Ma 74 UDC 681.3

ACTA
POLYTECHNICA
SCANDINAVICA

MATHEMATICS AND COMPUTING IN ENGINEERING SERIES No. 74

Efficient Transitive Closure Computation in Large Digraphs

Esko Nuutila

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

Dissertation for the degree of Doctor of Technology to be presented with due permission for public examination and debate in Auditorium E at Helsinki University of Technology (Espoo, Finland) on the 16th of June, 1995, at 12 o'clock noon.

HELSINKI 1995

Nuutila, E., **Efficient Transitive Closure Computation in Large Digraphs**. Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series No. 74, Helsinki 1995, 124 pages. Published by the Finnish Academy of Technology. ISBN 951-666-451-2. ISSN 1237-2404. UDC 681.3.

Keywords: algorithms, data structures, directed graphs, transitive closure, strong components, Tarjan's algorithm, intervals, chain decomposition, random graphs, simulation.

Abstract

This thesis examines new efficient transitive closure algorithms and representations. Two new transitive closure algorithms that are based on detecting the strong components are presented. Worst-case analysis and simulation experiments show that the new algorithms are more efficient than the previous algorithms that are based on strong component detection. The algorithms use Tarjan's algorithm for strong component detection. Also, two improved versions of Tarjan's algorithm are presented.

Two new compact transitive closure representations are presented. The first representation is based on intervals of consecutively numbered strong components. The representation generalizes a previous method for compressing the transitive closure of an acyclic graph. The new representation also handles cyclic graphs, and it can be computed more efficiently than the previous representation. The second new representation generalizes the chain decomposition representation for acyclic graphs to handle cyclic graphs.

Simulation experiments show that the interval representation is superior to the commonly used adjacency list representation. The experiments indicate that the interval representation requires typically at most a space linear to the number of vertices in the input graph. The experiments also indicate that with the interval representation and the new algorithms, the transitive closure can be computed typically in time linear to the size of the input graph.

Preface

This thesis is the result of my work in the data structure and algorithm research group of the Laboratory of Information Processing Science at Helsinki University of Technology during the years 1992 to 1994. The work was financed by the Academy of Finland, Helsinki Graduate School in Computer Science and Engineering, Emil Aaltonen's foundation, Leo and Regina Wainstein's foundation, and Jenny and Antti Wihuri's foundation. The Center for Scientific Computing provided computational resources for the simulation studies I did.

I thank professor Eljas Soisalon-Soininen, the head of our research group and my thesis supervisor. Without him, this thesis would not have been possible. Before Eljas joined our laboratory in 1991, I had more or less lost the hope of finding an interesting subject for a doctoral thesis. Eljas encouraged me to enter the field of algorithm research. In our discussions with Eljas, we soon found an interesting subject for my thesis. During my work, Eljas helped me by giving me new ideas and discussing the ones that I got.

I thank Vesa Hirvisalo for his friendship and support during all my years at Helsinki University of Technology. Vesa's studies on the disk behavior of my transitive closure algorithms and our discussions have been useful for my work.

I thank Ilkka Karanta and Pertti Laininen for their advice on statistical methods.

I thank professors Reino Kurki-Suonio and Martti Tienari for their thesis review and their valuable comments. I also thank Elizabeth Heap-Talvela for checking the English language of my thesis.

I present special thanks to my colleagues in room U401. The nice atmosphere of U401 has always cheered me up when the work has been difficult.

Finally, I thank my wife Raisa and my daughters Katariina and Emilia for their love, support, and patience.

Otaniemi, May 1995

Esko Nuutila

Contents

1 Introduction **7**
- 1.1 Nature and scope of the problem . . . 7
- 1.2 Goals . . . 9
- 1.3 Methodology . . . 9
- 1.4 Contributions . . . 10
- 1.5 Thesis outline . . . 11

2 Transitive closure problem **13**
- 2.1 Graph theoretic preliminaries . . . 13
- 2.2 Defining the problem . . . 18
  - 2.2.1 Related problems . . . 18
- 2.3 Previous work . . . 19
  - 2.3.1 Warshall's algorithm . . . 19
  - 2.3.2 Algorithms based on detecting the strong components . . . 20
  - 2.3.3 Algorithms based on matrix multiplication . . . 22
  - 2.3.4 Transitive closure computation in databases . . . 23
  - 2.3.5 Representations and dynamic updates . . . 30

