An experimental study on transitive closure representations

Esko Nuutila

Laboratory of Information Processing Science
Helsinki University of Technology
Otakaari 1, FIN-02150 Espoo, Finland

E-mail: `enu@cs.hut.fi`
Telephone: +358 9 451 3339
Telefax: +358 9 451 3293

Abstract

We present two new compact transitive closure representations. The first uses intervals and the second chains to store the closure. Both representations are based on previous methods designed for acyclic graphs. The new representations are applicable to all kinds of graphs, and can be efficiently constructed during a single traversal of the input graph. We compared experimentally the average size of these representations and traditional list based representations. The inputs were random graphs. The interval representation outperformed the other representations: it typically required a space at most linear to the number of vertices of the input graph. The chain representation did not save much space compared to a list representation. We also studied the complexity of constructing the interval representation. Our results indicate that in the models of random graphs that we used, the transitive closure can typically be computed in a time linear to the size of the input graph when the interval representation is used. A conclusion is that in a disk environment, the bottleneck of transitive closure computation is typically the traversal of the input, not the construction of the output. Our experimental method is based on a sound statistical approach and it is applicable to other experimental studies of algorithms as well.

Keywords — Transitive closure, directed graphs, algorithms, data structures, intervals, chain decomposition, experimental algorithm study.

TKO-B134
ISBN 951-22-3385-1
ISSN 1239-6893

TKK OFFSET 1996

1 Introduction

We study the effect of representation on the full transitive closure problem in large directed graphs. We present two new compact transitive closure representations and compare them experimentally with traditional representations. Our experimental method is based on sound statistical approach and it can be used in other experimental studies as well. Thus, the paper should be useful for two kinds of readers, those interested in transitive closure computation and those interested in experimental algorithm study.

Transitive closure computation is required in many applications, for instance, in the reachability analysis of transition networks representing distributed and parallel systems [1] and in the construction of parsing automata in compiler construction [2, 3]. Recently, efficient transitive closure computation has been recognized as a significant subproblem in evaluating recursive database queries [4].

The transitive closure often contains $O(n^2)$ edges even when the input graph contains only $O(n)$ edges. If the closure is represented using common set representations such as lists or bit vectors, merely listing the output takes more than a time linear to the size of the input. A solution to this problem is to use a specialized representation that is based on the topology of the transitive closure or the input graph. The representation should typically need a memory space at most linear to the size of the input. We present two specialized representations and study how well they satisfy this requirement.

The first representation is based on intervals of integers. The strong components of the input graph are numbered in a reverse topological order and a set of these numbers is stored as a collection of consecutive intervals. Although this representation takes $O(n^2)$ space in the worst case, it typically is compact; the transitive closure takes much less space than the input graph. Further, the representation can be computed efficiently during a single depth-first traversal of the input graph, and it can be used with efficient transitive closure algorithms like [5, 6, 7, 8, 9] that are based on strong component detection. We developed the representation from a method for compressing the transitive closure of an acyclic graph by Agrawal *et al.* [10].

The second representation is based on a chain decomposition of the condensation graph induced by the input graph. A chain decomposition is a partition of the set of vertices into subsets each representing a path in the graph. Each chain is topologically ordered, and a successor set of the transitive closure is represented as a collection of topologically smallest elements of chains present in the successor set. As the interval representation, the chain representation can be computed during a single depth-first traversal of the input graph. We developed the chain representation from the method for computing the transitive closure of an acyclic graph by Simon [11].

To compare the different representations, we studied their average-case behavior, since the results of a worst-case analysis are often overly pessimistic and even misleading. For instance, many transitive closure algorithms have approximately the same worst-case execution time, but behave differently with typical inputs. Unfortunately, the mathematical average-case analysis is much more difficult than the worst-case analysis. Only a couple of previous articles on transitive closure computation contain an average-case analysis [12, 13, 14, 11], and the approach used in these articles cannot be generalized to other methods for computing the transitive closure.

Our choice, therefore, was to study the average-case performance experimentally. A benefit of this choice was that we could mechanically apply the same technique for all representations, input graph models, and performance metrics. To introduce a new input graph model, we only had to implement a new input graph generator; to introduce a new performance metric, we only had to insert code for collecting the metric values. Another benefit was that the results were more accurate than a mathematical analysis could yield. A weakness in the experimental method is that implementing and testing the programs and running the experiments takes much time. Another weakness is that experiments do not give information on the asymptotic behavior. We only get information on the behavior in that part of the space of possible inputs that our experiments cover.

We confirmed the accuracy of the results by using sound statistical techniques in running the experiments and in analyzing the output metrics. We used *sequential procedures* [15] that automatically yield the desired statistical confidence. Thus, our experimental approach is related to the recent interest in developing an empirical science of algorithms that is based on rigorous methods of statistics and experimental design [16]. Ganguly *et al.* [17] used a similar approach in their study of transitive closure computation.

In the experiments, we compared the sizes of the interval and chain representations and two list representations. We also studied the complexity of constructing the interval representation when different transitive closure algorithms were used. Random graphs of two models $G(n, p)$ and $G(n, p, l)$ were used as inputs.

The experiments showed that the interval representation outperforms the other representations. The chain representation was clearly inferior to the interval representation and only slightly better than the list representation. The average space requirement for the interval representation was usually at most linear to the number of vertices in the input graph. The exception was model $G(n, p)$ when the expected outdegree np was slightly greater than one; then, the size seemed to be about $0.79 n \ln n$ words. Similarly, the average time requirement for constructing the interval representation was usually linear to the size of the input graph when algorithm COMP_TC [8, 18] or CR_TC [18] was used. The exception was model $G(n,p)$ when the expected outdegree np was slightly greater than one; then, the construction time seemed to be slightly less than quadratic to the number of vertices of the input graph.

Our results indicate that in models $G(n,p)$ and $G(n,p,l)$, the transitive closure can typically be computed in a time linear to the size of the input graph. Since the interval representation usually takes much less space than the input graph, the bottleneck in transitive closure computation in a disk environment is the traversal of the input graph, not the construction of the output.

The structure of the rest of the paper is the following. In section 2, we present an overview of the transitive closure problem and some important transitive closure algorithms. In section 3, we present the interval and the chain representations. In section 4, we describe our experimental method and the experiments comparing the sizes of the representations and the complexity of computing the interval representation. In section 5, we summarize the results and give the conclusions.

2 Transitive closure computation

We consider a directed graph $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. If $(v, w) \in E$, vertex v is *adjacent to* vertex w and w is *adjacent from* v. The *transitive closure* of G is a graph $G^+ = (V, E^+)$ such that for all $v, w \in V$ there is an edge $(v, w) \in E^+$ if and only if there is a non-null path from v to w in G. The *successor set* of a vertex v is the set $Succ(v) = \{w \mid (v, w) \in E^+\}$. A *strong component* of G is a maximal subset $C \subseteq V$ such that for each $v, w \in C$ there is a path from v to w (and vice versa). The *condensation graph induced by the strong components of graph* G is a graph $\overline{G} = (\overline{V}, \overline{E})$ such that $\overline{V}$ is the set of the strong components of G and $\overline{E}$ contains an edge (X, Y) if and only if E contains an edge (u, v) such that u and v are in components X and Y, respectively. A *topological order* of the vertex set V of a graph $G = (V, E)$ is any total order $\leq$ of V such that $v \leq w$ if edge $(v, w) \in E$. If $v \leq w$, we say that v is *topologically smaller* than w and w is *topologically greater* than v. The reverse relation of a topological order $\leq$ is a *reverse topological order*.

Several variants of the transitive closure problem exist. In the *full transitive closure problem*, we should find all pairs of vertices in the input graph that are connected via non-null paths. In the *single-source transitive closure problem*, we should find all vertices that are reachable from a given vertex via non-null paths. In the *multi-source transitive closure problem*, we should find the vertices that are reachable from a given set of vertices via non-null paths. We consider here the full transitive closure problem.

Many transitive closure algorithms have been presented in the literature. We discuss here only some of them; for surveys of transitive closure computation, see [18, 19].

Warshall's algorithm [20] is probably the best-known transitive closure algorithm. The algorithm transforms an adjacency matrix representing the input graph into an adjacency matrix representing the transitive closure by scanning the potential paths of the input graph in three nested loops. Thus, the algorithm is short and simple to implement, but its best-case and worst-case execution times are $O(n^2)$ and $O(n^3)$, respectively, which make it inferior to some newer algorithms.

