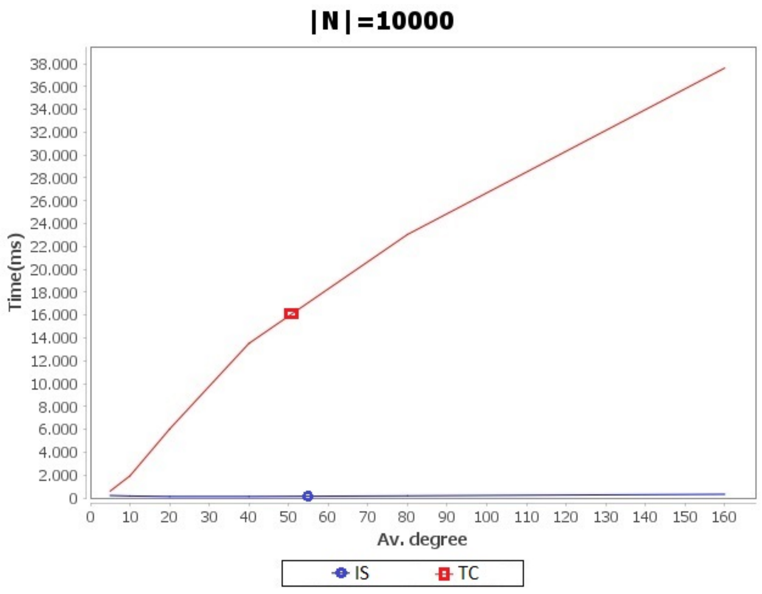


Figure 6 Run time comparison between the Indexing Scheme (blue line) and TC (red line) for ER model on graphs of 10000 nodes, see Table 4.