3 Transitive closure computation using strong components **35**
- 3.1 Strong component detection – Tarjan's algorithm . . . 35
  - 3.1.1 Analysis . . . 37
  - 3.1.2 Improvements . . . 42
- 3.2 Adapting Tarjan's algorithm to transitive closure computation . . . 48
- 3.3 New algorithm CR_TC . . . 56
- 3.4 New algorithm STACK_TC . . . 60
- 3.5 Comparisons with previous algorithms . . . 65

4 Representing successor sets **71**
- 4.1 Properties of common set representations . . . 71
- 4.2 Interval representation . . . 72
- 4.3 Chain representation . . . 76
- 4.4 Avoiding multiple paths . . . 80

5 Simulations **83**
- 5.1 Method . . . 83
  - 5.1.1 Mathematical algorithm analysis . . . 83

5.1.2 Performance evaluation by simulation 84
5.1.3 Inputs 87
5.1.4 Performance evaluation design 90
5.2 The size of transitive closure representations 92
5.2.1 Experimental setting 92
5.2.2 Results 93
5.2.3 Discussion 96
5.3 Successor set construction 97
5.3.1 Experimental setting 97
5.3.2 Results 98
5.3.3 Discussion 103
5.4 Execution time 105
5.4.1 Experimental setting 105
5.4.2 Results 105
5.4.3 Discussion 105

6 Conclusions **111**
6.1 Summary of the main results 111
6.2 Further research 112
Bibliography 114

Chapter 1

Introduction

Efficient computation of the transitive closure of a directed graph is required in many applications, for instance, in the reachability analysis of transition networks representing distributed and parallel systems [20, 38] and in the construction of parsing automata in compiler construction [12, 114]. Recently, efficient transitive closure computation has been recognized as a significant subproblem in evaluating recursive database queries [24].

Several transitive closure algorithms have been presented during the last thirty years. Despite the increased efficiency of computers, the need for more efficient transitive closure algorithms and representations remains. This is for two reasons. First, the size of the inputs seems to grow in proportion to the growth of the memory capacity. Since the CPU speed has grown at the same rate as the memory capacity, only linear algorithms have retained their execution times on typical inputs. Traditional transitive closure algorithms, such as [36, 40, 91, 101, 105, 128, 129], are not linear. Second, typical inputs and outputs in modern applications, e.g., in the area of databases, do not fit into the main memory. Traditional transitive closure algorithms are designed for main memory operation.

In the study described here, we developed new efficient algorithms and representations for transitive closure computation. We have published part of these results previously in [92, 93, 94, 95, 96].

In the rest of the introduction, we discuss the nature and the scope of the problem, list the goals of our study, explain some methodological choices we have made, list the main results of the study, and present the thesis outline. We define the terminology used in the thesis and review previous literature on transitive closure in Chapter 2.

1.1 Nature and scope of the problem

Several variants of the transitive closure problem exist. In the *all-pairs 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 compute the vertices that are reachable from a given set of vertices via non-null paths.

We studied the all-pairs transitive closure problem. We assumed that the input graphs may be large, i.e., they may require a considerable portion of the main memory of the computer

or they may even be too large to fit into the main memory and must reside in the secondary memory.

The reader may wonder whether the problem is reasonable. Many input graphs of n vertices and $O(n)$ edges have a transitive closure of $O(n^2)$ pairs. If such input graphs do not fit into the main memory, their transitive closures cannot fit into any reasonable amount of secondary memory!

This is true if traditional set representations, such as bit matrices, bit vectors, or lists, are used for representing the transitive closure. These representations do not take advantage of the properties of directed graphs. To overcome this problem, we studied techniques for storing the transitive closure more compactly than with traditional representations

An example of an application that requires the all-pairs transitive closure computation of large graphs is the reachability analysis of transition networks representing distributed and parallel systems [20, 38]. Another example is the view materialization of transitive relationships in data and knowledge bases [4, 67]. The transitive closure of a relation can be precomputed and stored to enable rapid evaluation of transitive queries.

A major difficulty in transitive closure computation is the avoidance of redundant operations. A directed graph may contain pairs of vertices that are connected via multiple paths. Several vertices may have the same set of successor vertices. The graph in Figure 1.1 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.

Most of the redundant operations in many algorithms are caused by the strong components of the input graph [44], since all vertices in a strong component have the same successor set. For example in Figure 1.1, vertices f, g, h, and i are in the same strong component. Some transitive closure algorithms presented in the literature try to avoid these redundancies by detecting the strong components [36, 40, 62, 64, 91, 101, 105]. Unfortunately, these algorithms do not efficiently avoid all redundancies caused by strong components. Some algorithms generate a partial successor set for each vertex of a component. Others scan the whole input graph more than once. In computing the successor sets, the algorithms do unnecessary set operations. The algorithm that we seek should avoid these deficiencies. Further, we should find a mechanism to efficiently eliminate the redundant computations caused by multiple paths between pairs of vertices.