Transitive closure computation is closely related to matrix multiplication. Munro [21] and Fischer and Meyer [22] presented transitive closure algorithms that use matrix multiplication and that have the same worst-case execution time as a single matrix multiplication. These are the asymptotically fastest transitive closure algorithms when the input graph is dense, i.e., contains near n^2 edges. The fastest known matrix multiplication algorithm [23] has the worst-case bound $n^{2.376}$, but is not practical.

A major difficulty in transitive closure computation is the avoidance of redundant operations. A graph may contain pairs of vertices that are connected via multiple paths. Many vertices may have the same successor set. Figure 1 illustrates these redundancies. Two paths lead from vertex a to vertex f and all its successors. Vertices d, e, f, g, h, and i have the same successor set.

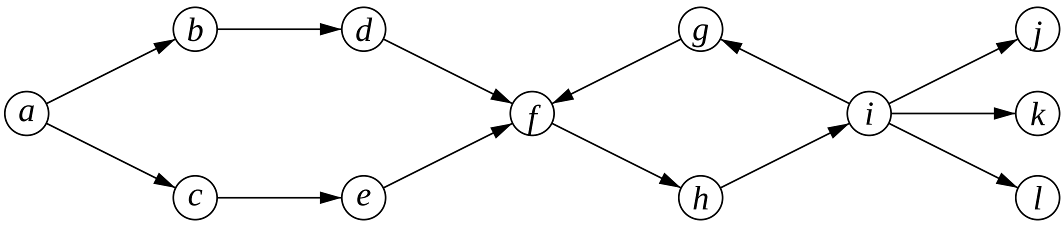


FIGURE 1: A graph with vertices having the same successor sets and with multiple paths between vertices.

The strong components cause most redundant operations in many transitive closure algorithms [17], since all vertices of a strong component have the same successor set. A class of practical and efficient transitive closure algorithms is based on strong component detection. Purdom [24] and Munro [21] presented the first of these algorithms. Their algorithms detect the strong components, build the condensation graph, compute the transitive closure of the condensation graph, and finally compute the transitive closure of the input graph from the transitive closure of the condensation graph. Later, Eve and Kurki-Suonio [6], Ebert [5], Schmitz [9], Ioannidis *et al.* [7], and Nuutila and Soisalon-Soininen [8, 18, 25, 26] presented transitive closure algorithms that use the strong components without generating the condensation graph. All these algorithms are based on Tarjan's strong component algorithm [27].

In [18], we showed that algorithm COMP_TC [8] has the smallest worst-case execution time of these algorithms, namely $O(n\overline{e}_r + \min(ns, e' \log n))$, where $\overline{e}_r$ is the number of edges in the transitive reduction of the condensation graph, s is the number of strong components, and e' is the size of a certain subset of edges. This worst-case bound is also smaller than that of Purdom's algorithm. The term $\min(ns, e' \log n)$ is usually negligible to $n\overline{e}_r$. The algorithm by Goralcikova and Koubek [28] has a worst-case bound $O(n\overline{e}_r)$, but it does several passes over the input graph making the algorithm impractical. Algorithm COMP_TC uses only a single traversal. We also presented a large collection of experiments for comparing the average-case performance of the transitive closure algorithms that are based on strong component detection. Algorithm COMP_TC was more efficient than the other algorithms in all these experiments.

Recently, several transitive closure algorithms have been presented for databases, see [29, 18] for surveys of these algorithms. Some work concentrates on transitive closure computation in parallel or distributed environments, see, for instance, [30, 31, 32, 33].

3 Transitive closure representations

The successor sets of a transitive closure are usually implemented using common set representations such as lists, bit matrices, bit vectors, or binary trees, or hybrid representations, for example, lists augmented with bit vectors to speed up membership tests. In a database environment, the closure is usually represented as a relation table.

A weakness in common set representations is that they use no structural properties of the transitive closure to reduce the memory requirements; a transitive closure $G^+ = (V, E^+)$ takes at least $O(|V| + |E^+|)$ space. We need a specially designed representation. Besides the small memory requirements, it should have also other properties: doing the set operations and querying the results of the computation should be efficient. We describe next two such representations. The first is based on intervals of strong component numbers and the second on a chain decomposition of the input graph.

3.1 Interval representation

The interval representation is a new successor set representation that can be computed during a single depth-first traversal of the input graph. Contrary to the common set representations, the interval representation of a successor set S often requires a space smaller than $O(|S|)$, and the union of two sets may be smaller than either of its arguments. The representation of a transitive closure $G^+ = (V, E^+)$ usually requires a space much smaller than $O(|E^+|)$, and the best-case execution time of a transitive closure algorithm can be less than $O(|E^+|)$. The representation is based on a method by Agrawal *et al.* [10] for compressing the transitive closure of an acyclic graph.

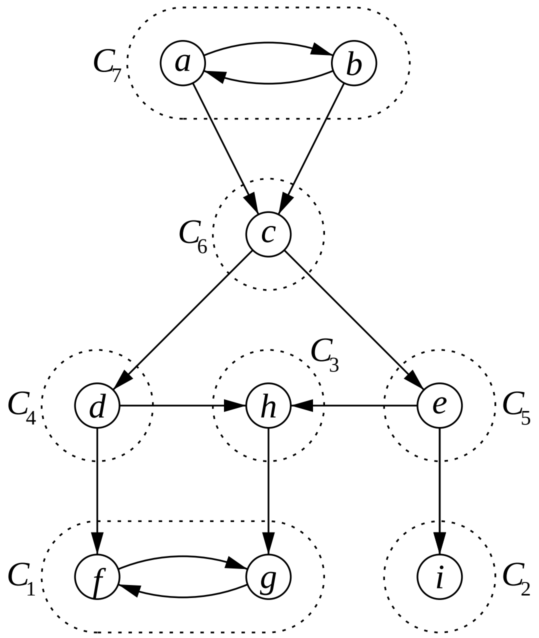
The representation consists of two parts: a method for storing sets of integers compactly and a method for numbering the strong components. The method for storing sets of integers is the following. Consider a set $S \subseteq \{1, 2, \ldots, n\}$. If S contains sequences of consecutive integers $i, i+1, \ldots, j$, we represent S by storing only the pair of endpoints $[i, j]$ of each sequence. We call such a pair an *interval*. When new intervals are added to a set, overlapping intervals are merged together. Also, two intervals $[i, j]$ and $[j+1, k]$ are combined into an interval $[i, k]$.

For example, a set $S_1 = \{0, 1, 2, 4, 5, 6, 8\}$ is stored as three intervals $[0, 2]$, $[4, 6]$, and $[8, 8]$ and a set $S_2 = \{0, 2, 3, 4, 5, 9, 10\}$ as three intervals $[0, 0]$, $[2, 5]$, and $[9, 10]$. If we insert number 7 to S_1, we get the set $S_1' = \{0, 1, 2, 4, 5, 6, 7, 8\}$, which is stored as two intervals $[0, 2]$ and $[4, 8]$. If we take the union of S_1' and S_2, we get the set $\{0, 1, 2, \ldots, 10\}$, which is stored as a single interval $[0, 10]$.

The maximum number of intervals needed to store a set $S \subseteq \{1, 2, \ldots, n\}$ is $\lceil n/2 \rceil$. This upper bound is reached when S consists of all odd integers between 1 and n. The expected number of intervals needed is harder to compute, since it depends on the distribution of the integers. If the distribution is uniform, i.e., each element of $\{1, 2, \ldots, n\}$ has the same probability of being drawn, the expected number of intervals needed to store S is $k - k(k-1)/n$, where $|S| = k$ (see [18]). If the integers are likely to be near some value x, the number of intervals needed may be much smaller.

The integers that we store in the intervals are the reverse topological numbers of the strong components. We use Tarjan's algorithm to detect the components in a reverse topological order. Thus, the numbering is a byproduct of the strong component detection, and no extra computations are needed. With this numbering,

(a)



(b)

Component	Successors
C_1	$\{[1,1]\}$
C_2	$\{\}$
C_3	$\{[1,1]\}$
C_4	$\{[1,1],[3,3]\}$
C_5	$\{[1,3]\}$
C_6	$\{[1,5]\}$
C_7	$\{[1,7]\}$

FIGURE 2: **(a)** Graph G with seven strong components. **(b)** The successor sets.

we usually need much fewer intervals than with the uniform distribution.