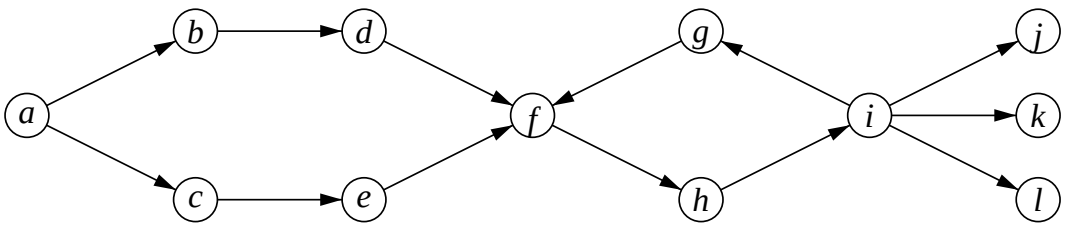


FIGURE 1.1: A directed graph with vertices having the same successor sets and with multiple paths between vertices.

1.2 Goals

Our first goal was to find new algorithms and data representations for transitive closure computation that are more efficient than previous methods presented in the literature. We searched for transitive closure algorithms that require a time linear to the size of the input graph in the average case.

Our second goal was to find a representation for transitive closure that makes it possible to store the transitive closure of a typical input graph in a reasonable amount of memory. To reach the first goal, we needed a representation that requires at most a space linear to the size of the input graph in the average case.

Our third goal was that the methods that we develop can be used efficiently both in a main memory environment, i.e., when the input graph, the intermediate data structures and the output fit into the main memory simultaneously, and in a secondary memory environment, i.e., when the input graph does not fully fit into the main memory together with the output and the intermediate data structures.

1.3 Methodology

In the design and analysis of algorithms and data structures, we mostly used the scientific method that is discussed in text books like [10, 11, 107]. We discuss here one methodological choice that we made, since it is somewhat unconventional.

We needed a way to study the average-case performance of algorithms. Unfortunately, the mathematical average-case analysis is much more difficult than the mathematical worst-case analysis. No general method for mathematical average-case analysis exists, and the result of a successful average-case analysis is usually a rough asymptotic estimate. Only a couple of previous articles on transitive closure computation contain an average-case analysis [18, 79, 106], and the approach used in these articles cannot be generalized to other transitive closure algorithms.

Our choice, therefore, was to study the average-case performance by computer simulations. A benefit of this choice was that we could use the same technique for all input graph models and for all 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 numerically more accurate than a mathematical analysis could yield.

We confirmed the accuracy of the results by using sound statistical techniques in running the simulations and in analyzing the simulation outputs. We used *sequential simulation procedures* [83, 84, 99] that automatically yield the desired statistical confidence interval for the estimated measure.

A weakness in simulations is that they do not give information on the asymptotic performance. We only get information on how the algorithm behaves in that area of the space of possible inputs that our simulations cover. Luckily, it is easy to show analytically the asymptotic average-case performance of the algorithms that we developed in parts of the space of inputs, namely when the inputs are dense graphs.

Another weakness, which we gradually detected, is that simulation studies require much

more time than mathematical analysis. Implementing and testing the simulation programs took much time, but designing and running the simulation experiments took even more time.

1.4 Contributions

The main results of this thesis are the following:

- Two efficient transitive closure algorithms, called CR_TC and STACK_TC, presented in section 3.3 and section 3.4, respectively. The new algorithms are based on detecting the strong components of the input graphs using Tarjan's algorithm [118]. Unlike previous transitive closure algorithms that are based on strong component detection [36, 40, 64, 101, 105], the new algorithms scan the input graph exactly once without generating a partial successor set for most vertices of the input graph. All previous algorithms either scan the whole input graph more than once or generate a non-empty partial successor set for most vertices of the input graph. The worst-case analysis and the simulations that we present in Chapter 5 show that algorithm STACK_TC is more efficient than the previous algorithms.

- Two improved versions of Tarjan's algorithm [118] for strong component detection. The improved versions eliminate unnecessary stack operations in Tarjan's algorithm. These algorithms are presented in section 3.1.2.

- A compact transitive closure representation that can be efficiently used with our new transitive closure algorithms. The representation, presented in section 4.2, is based on intervals of consecutively numbered strong components, and it generalizes the method for compressing the transitive closure of an acyclic digraph presented by Agrawal et al. [4]. Our representation can be used for all kinds of input graphs and it can be computed during a single depth-first traversal of the graph, whereas the representation by Agrawal et al. [4] requires several traversals of the graph.

- Another new representation that can be used with our new transitive closure algorithms. The representation, presented in section 4.3, is based on the chain decomposition method for acyclic digraphs by Simon [112]. This new representation can also be used for all kinds of input graphs and it can be computed during a single depth-first traversal of the graph, whereas the representation by Simon [112] requires several traversals of the graph. According to our experiments (see section 5.2), this new representation is usually inferior to the interval representation.

- Experimental results indicating that the interval representation usually requires at most a space linear to the number of vertices of the input graph (see section 5.2). In a random graph model $G(n, p, l)$, the average size of the interval representation was always at most linear to the number of vertices of the input graph. In a random graph model $G(n, p)$, the average size was at most linear to the number of vertices, except when the expected outdegree of the graph was slightly greater than one. In this case, the size was approximately $0.55n \log n$. Note that these results are not asymptotic, since simulation

studies only give information on the performance in the part of the space of inputs that is covered in the simulation runs.

- Experimental results indicating that algorithm STACK_TC with the interval representation usually requires a time linear to the size of the input graph (see sections 5.3 and 5.4). Model $G(n, p)$ was an exception: when the expected outdegree was slightly greater than one, the execution time seemed to be quadratic to the number of vertices.

1.5 Thesis outline

In Chapter 2, we define the graph theoretic terminology that we use in the thesis, define the problem and discuss its variations, and describe previous solutions to the problem.

In Chapter 3, we study transitive closure algorithms that are based on strong component detection. We first describe Tarjan's strong component algorithm [118] and present some improvements to it. Then, starting from a simple adaptation of Tarjan's algorithm, we develop and analyze two new transitive closure algorithms STACK_TC and CR_TC. In the end of the chapter, we compare these algorithms to previous algorithms and show that the new algorithms are more efficient.

In Chapter 4, we study methods for representing transitive closure efficiently. We describe two new representations, one that is based on intervals of strong component numbers and another that is based on the chain decomposition of the input graph. We study how the new representations eliminate redundant computations caused by multiple paths between pairs of vertices.

In Chapter 5, we present simulation experiments on the average-case performance of the algorithms and representations developed in Chapters 3 and 4 and compare them to previous methods presented in the literature.

In Chapter 6, we present the conclusions of the thesis.

Chapter 2

Transitive closure problem

In this chapter, we define the transitive closure problem and the terminology needed in our study. We also describe briefly other problems that are related to the transitive closure problem. After that, we review previous solutions to the problem.

2.1 Graph theoretic preliminaries

Since the definitions of graph theoretical concepts differ somewhat in the literature, we define here the basic concepts. The definitions are adapted from references [10, 11, 89, 107, 114, 118]. A reader who is familiar with graph theory may skip this subsection and refer to these definitions later if it is needed. We define some other concepts later in the text where their relevance is more obvious.

Definition. A directed *graph* G is a pair (V, E) where V is a set of elements called *vertices* and $E \subseteq V \times V$ is a set of ordered pairs called *edges*. The cardinality of V is denoted by n and the cardinality of E by e. The *size of graph* G is $n + e$. Given an edge (v, w), v is the *tail* and w the *head* of the edge. A *subgraph* of a graph $G = (V, E)$ is a graph $S = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$.

In this thesis, we study only directed graphs. From here on, the word "graph" always refers to a directed graph; we omit the qualifier "directed." The graphs that we consider are finite.

Some presentations, e.g., [89], prohibit self-loop edges of form (v, v) in a graph. A consequence is that the transitive closure of a cyclic graph is not a graph, since it always has self-loops. Since the self-loops introduce no difficulties in our presentation, we allow them.

Definition. If (u, v) is an edge of G, we say that u is *adjacent to* v, and v is *adjacent from* u. The number of vertices adjacent to v is the *in-degree* of v, denoted $Indeg(v)$, and the number of vertices adjacent from v is the *out-degree* of v, denoted $Outdeg(v)$.

The in-degrees and out-degrees are connected by the following equation:

$$\sum_{v \in V} Indeg(v) = \sum_{v \in V} Outdeg(v) = e \tag{2.1}$$

To process graphs, we have to select a representation for them. One common representation is the *adjacency matrix*.