In Figure 2, we present an example graph and its successor sets as intervals. We assume that Tarjan's algorithm traverses the vertices in order: g, f, i, a, c, d, h, e, b. The strong components are encircled. In this traversal, the maximum number of intervals needed to represent a successor set is 2, and the total number of intervals is 7.

A collection of intervals can be stored in several ways. Since the intervals are distinct, they can be ordered and stored in an ordered binary tree. The binary tree can be balanced, but usually this is not necessary, since the trees are small. Another possibility is to store the intervals in an ordered array. An ordered array requires less space than a binary tree, and the memory allocation is simpler, but the insertion operations may be more expensive than in a balanced tree. Ordered arrays are especially suitable with transitive closure algorithms like algorithm COMP_TC [8] and Schmitz's algorithm [9], in which at most one successor set is under construction at any moment.

The union of two successor sets S_1 and S_2 takes $O(|I_1| + |I_2|)$ time, where I_1 and I_2 are the collections of intervals representing S_1 and S_2. Testing the presence of an element x in set S takes $O(\log(|I|))$ time in the worst case if the intervals are stored in a balanced binary tree or an ordered array and $O(|I|)$ time if the intervals are stored in an unbalanced binary tree. Inserting an element x into set S takes

$O(\log(|I|))$ time in the worst case if the intervals are stored in a balanced binary tree and $O(|I|)$ time if the intervals are stored in an unbalanced binary tree or an ordered array.

The idea of the interval representation came from [10], where Agrawal *et al.* studied how the transitive closure can be precomputed and stored in a compressed form. Their inputs were restricted to acyclic graphs. The stored closure is used to directly answer queries about successors of a vertex without computing the successors on-the-fly. They also showed how to incrementally update the stored closure when the input graph changes. In their method, an optimal spanning tree is computed for the input graph and that spanning tree is traversed in a reverse topological order to construct the sets of intervals. Several passes over the input graph are required, whereas our method requires just one pass. According to our experience, computing the optimal spanning tree is not necessary; our method usually yields an equally small (or smaller) representation by merging the adjacent or overlapping intervals. In their method, this merging is not done.

3.2 Chain representation

The chain representation is another compact way to represent the successor sets. Originally, it was presented for acyclic graphs [11] and it required several passes over the input graph. We show how this representation is computed during a single depth-first traversal of the graph and even when the graph is cyclic.

The chain representation belongs to a class of representations based on partitioning the vertices into a collection of disjoint subsets. Let $\Pi = \{C_i \mid C_i \subseteq V\}$ be a partition of the vertex set V; $\bigcup_{C_i \in \Pi} C_i = V$ and $C_i \cap C_j = \{\}$ when $i \neq j$. Let S be a successor set. We represent S as a collection of intersections, one per each subset in Π, i.e., $S = \bigcup_{C_i \in \Pi} C_i \cap S$. Goralcikova and Koubek [28] described the first such representation; their partitioning was based on topological levels of the acyclic input graph. Unfortunately, no compact way to store the intersections in this partitioning has been presented.

In the chain representation, a subset of the partition represents a path in the acyclic input graph. The vertices in a chain are ordered topologically. An intersection $S \cap C_i$ is represented by the topologically smallest vertex v of the chain C_i that is in the intersection; each vertex in chain C_i that is topologically greater than v is also in the intersection. If k is the number of chains in the partitioning, a successor set S requires $O(\min(k, |S|))$ space.

An acyclic graph can be partitioned into chains in many different ways; the number of chains varies in different partitions. Simon [11] used the topological ordering of the input graph to compute the chains. Jagadish [34] presented several ways to construct the chains. He first showed that a minimal partition can be computed by the minimum flow algorithm. Unfortunately, this takes $O(n^3)$ time, i.e., it is often slower than the transitive closure computation. Jagadish then presented a collection of heuristics that can be used to construct the chains. His experiments indicated that the topological ordering approach by Simon [11] usually yields a smaller partition than the other heuristics.

Our approach to construct the chains is based on strong components; thus, our method is not limited to acyclic input graphs. The strong components are detected in a reverse topological order by Tarjan's algorithm. When a component C is detected, all components adjacent from it are already assigned to a chain. The procedure in Figure 3 finds a chain for component C.

```
procedure ASSIGN_TO_CHAIN(C);
begin
    for each component X adjacent from C in topological order do
        if TOP(CHAIN(X)) = X then begin
            PUSH(C, CHAIN(X));
            return
        end
    ch := create a new chain;
    PUSH(C, ch)
end
```

FIGURE 3: Procedure ASSIGN_TO_CHAIN finds a chain for component C.

The components adjacent from C are processed in a topological order. The chains are stored as stacks. If a component X adjacent from C is the topmost element of its chain, we extend that chain by C. If no such component is found, we cannot attach C to any existing chain, and we create a new chain containing C.

Consider again graph G of Figure 2. In Figure 4(a), we present the two chains produced by one possible traversal of the graph. Also here, we assume that Tarjan's algorithm traverses the vertices in order: g, f, i, a, c, d, h, e, b. In Figure 4(b), we present the successor sets as pairs of chains and components. For instance, pair (Chain $2, C_5$) means that the successor set contains components C_5 and C_2 of chain 2.

When chains are used, two basic methods for storing the successors sets exist. A successor set can be represented as a vector containing one element per each chain or as a collection of pairs (C, v), where C is a chain and v is the topologically

(a)

Chain 1	Chain 2
	C_7
C_4	C_6
C_3	C_5
C_1	C_2

(b)

Component	Successors
C_1	$\{(\text{Chain } 1, C_1)\}$
C_2	$\{\}$
C_3	$\{(\text{Chain } 1, C_1)\}$
C_4	$\{(\text{Chain } 1, C_3)\}$
C_5	$\{(\text{Chain } 1, C_3), (\text{Chain } 2, C_2)\}$
C_6	$\{(\text{Chain } 1, C_4), (\text{Chain } 2, C_5)\}$
C_7	$\{(\text{Chain } 1, C_4), (\text{Chain } 2, C_7)\}$

FIGURE 4: **(a)** The chains of graph G. **(b)** The successor sets.

smallest element of C that is in S. Simon [11] proposed the former and Jagadish [34] the latter method. We use the pair approach, since according to our experience, it usually requires much less space than the vector representation. The first elements of the pairs can be eliminated if we use an auxiliary table mapping the components into chains.

In the vector approach, the union operation always takes $O(k)$ time, and the insertion operation and the membership test take a constant time. In the pair approach, the union operation takes $O(k)$ time in the worst case. The insertion operation and the membership test require $O(k)$ time in the worst case if the pairs are stored in a linear list. This can be reduced to $O(\log k)$ by using a balanced binary tree.

The chain representation provides a good worst-case bound for transitive closure computation [11]. Goralcikova and Koubek [28] showed that the transitive closure can be computed essentially by doing $\overline{e}_r$ unions of successor sets, i.e., one union per each edge in the transitive reduction of the condensation graph induced by the strong components. When the chain representation is used, the worst-case bound for computing the transitive closure is therefore $O(k\overline{e}_r)$. For the other representations discussed above, the corresponding worst-case bound is $O(n\overline{e}_r)$.

4 Experiments

We describe now the experiments that we did to compare the average performance of the representations. We start by discussing the problems in mathematical analysis and describe how the experimental method can be used to study algorithm behavior reliably.

4.1 Problems in mathematical algorithm analysis

Mathematical worst-case analysis is the most common method for studying the performance of algorithms. It is usually easy, but the results may be overly pessimistic; an algorithm may run much faster with typical inputs than in the worst case. Comparing the efficiency of two algorithms only by their worst-case bounds may give misleading conclusions. For instance, many transitive closure algorithms have approximately the same worst-case time, but behave differently with typical inputs.

To get a more realistic picture, we should analyze the average-case performance, but this is much more difficult than worst-case analysis. First, we have to assign each input a probability of occurrence. It is not obvious what these probabilities should be. To make the analysis easier, we may have to choose probabilities that are not realistic. Second, given an algorithm and an input model, computing the average-case execution time or space is often not possible. As a general rule, average-case analysis using realistic input models is beyond the limits of the current analytical methods [16, 35]. The average-case analysis may be easier for probabilistic algorithms, which use the distribution of the input model in their behavior. If the inputs are simple combinatorial structures like binary trees, a divide-and-conquer strategy may be possible.

In analyzing the average-case performance of transitive closure algorithms and representations, we face these problems. The input model is not obvious. Most algorithms and representations are not probabilistic. The inputs are not simple combinatorial structures; in addition to the number of vertices and edges, the topology of the graph and the order, in which the vertices and edges are processed, affect the behavior.