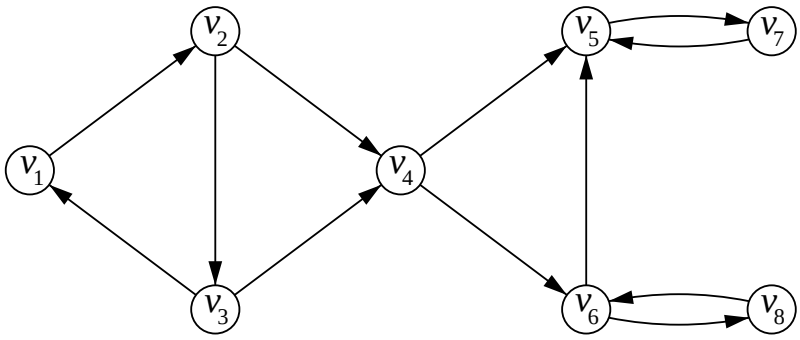
Definition. An *adjacency matrix* of a graph $G = (V, E)$ is an $n \times n$ Boolean matrix A such that $A[i, j]$ is **true** iff (if and only if) G has an edge (v_i, v_j).

Checking the presence of an edge (v_i, v_j) takes $O(1)$ time in an adjacency matrix. Enumerating the vertices adjacent to or adjacent from a vertex v takes $\Theta(n)$ time regardless of the in-degree or out-degree of v. The main disadvantage of the adjacency matrix is that it takes $\Theta(n^2)$ space even when the graph is *sparse*, i.e., it has much fewer than n^2 edges. For instance, if the adjacency matrix resides on a disk file, simply to read in the matrix takes $\Omega(n^2)$ time. A more suitable representation for sparse graphs is the *adjacency list representation*.

Definition. An *adjacency list* $AdjFrom(v)$ of vertex v is a list that contains the vertices adjacent from v. The adjacency list representation of a graph consists of the adjacency lists of its vertices.

The adjacency list representation takes $O(n + e)$ space. Enumerating the vertices adjacent from a vertex v takes $O(Outdeg(v))$ time, and enumerating all edges of the graph takes $O(n+e)$ time. Checking the presence of an edge (v, w) takes $O(Outdeg(v))$ time. Enumerating the vertices adjacent to a vertex v may take $O(n + e)$ time, since we must check the presence of v in each adjacency list. If the vertices adjacent to a vertex v are often needed, we can use another list $AdjTo(v)$ for storing them. If the graph is dense, i.e., the number of edges is close to n^2, the adjacency matrix representation is more economical, since the constant costs for storing an edge are higher in an adjacency list than in an adjacency matrix.

Example 2.1. In Figure 2.1, we present an example graph $G = (V, E)$. The vertices are shown as circles and the edges as arrows going from the tail of the edge to the head of the edge. Below G we present its adjacency matrix and adjacency list representations. □



	1	2	3	4	5	6	7	8
1	0	1	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0
3	1	0	0	1	0	0	0	0
4	0	0	0	0	1	1	0	0
5	0	0	0	0	0	0	1	0
6	0	0	0	0	1	0	0	1
7	0	0	0	0	1	0	0	0
8	0	0	0	0	0	1	0	0

$AdjFrom(v_1) = \{v_2\}$
$AdjFrom(v_2) = \{v_3, v_4\}$
$AdjFrom(v_3) = \{v_1, v_4\}$
$AdjFrom(v_4) = \{v_5, v_6\}$
$AdjFrom(v_5) = \{v_7\}$
$AdjFrom(v_6) = \{v_5, v_8\}$
$AdjFrom(v_7) = \{v_5\}$
$AdjFrom(v_8) = \{v_6\}$

FIGURE 2.1: An example graph $G = (V, E)$ and its representations.

Definition. A *path from vertex v_0 to vertex v_k* in G, denoted $v_0 \xrightarrow{*} v_k$, is a sequence of edges of the form $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, where each edge is in E. The *length of a path* is the number of edges in it. A path from v to u is *non-null*, denoted $v \xrightarrow{+} u$, if its length is positive. A path is *simple* if all edges and vertices on the path, except possibly the first and the last vertex, are distinct. A *cycle* is a non-null simple path that begins and ends at the same vertex. A graph that contains no cycles is *acyclic*. If we do not explicitly say that a graph is acyclic, we assume that it may contain cycles, i.e., it may be *cyclic*.

Example 2.2. In Figure 2.1, $((v_1, v_2), (v_2, v_3), (v_3, v_4))$ is a simple path and path $((v_1, v_2), (v_2, v_3), (v_3, v_1))$ is a cycle. Path $((v_4, v_6), (v_6, v_8), (v_8, v_6), (v_6, v_5))$ is not simple. Graph G of Figure 2.1 is cyclic. □

Definition. The *transitive closure of graph* $G = (V, E)$ is a graph $G^+ = (V, E^+)$ such that E^+ contains an edge (v, w) iff G contains a non-null path $v \xrightarrow{+} w$. The size of the transitive closure is denoted by e^+. The *successor set* of a vertex v is the set $Succ(v) = \{w \mid (v, w) \in E^+\}$, i.e., the set of all vertices that can be reached from vertex v via non-null paths. The *predecessor set* of a vertex v is the set $Pred(v) = \{u \mid (u, v) \in E^+\}$, i.e., the set of all vertices that v is reachable from via non-null paths. The vertices adjacent from vertex v are the *immediate successors of* v and the vertices adjacent to v are the *immediate predecessors of* v.

Example 2.3. The transitive closure of graph G of Figure 2.1 is presented in Figure 2.2. As we see, the successor set of vertices v_1, v_2, and v_3 is V; the successor set of vertices v_4, v_6, and v_8 is $\{v_5, v_6, v_7, v_8\}$; and the successor set of v_5 and v_7 is $\{v_5, v_7\}$. Similarly, several vertices have a common predecessor set. □

Definition. The *reflexive transitive closure of graph* $G = (V, E)$ is a graph $G^* = (V, E^*)$ such that E^* contains an edge (v, w) iff G contains a (possibly null) path $v \xrightarrow{*} w$. The size of the reflexive transitive closure is denoted by e^*.

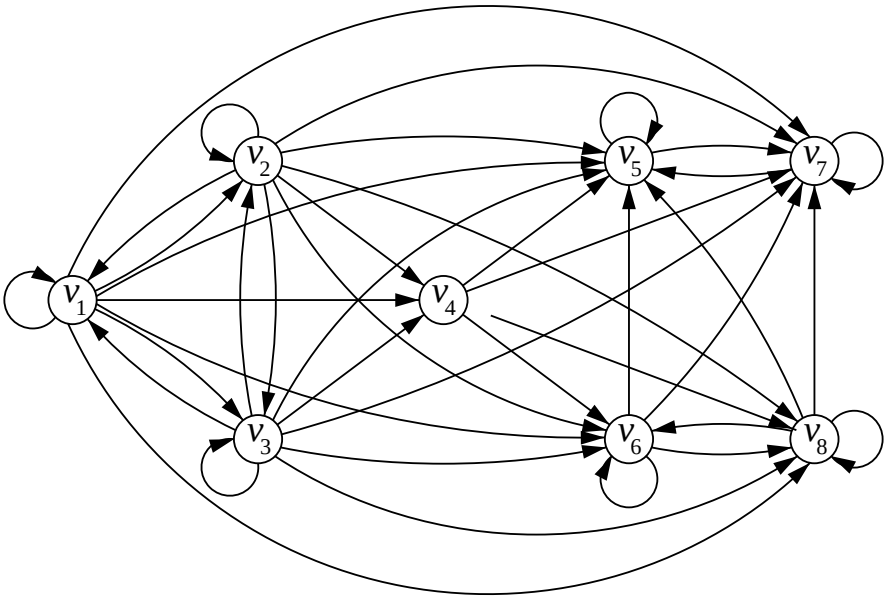


FIGURE 2.2: The transitive closure of graph G of Figure 2.1.

Example 2.4. The transitive closure of graph G, presented in Figure 2.2, can be changed to the reflexive transitive closure of graph G by inserting the edge (v_4, v_4), since $E^* = E^+ \cup I$ where $I = \{(v, v) \mid v \in V\}$, the identity relation. □

Definition. Two vertices v and w in G are *path equivalent* iff G contains a path $v \xrightarrow{*} w$ and a path $w \xrightarrow{*} v$. Path equivalence divides V into maximal disjoint sets of path equivalent vertices. These sets are called the *strong components* of G. The strong component containing vertex v is denoted $Comp(v)$. A strong component that contains only one vertex is called a *trivial component*. The *condensation graph* $\overline{G} = (\overline{V}, \overline{E})$ *induced by the strong components of graph* G is a graph such that $\overline{V}$ is the set of the strong components of G and $\overline{E}$ contains an edge (X, Y) iff E contains an edge (u, v) such that $Comp(u) = X$ and $Comp(v) = Y$.

Each vertex of a strong component has the same successor set; this can be used in designing efficient transitive closure algorithms.

Example 2.5. The strong components of graph G are encircled in Figure 2.3(a). The condensation graph $\overline{G}$ induced by the strong components of G is shown in Figure 2.3(b). Note that the condensation graph is always acyclic if we remove the self-loop edges. □

Definition. 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) is 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 $\leq$ of a vertex set V is called a *reverse topological order* of V.