Thus, the average-case performance of transitive closure algorithms has been mathematically analyzed in only a few papers. Bloniarz *et al.* [12] studied the average-case performance of a simple transitive closure algorithm. Schnorr [14] and Karp [13] presented probabilistic transitive closure algorithms and average-case analysis of these algorithms. Simon [11] analyzed the average-case performance of his chain decomposition algorithm, but the algorithm only handles acyclic inputs, which simplifies the analysis. In all these papers, a simple input model was used to make the analysis possible.

4.2 Experimental algorithm study

Using computer experiments is another method for studying the performance of algorithms. This has traditionally been considered as a poor substitute for mathematical analysis, but the problem in most experimental studies is more in the lack of scientific discipline than in the experimental approach itself [16]. For instance, the average-case performance of an algorithm is studied by applying it on a small set of arbitrarily selected inputs, and the mean of the measured values is presented as the true average-case performance metric. This approach gives no information on the accuracy of the results.

We need an experimental method that is based on rigorous tools of statistics and experimental design. Hooker [16] calls this "an empirical science of algorithms." In addition of being applicable when mathematical analysis fails, the experimental method has the benefit that it can be mechanically applied to different input models, algorithms, data representations, and performance metrics. For instance, to introduce a new input model, we only have to implement a new input generator, and to introduce a new performance metric, we only have to insert code for collecting the metric values. The results are often more accurate than in a mathematical analysis.

A weakness in the experimental method is that it usually takes more time than mathematical analysis. We have to implement the programs, design and run the experiments, and analyze the output data. Another problem is that we get no information on the asymptotic behavior: we cannot say that the run time of the algorithm is $O(f(n))$. Thus, the experimental approach and the mathematical approach have different areas of applicability and they both are needed.

4.2.1 Estimation theory

A weakness in many experimental studies is that the random nature of the inputs and outputs is not taken into account; this lowers the credibility of the results. The theory of estimation [36] gives us tools for managing the variation in the inputs and outputs.

We assume that a random sample $(x_1, x_2, \ldots, x_n)$, for instance a series of measurement data, is the realization of independent random variables $X_1, X_2, \ldots, X_n$ with the same distribution. The goal is to find values of properties that describe the distribution. Typically, we want to know μ_x, the expected value of the distribution. An unbiased estimator of μ_x is the sample mean

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{1}$$

We assess the accuracy of the estimate by constructing a *confidence interval*. A $100(1-\alpha)\%$ confidence interval of parameter θ is an interval $[L_1, L_2]$ containing the actual value of θ with probability $(1-\alpha)$. If the random variables are normally distributed, the confidence interval of the expected value is $[\bar{x} - \Delta_x, \bar{x} + \Delta_x]$. Δ_x is computed using Student's t-distribution and the estimator $\hat{\sigma}^2[\bar{x}]$ of the variance as follows:

$$\Delta_x = t_{n-1,1-\alpha/2}\hat{\sigma}[\bar{x}] \tag{2}$$

$$\hat{\sigma}^2[\bar{x}] = \frac{1}{n(n-1)}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{3}$$

This approach can also be used when random variables X_i are not normally distributed, if the variables are independent and have an identical distribution. When the size of the sample increases, the sum of the random variables $X = X_1 + X_2 + \ldots + X_n$ tends to the normal distribution. In practice, the estimator gives good approximation when $n > 30$.

Equations 2 and 3 can be used after the experiments to compute the confidence interval. The problem is that if we fix the number of inputs, i.e., the size of the sample, before the experiments, the accuracy of the result may be too low or we may waste time in processing unnecessary inputs. This problem is solved by using *sequential procedures* [15], in which new inputs are processed until the desired accuracy is reached. We describe the stopping condition of the procedure in absolute or in relative terms. An absolute condition usually states that $|\Delta_x| \leq \delta_{\max}$, i.e., the error of the estimate must be at most $\delta_{\max}$. A relative condition can be, for instance, $|\Delta_x/\bar{x}| \leq \epsilon_{\max}$. In practice, we often use a combination of an absolute and a relative stopping condition.

4.2.2 Input models

In selecting the input model for an experimental study, we face partially the same problems as in mathematical analysis. We must choose the set of possible inputs and assign a probability to each input. A benefit of the experimental method is that we can study many input models using the same approach. We can usually apply more complicated input models than in mathematical analysis, but the generation of inputs must be fast, which is not required in mathematical analysis.

In our experiments on transitive closure representations, we first decided to use the model of random graphs $G(n,p)$ that is widely used in the literature [37, 13].

Definition. The model $G(n,p)$ is a probability space containing all graphs having a vertex set $V = \{1, 2, \ldots, n\}$ and an edge set $E \subseteq V \times V$. Each possible edge of a graph exists with probability p independently of the existence of other edges; thus, the probability of a given graph with e edges is $p^e(1-p)^{n^2-e}$.

Although all possible graphs can be generated in $G(n,p)$, the graphs are usually similar in some respects. Karp [13] showed that when n is large and np (the expected outdegree) is greater than one, it is likely that the graph contains one large component, and all other components are trivial, i.e., contain a single vertex. The percentage of vertices in the large component increases rapidly as the expected outdegree increases. We detected that also when n is small, e.g., $n = 100$, the graph usually consists of one large component and a set of trivial components. If we would have used $G(n, p)$ as our only input model, we would have gained no information on graphs with many nontrivial components. Thus, we decided to also use another model of random graphs $G(n, p, l)$.

Definition. The model $G(n, p, l)$ is a probability space containing all graphs having a vertex set $V = \{1, 2, \ldots, n\}$ and an edge set $E \subseteq \{(i, j) \mid (j - i) \bmod n \leq l\}$. Each possible edge of a graph exists with probability p independently of the existence of other edges; thus, the probability of a given graph with e edges is $p^e(1-p)^{n(2l+1)-e}$.

Parameter l is called the *locality*. We detected that when l was small, the graphs of $G(n, p, l)$ had many nontrivial components. When l increased, $G(n, p, l)$ started to resemble $G(n, p)$. It seems that $G(n, p, l)$ with a small value of l is a good model for studying sparse graphs with many nontrivial components. Ioannidis *et al.* [7] used a similar model in their experiments. A variation of model $G(n, p, l)$ is a model where the vertices are in a grid, but we did not use this model.

4.3 The size of transitive closure representations

4.3.1 Experimental setting

Our goal was to find out which representations need the smallest and the largest space on the average in different input models. We also wanted to find out how the sizes vary with different input parameter values. The representations were

1. Successor lists when each vertex has a separate list containing its successor vertices. We call this representation the $v \times v$ *-representation*, since its size equals the size of the transitive closure relation represented as pairs of vertices.

2. Successor lists when strong components are detected, i.e., there is one list for each component and it contains the successor components. We call this representation the $c \times c$ *-representation*, since its size equals the size of the transitive closure relation of the condensation graph represented as pairs of components.

3. Interval representation.

4. Chain representation.

Bit vector and bit matrix representations were not compared, since they typically require $O(n^2)$ memory space.

The inputs were sparse random graphs of $G(n,p)$ and $G(n,p,l)$. We studied no dense graphs, since the situation is obvious: if the strong components are detected, the space requirement is at most $O(n)$ and otherwise $O(n^2)$. The input parameters were the number of vertices n, the expected outdegree d, and the locality l. The edge probability p was computed from d, n, and l. Parameter n had values 100, 300, 1000, 3000, 10000, 25000, and 50000, d had values ranging from 0.0 to 6.0 by 0.1, and l had values 5, 10, and 20. We could not use vertex counts $n > 10000$ in model $G(n,p)$ with the chain representation, since it required too much memory. This was not a problem in the list representations, since their sizes were computed from the interval representation.

For each combination of n, d, and l we computed a 90% confidence interval of the average size. The relative error was at most 10% or the absolute error at most $n/100$. The combined condition was used, since the sizes varied much in different parts of the input space. The size was counted as the number of memory words required. We assumed that one element of a list takes one word, and an interval takes two words. A pair in the chain representation takes one word, since we use an auxiliary table mapping components into chains.