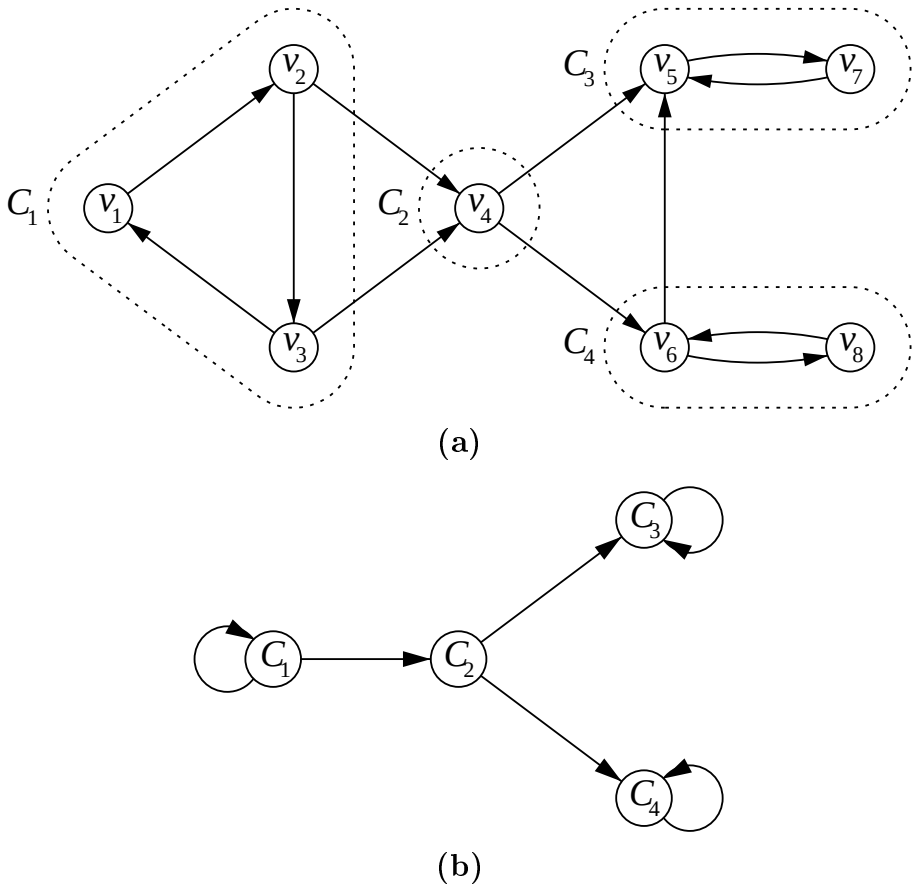


FIGURE 2.3: **(a)** The strong components of graph G of Figure 2.1 and **(b)** the condensation graph induced by the strong components of G.

Each acyclic graph has at least one topological order. Since a condensation graph $\overline{G}$ is an acyclic graph augmented possibly with self-loop edges, each condensation graph $\overline{G}$ also has a topological order. No graph that has cycles of more than one vertex has a topological order.

Definition. The *transitive reduction* $G_r = (V, E_r)$ of a graph $G = (V, E)$ is a graph that has the same transitive closure as G, but has as few edges as possible. The size of the transitive reduction is denoted by e_r. The transitive reduction is not necessarily unique.

Definition. A (directed rooted) *tree* T is an acyclic graph satisfying the following properties:

1. T has exactly one vertex r (called the *root*) that is the head of no edge.
2. Each vertex except the root is the head of exactly one edge.
3. From the root r to each vertex v exists a unique path $r \stackrel{*}{\rightarrow} v$.

If v is a vertex in a tree T, a *subtree* T_v is the maximal subgraph of T that has $\{v\} \cup Succ(v)$ as its vertex set. A graph consisting of a collection of trees is a *forest*. A tree T (a forest F) is a *spanning tree* (a *spanning forest*) of a graph G if T (F) is a subgraph of G and T (F) contains all the vertices of G.

Example 2.6. Figure 2.4 shows one spanning tree of graph G of Figure 2.1. □

The set of edges $E \subseteq V \times V$ is a binary relation. Conversely, every binary relation R in a domain V can be seen as a directed graph with vertex set V and edge set R. We can formulate the transitive closure also in the terms of binary relations.

Definition. The *transitive closure of a binary relation* $E \subseteq V \times V$ is a relation $E^+ \subseteq V \times V$: $E^+ = \bigcup_{n>0} E^n$ where $E^0 = I$ and $E^{n+1} = E^n \circ E = E \circ E^n$. Here I is the identity relation and $\circ$ is the composition operator.