4.3.2 Results

We present the results using three dimensional plots with the number of vertices and the expected outdegree on the x- and y-axis. On the z-axis, we present the

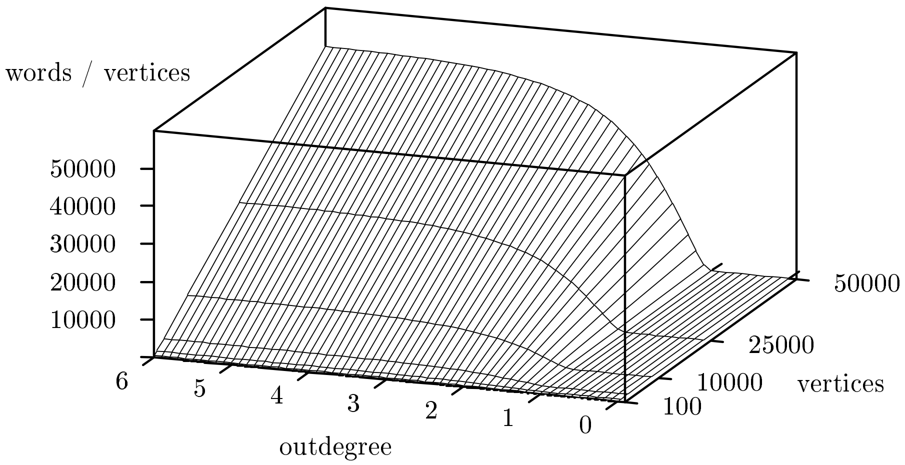


FIGURE 5: The size of the $v \times v$ -representation in model $G(n,p)$.

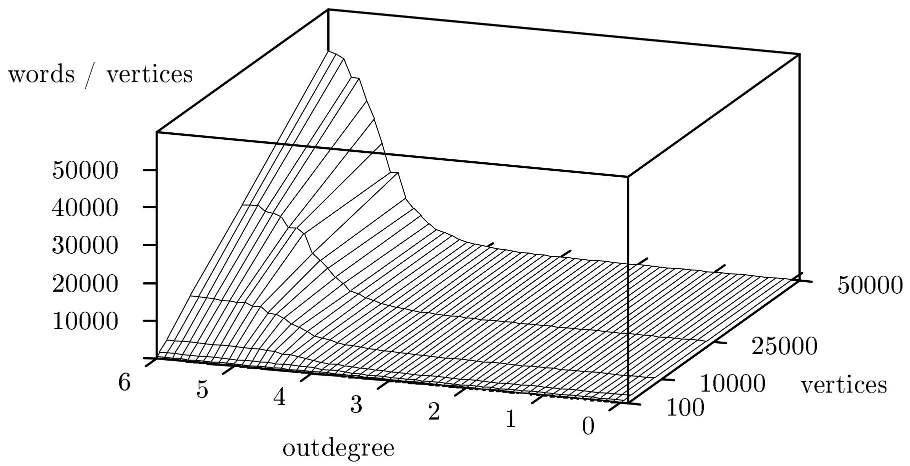


FIGURE 6: The size of the $v \times v$ -representation in model $G(n, p, l = 5)$.

number of required memory words divided by the number of vertices. Note that the scale of the z-axis varies in different plots. For each number of vertices, we also computed the maximum of the average sizes when the expected outdegree was varied. By using regression models, we estimated the dependency of these maxima on the number of vertices.

The average size of the $v \times v$ -representation

In each input model, the size of the $v \times v$ -representation increased monotonically as the expected outdegree d grew. The growth was fastest at those values of d where large components started to emerge. When the size of components increased, the size of the representation rapidly approached n^2 words. In model $G(n, p)$, this happened between outdegrees one and two (Figure 5), in model $G(n, p, l = 20)$ between outdegrees two and three, in model $G(n, p, l = 10)$ between outdegrees three and four, and in model $G(n, p, l = 5)$ between outdegrees five and six (Figure 6). Thus, the rapid growth started earlier with greater values of locality l.

The average size of the $c \times c$ -representation

In models $G(n, p)$ and $G(n, p, l = 20)$, the size of the $c \times c$ -representation first increased monotonically as d grew and then decreased monotonically (Figure 7). The maximum average size was about $0.063n^2$ words in $G(n, p)$, reached at $d \approx 1.5$, and about $0.11n^{1.67}$ words in $G(n, p, l = 20)$, reached at $d \approx 2.5$. In model $G(n, p, l = 10)$, the behavior was more complicated (Figure 8). With each number of vertices n, the average size had two local maxima, at $d \approx 2$ and at $d \approx 4$. The latter maximum was the global maximum, about $0.61n^{1.25}$ words. In model $G(n, p, l = 5)$, the behavior was simpler: the average size increased when the outdegree grew from $d = 0$ to $d \approx 2$ and then decreased (Figure 9). The maximum average size was about $2.8n$ words.

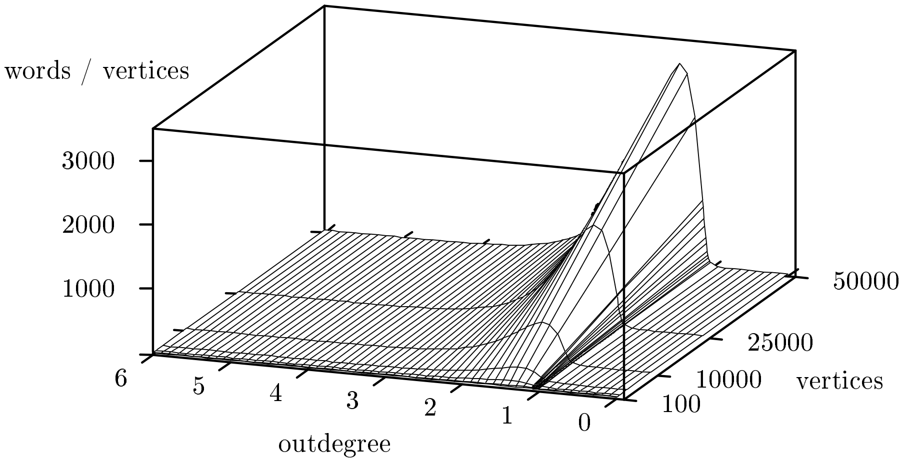


FIGURE 7: The size of the $c \times c$ -representation in model $G(n,p)$.

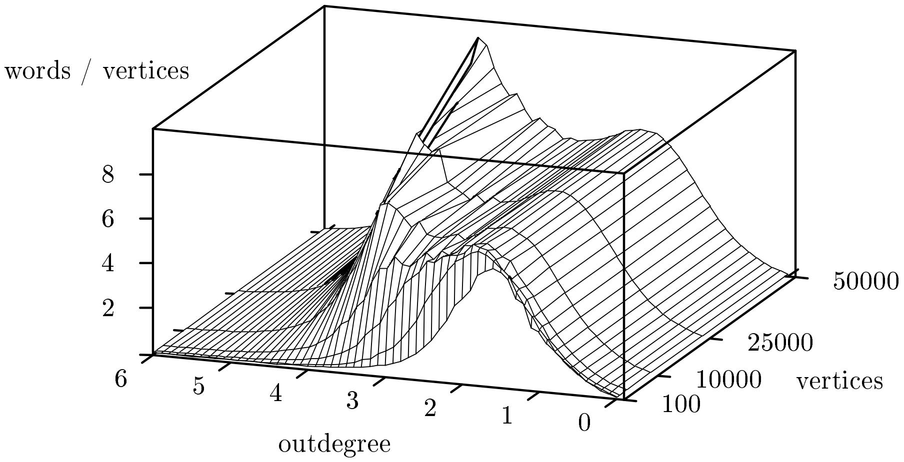


FIGURE 8: The size of the $c \times c$ -representation in model $G(n,p,l=10)$.

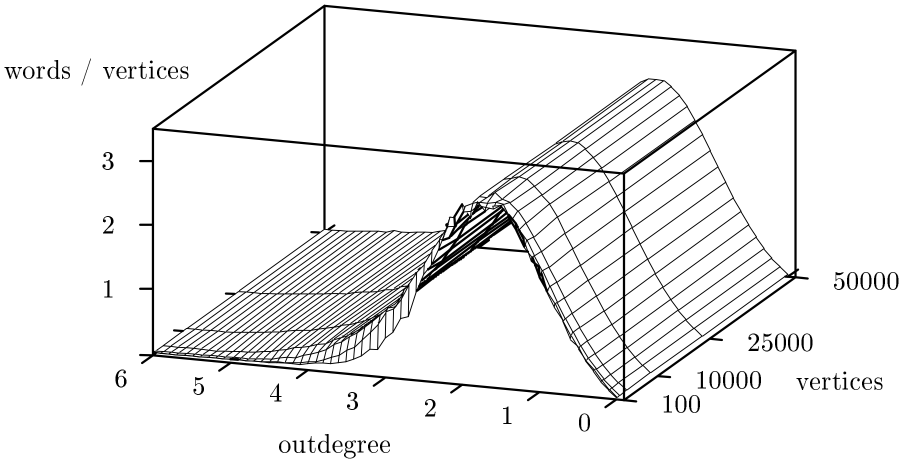


FIGURE 9: The size of the $c \times c$ -representation in model $G(n,p,l=5)$.

The average size of the chain representation

The chain representation behaved roughly like the $c \times c$ -representation in all input models: the size of the $c \times c$ -representation was at most twice as large as the size of the chain representation in the experiments.

The average size of the interval representation

In models $G(n,p,l)$, the average size first increased when the outdegree grew and then decreased. With all values of l, the average size grew linearly to the number of vertices. The maximum average size was about $2.2n$ words in $G(n,p,l=5)$, $2.6n$ words in $G(n,p,l=10)$, and $3.3n$ words in $G(n,p,l=20)$. In model $G(n,p)$, the behavior was otherwise similar to the behavior in models $G(n,p,l)$ except that the maximum average size did not grow linearly to the number of vertices, but was about $0.79n \ln n$ words (Figure 10).

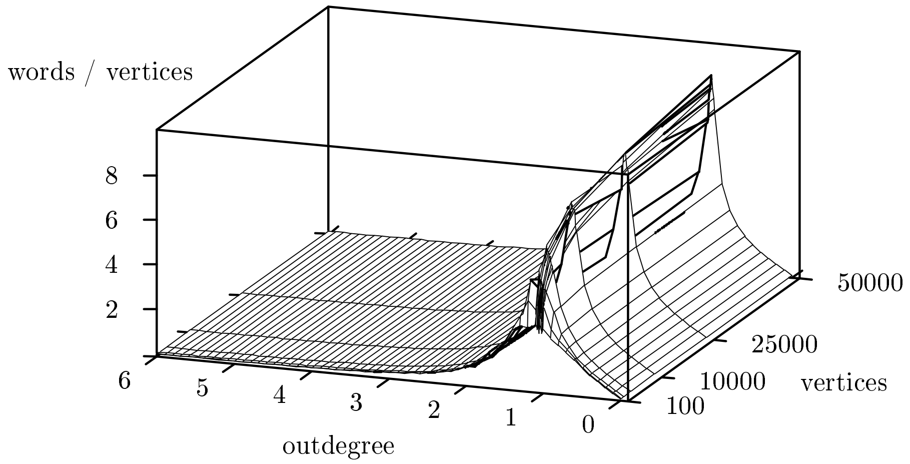


FIGURE 10: The size of the interval representation in model $G(n,p)$.

4.3.3 Discussion

Table I shows the maximum average sizes of the representations at values $n = 10000$, $n = 25000$, and $n = 50000$. In addition, it shows the estimated growth of the maximum average size as a function of n.

In all input models, the maximum average size was largest in the $v \times v$ -representation and smallest in the interval representation. The $v \times v$ -representation required quadratic space in all input models and in most expected outdegrees. Hence, the $v \times v$ -representation can only be used with small inputs. Strong components must be used if larger inputs are processed. Contrary to our initial expectations, the chain representation saved very little space compared to the $c \times c$

TABLE I: The maximum average sizes of the representations.

Representation	Locality	$n = 10000$	$n = 25000$	$n = 50000$	Growth
$v \times v$	none	1.0×10^8	6.3×10^8	2.5×10^9	n^2
	$l = 20$	1.0×10^8	6.3×10^8	2.5×10^9	n^2
	$l = 10$	1.0×10^8	6.3×10^8	2.5×10^9	n^2
	$l = 5$	1.0×10^8	6.3×10^8	2.5×10^9	n^2
$c \times c$	none	6.3×10^6	3.9×10^7	1.6×10^8	$0.063n^2$
	$l = 20$	$590,000$	2.5×10^6	7.9×10^6	$0.11n^{1.67}$
	$l = 10$	$65,000$	$200,000$	$460,000$	$0.61n^{1.25}$
	$l = 5$	$29,000$	$71,000$	$140,000$	$2.8n$
Chains	none	3.4×10^6	–	–	$0.034n^2$
	$l = 20$	$470,000$	2.0×10^6	6.5×10^6	$0.067n^{1.7}$
	$l = 10$	$55,000$	$170,000$	$410,000$	$0.55n^{1.25}$
	$l = 5$	$14,000$	$35,000$	$70,000$	$1.4n$
Intervals	none	$73,000$	$200,000$	$430,000$	$0.79n \ln n$
	$l = 20$	$33,000$	$82,000$	$160,000$	$3.3n$
	$l = 10$	$27,000$	$66,000$	$130,000$	$2.6n$
	$l = 5$	$21,000$	$54,000$	$110,000$	$2.2n$

-representation. Thus, the chain representation seems to be useful only with acyclic graphs. The interval representation was the clear winner. Its maximum average size was linear to the number of vertices in models $G(n, p, l)$ and approximately $0.79n \ln n$ in model $G(n, p)$. The maximum average sizes of all other representations were quadratic to the number of vertices at least in one of the input models. The interval representation was usually smaller than the input graph.

In other experiments [18], we compared another list representation and another chain representation to the representations discussed above. In the list representation, each strong component had a single successor list that contained vertices instead of components. In the chain representation, the successor sets were represented as fixed-width vectors. Both representations were clearly inferior to the $c \times c$ -representation.

4.4 The complexity of successor set construction

4.4.1 Experimental setting

Our goal was to compare the average complexity of constructing the successor sets in the interval representation when different transitive closure algorithms are used. The algorithms were algorithm COMP_TC [8, 18], Schmitz's algorithm [9], algorithm CR_TC [8], and Ebert's algorithm [5]. In Ebert's algorithm, intervals were based on the depth-first numbers of the vertices because the algorithm builds the successor sets using vertices instead of strong components.

Since the sizes of the operands affect the complexity of a set operation, our metric was the total number of intervals accessed in all successor set operations during the transitive closure computation. We call this the *access count*. We also counted separately the numbers of intervals accessed in unions, insertions, and membership tests. The inputs were sparse random graphs of models $G(n, p)$, $G(n, p, l = 5)$, and $G(n, p, l = 10)$. Parameter n had values 100, 300, 1000, 3000, 10000, 25000, 50000, and parameter d had values ranging from 0.0 to 10.0 by 0.1. For each combination of n and d we computed a 95% confidence interval of the average access count. The relative error was at most 5% or the absolute error at most $n/100$.

4.4.2 Results

We present the results using three dimensional plots with the number of vertices and the expected outdegree on the x- and y-axis. On the z-axis, we present the average access count divided by the number of vertices. Note that the scales of the

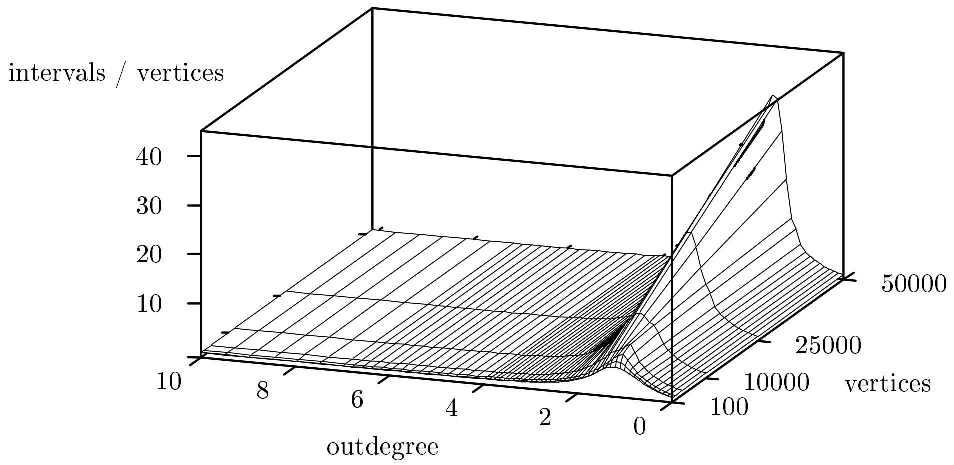


FIGURE 11: The average access count divided by the number of vertices in algorithm COMP_TC: model $G(n, p)$.

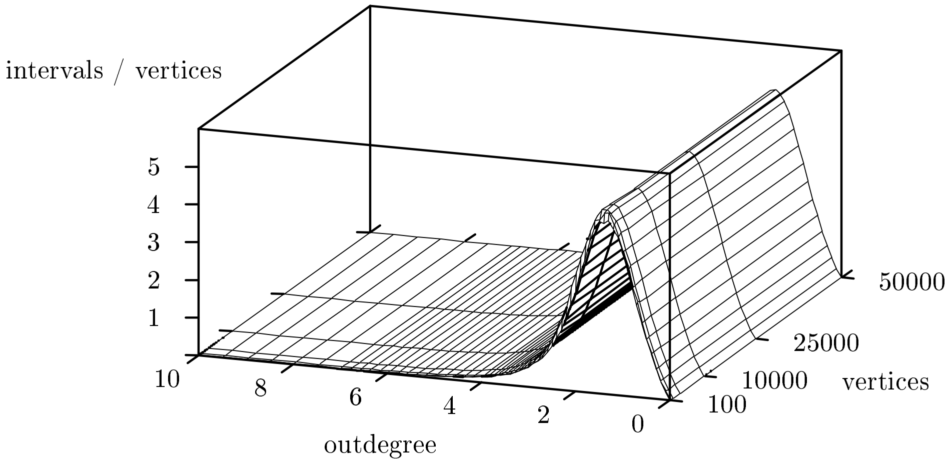


FIGURE 12: The average access count divided by the number of vertices in algorithm COMP_TC: model $G(n, p, l = 10)$.

x-axis and the z-axis vary in different plots. For each number of vertices, we also computed the maximum of the average access counts when the expected outdegree was varied. By using regression models, we estimated the dependency of these maxima on the number of vertices.

Algorithm COMP_TC

In all input models, the average access count increased as the expected outdegree grew from $d = 0.0$ to $d \approx 1.5$ and then decreased. In model $G(n, p)$, the maximum average access count was about $0.006n^{1.8}$ intervals. In models $G(n, p, l = 5)$ and $G(n, p, l = 10)$, the maximum average access counts were about $4.1n$ and $4.8n$

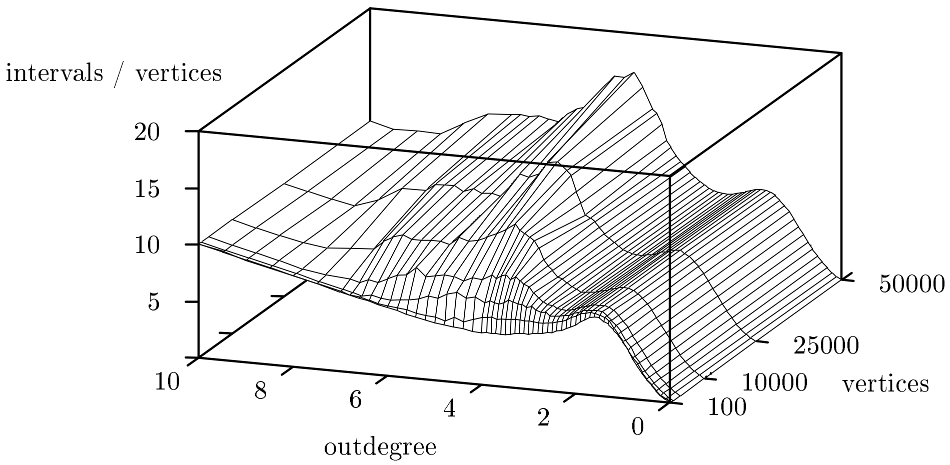


FIGURE 13: The average access count divided by the number of vertices in Schmitz's algorithm: model $G(n, p, l = 10)$.

intervals, respectively, i.e., linear to the number of vertices (Figure 12). In all models, the access counts in unions and insertions were roughly equal.

Schmitz's algorithm

In input model $G(n,p)$, the average access count increased as the expected outdegree grew from $d = 0$ to $d \approx 2$ and then decreased. Unlike in algorithm COMP_TC, in Schmitz's algorithm the access count started to grow again after $d \approx 8.0$. This growth was linear to the expected outdegree and was caused by insertions. The maximum average access count was about $0.015n^{1.8}$ intervals. In models $G(n,p,l=5)$ and $G(n,p,l=10)$, the average access count mostly increased between outdegrees from $d = 0.0$ to $d \approx 5.0$ and then decreased, but started to increase again at $d \approx 8.0$ (Figure 13). The maximum average access count was $5.5n^{1.06}$ intervals in $G(n,p,l=5)$ (reached at $d \approx 4.0$) and $1.5n^{1.22}$ intervals in $G(n,p,l=10)$ (reached at $d \approx 6.0$).

Algorithm CR_TC

In all input models, algorithm CR_TC behaved much like algorithm COMP_TC. The maximum average access count was $4.0n$ intervals in $G(n,p,l=5)$ and $4.9n$ intervals in $G(n,p,l=10)$. In model $G(n,p)$, the maximum average access count was $0.0035n^{1.9}$, i.e., slightly greater than in algorithm COMP_TC.

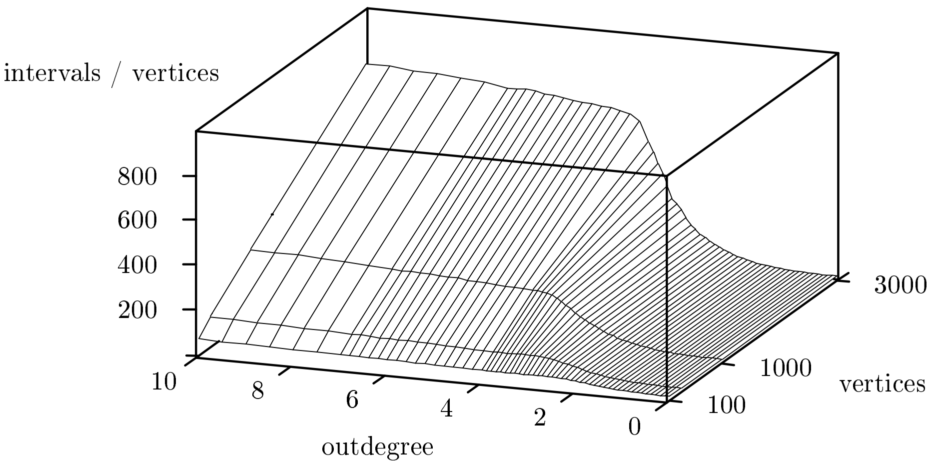


FIGURE 14: The average access count divided by the number of vertices in Ebert's algorithm: model $G(n,p,l=5)$.

Ebert's algorithm

We encountered memory space problems with Ebert's algorithm and could not use vertex counts $n > 3000$. In all models, the maximum average access counts were quadratic or almost quadratic to the number of vertices (Figure 14).

4.4.3 Discussion

Table II lists the maximum average access counts at values $n = 3000$, $n = 10000$, $n = 25000$, and $n = 50000$ when the expected outdegree was between 0.0 and 10.0. In addition, it shows the estimated growth of the maximum average access count as a function of n. The results show that constructing the interval representation is economic when algorithm COMP_TC or CR_TC is used. The average number of accessed intervals was at most linear to the number of vertices in models $G(n, p, l = 5)$ and $G(n, p, l = 10)$. Since the other operations in the algorithms take $O(n + e)$ time, these results indicate that in model $G(n, p, l = 5)$ and $G(n, p, l = 10)$, computing the transitive closure should typically require a time linear to the size of the input graph. The CPU-time measurements that we have done confirm these results. Other experiments that we have done indicate that the situation is similar in all models $G(n, p, l)$ with a small value of l. In model $G(n, p)$, the average number of accessed intervals was also at most linear to the number of

TABLE II: The maximum average access counts.

Algorithm	Loc.	$n = 3000$	$n = 10000$	$n = 25000$	$n = 50000$	Growth
COMP_TC	none	$25,000$	$110,000$	$490,000$	1.8×10^6	$0.006n^{1.8}$
	$l = 10$	$14,000$	$48,000$	$120,000$	$240,000$	$4.8n$
	$l = 5$	$12,000$	$41,000$	$100,000$	$210,000$	$4.1n$
Schmitz	none	$42,000$	$250,000$	1.2×10^6	4.2×10^6	$0.015n^{1.8}$
	$l = 10$	$30,000$	$120,000$	$350,000$	$830,000$	$1.5n^{1.22}$
	$l = 5$	$30,000$	$100,000$	$250,000$	$530,000$	$5.5n^{1.06}$
CR_TC	none	$27,000$	$150,000$	$800,000$	2.9×10^6	$0.0035n^{1.9}$
	$l = 10$	$15,000$	$49,000$	$120,000$	$250,000$	$4.9n$
	$l = 5$	$12,000$	$40,000$	$100,000$	$200,000$	$4.0n$
Ebert	none	2.3×10^6	–	–	–	$0.25n^2$
	$l = 10$	2.1×10^6	–	–	–	$0.52n^{1.9}$
	$l = 5$	2.3×10^6	–	–	–	$0.55n^{1.9}$

vertices except when the expected outdegree was slightly greater than one. Then the average access count was slightly less than quadratic to the number of vertices. Schmitz's algorithm was considerably less economic than COMP_TC and CR_TC in models $G(n, p, l = 5)$ and $G(n, p, l = 10)$, but roughly equally economic in model $G(n, p)$. Ebert's algorithm seems to be useless with the interval representation. The reason for its poor behavior is that the successor sets contain vertices instead of strong components.

5 Conclusions

We presented two new transitive closure representations. The first representation is based on intervals. Each strong component is represented as its reverse topological number, and a successor set is represented as a collection of intervals of consecutive numbers. The representation generalizes the method for compressing the transitive closure of an acyclic graph presented by Agrawal *et al.* [10]. The new representation allows cyclic inputs, and it can be computed during a single depth-first traversal, whereas the representation by Agrawal *et al.* requires many traversals of the input graph.

The second representation uses a chain decomposition of the condensation graph induced by the strong components. This representation generalizes a previous method for storing the closure of an acyclic graph presented by Simon [11]. The new representation allows cyclic inputs and it can be computed during a single depth-first traversal whereas Simon's representation requires several passes over the input graph.

We studied experimentally the average behavior of the representations. We compared the average size of the interval representation, the chain representation, and two list representations; random graphs of two models $G(n, p)$ and $G(n, p, l)$ were used as inputs. The interval representation required a much smaller space than the other representations. The average size of the interval representation was always at most linear to the number of vertices in model $G(n, p, l)$, and also in $G(n, p)$ except when the expected outdegree np was slightly greater than one. In this case, the average size was about $0.79n \ln n$. The chain representation did not save much space compared to a list representation containing strong components.

We also studied the average complexity of constructing the interval representation with different transitive closure algorithms. When algorithm COMP_TC [8, 18] or CR_TC [18] was used, the average complexity was always at most linear to the number of vertices in models $G(n, p, l = 5)$ and $G(n, p, l = 10)$, and also in model $G(n, p)$ except when the expected outdegree was slightly greater than one.

In this case, the average complexity was slightly less than quadratic to the number of vertices. Schmitz's algorithm [9] was less economic than COMP_TC and CR_TC in models $G(n,p,l=5)$ and $G(n,p,l=10)$. Ebert's algorithm [5] required much memory and was very inefficient.

The results indicate that in models $G(n,p)$ and $G(n,p,l)$, the transitive closure can typically be computed in a time linear to the size of the input when the interval representation is used with algorithm COMP_TC or CR_TC. With algorithm COMP_TC, the interval representation can be stored in a continuous block of memory. Since the interval representation is typically smaller than the input graph, the bottleneck of transitive closure computation in a disk environment is the traversal of the input graph, not the production of the successor sets. We study this subject further in [38].

References

[1] Bolognesi and Smolka. Fundamental results for the verification of observational equivalence: A survey. In *Proceedings of the IFIP Int. Conf. on Protocol Specification, Testing and Verification VII*. North-Holland, 1988.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[3] S. Sippu and E. Soisalon-Soininen. *Parsing Theory*, volume I, Languages and Parsing. Springer-Verlag, Berlin, 1988.

[4] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[5] J. Ebert. A sensitive transitive closure algorithm. *Information Processing Letters*, 12:255–258, 1981.

[6] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Informatica*, 8:303–314, 1977.

[7] Y.E. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM Transactions on Database Systems*, 18(3):512–576, September 1993.

[8] E. Nuutila. An efficient transitive closure algorithm for cyclic digraphs. *Information Processing Letters*, 52:207–213, 1994.

[9] L. Schmitz. An improved transitive closure algorithm. *Computing*, 30:359–371, 1983.

[10] R. Agrawal, A. Borgida, and H.V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the ACM-SIGMOD 1989 Conference on Management of Data*, pages 253–262, Portland, Oregon, May-June 1989.

[11] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1-3):325–346, 1988.

[12] P.A. Bloniarz, M.J. Fischer, and A.R. Meyer. A note on the average time to compute transitive closures. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming, 3rd International Colloquium, Edinburgh*, pages 425–434. Edinburgh Press, July 1976.

[13] R. M. Karp. The transitive closure of a random digraph. *Random Structures and Algorithms*, 1(1):73–93, 1990.

[14] C.P. Schnorr. An algorithm for transitive closure with linear expected time. *SIAM Journal of Computing*, 7:127–133, 1978.

[15] K. Pawlikowski. Steady-state simulation of queueing processes: A survey of problems and solutions. *ACM Computing Surveys*, 22(2):123–170, June 1990.

[16] J.N. Hooker. Needed: an empirical science of algorithms. *Operations Research*, 42(2):201–212, March-April 1994.

[17] S. Ganguly, R. Krishnamurthy, and A. Silberschatz. An analysis technique for transitive closure algorithms: A statistical approach. In *Proceedings of the IEEE 7th International Conference on Data Engineering*, pages 728–735, Kobe, Japan, April 1991.

[18] E. Nuutila. *Efficient transitive closure computation in large digraphs.* PhD thesis, Helsinki University of Technology, 1995. Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series, No. 74.

[19] J. Van Leeuwen. Graph Algorithms. Transitive reduction and transitive closure. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 10, section 1.4, pages 539–544. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[20] S. Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[21] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[22] M.J. Fischer and A.R. Meyer. Boolean matrix multiplication and transitive closure. In *Conference Record 1971 12th Annual Symposium on Switching and Automata Theory*, pages 129–131, East Lansing, Michigan, October 1971. IEEE Computer Society.

[23] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 1–6, 1987.

[24] P. Purdom. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

[25] E. Nuutila and E. Soisalon-Soininen. Efficient transitive closure computation. Technical Report TKO-B113, Helsinki University of Technology, Laboratory of Information Processing Science, 1993.

[26] E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49:9–14, 1994.

[27] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[28] A. Goralcikova and V. Koubek. A reduct and closure algorithm for graphs. In *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 301–307. Springer-Verlag, 1979.

[29] S. Dar. *Augmenting Databases with Generalized Transitive Closure.* PhD thesis, University of Wisconsin, Madison, Computer Sciences Department, 1993. Technical Report #1206.

[30] R. Agrawal and H.V. Jagadish. Multiprocessor transitive closure algorithms. In *Proceedings of International Symposium on Databases and Distributed Systems*, pages 55–66, Austin, Texas, December 1988.

[31] F. Cacace, S. Ceri, and M.A.W. Houtsma. An overview of parallel strategies for transitive closure on algebraic machines. In *Proceedings of the Workshop on Parallel Database Systems*, volume 503 of *Lecture Notes in Computer Science*, pages 44–62. Springer-Verlag, 1990.

[32] M.A.W. Houtsma, P.M.G. Apers, and G.L.V. Schipper. Data fragmentation for parallel transitive closure strategies. In *Proceedings of the IEEE 9th International Conference on Data Engineering*, pages 447–456, Vienna, Austria, April 1993.

[33] Y-N. Huang and J-P. Cheiney. Parallel computation of direct transitive closures. In *Proceedings of the IEEE 7th International Conference on Data Engineering*, pages 192–199, Kobe, Japan, April 1991.

[34] H.V. Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems*, 15(4), December 1990.

[35] C. McGeoch. Analyzing algorithms by simulation. *Computing Surveys*, 24(2):195–212, June 1992.

[36] E.J. Dudewicz and N.M. Satya. *Modern Mathematical Statistics.* John Wiley & Sons, Inc., New York, 1988.

[37] B. Bollobás. *Random Graphs.* Academic Press, New York, 1985.

[38] V. Hirvisalo, E. Nuutila, and E. Soisalon-Soininen. Disk performance analysis of transitive closure algorithms, 1996. Submitted for publication.