We use the graph theoretic formulation except when we describe previous algorithms that are based on the relational formulation. In these relationally oriented algorithms, the graph is usually represented as a table of pairs, and one or more indices are used to efficiently access the edges leaving or entering a vertex.

Example 2.7. Consider the example graph $G = (V, E)$ of Figure 2.1. We can compute the transitive closure of any graph by using only its simple paths. Generally, the longest simple path in a graph has at most n edges, but in our example graph the longest simple path has five edges. Thus, $E^+ = E \cup E^2 \cup E^3 \cup E^4 \cup E^5$. □

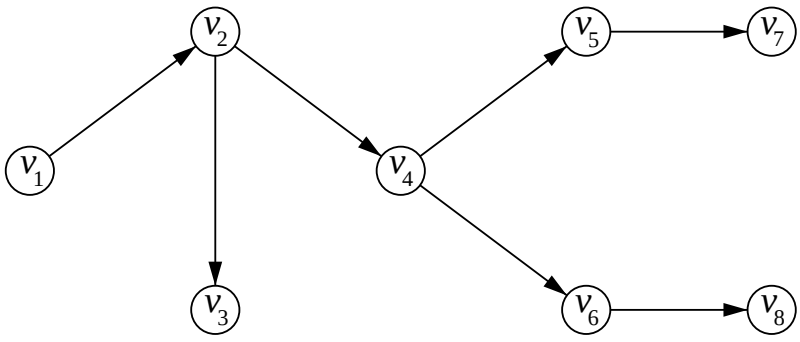


FIGURE 2.4: A spanning tree of graph G of Figure 2.1.

In the analysis of the algorithms, we assume a random access machine (RAM) model [10]. We use the Big Oh (O), the Big Omega (Ω), and the Big Theta (Θ) notations [46] to discuss the growth rate of a function that describes the execution time or the memory space required by a program. They are defined as follows:

Definition.

$$f(n) = O(g(n)) \Leftrightarrow (\exists c, n_0(c))(\forall n)(n \geq n_0 \Rightarrow f(n) \leq cg(n))$$

$$f(n) = \Omega(g(n)) \Leftrightarrow (\exists c, n_0(c))(\forall n)(n \geq n_0 \Rightarrow f(n) \geq cg(n))$$

$$f(n) = \Theta(g(n)) \Leftrightarrow (\exists c_1, c_2, n_0(c_1, c_2))(\forall n)(n \geq n_0 \Rightarrow c_1 g(n) \leq f(n) \leq c_2 g(n))$$

2.2 Defining the problem

In the *transitive closure problem*, we are given a directed graph $G = (V, E)$ and we should compute its transitive closure $G^+ = (V, E^+)$. This problem is called the *all-pairs transitive closure problem* [130] or the *full transitive closure problem*.

The input of the transitive closure problem may be in different forms. If not stated otherwise, we assume that the input graph is represented in the adjacency list form. Note that we require no data structure that contains the vertices adjacent to a vertex v, only the vertices adjacent from v, nor do we require any special ordering of the adjacency lists.

Also the output of the computation may be in different forms. If not stated otherwise, we assume that the output of the transitive closure problem is given as successor lists that represent the successor sets of the vertices.

2.2.1 Related problems

We list below problems that are closely related to the transitive closure problem. In the rest of the thesis, we only study the all-pairs transitive closure problem.

In the *reflexive transitive closure problem*, we are given a directed graph $G = (V, E)$ and we should compute its reflexive transitive closure. With only slight modifications a transitive closure algorithm can be applied to the reflexive transitive closure problem and vice versa.

In the *single-source transitive closure problem*, we are given a graph $G = (V, E)$ and a vertex x of V, whose successors we should compute. This problem can be solved by a simple graph search algorithm such as depth-first or breadth-first search. We start the search at vertex x and collect each vertex that the search reaches into the result set $Succ(x)$. In the *multi-source transitive closure problem*, we are given a graph $G = (V, E)$ and a subset $X \subseteq V$. We should compute the successors of the vertices in X. This problem can further be divided into *strong* and *weak multi-source transitive closure problems*. In the strong problem, we should compute its own successor set for each vertex of X. In the weak problem, we should compute the union of the successor sets of the vertices in X. Like the single-source problem, the weak multi-source problem can be solved by a graph search algorithm. The strong multi-source problem could be solved by first computing the transitive closure of the whole input graph and then selecting the appropriate successor sets, but computing directly the multi-source transitive

closure is apparently more efficient. Single-source and multi-source problems are sometimes called *partial transitive closure problems* as opposed to the full transitive closure problem.

The transitive closure problem is an example of a *closed semiring problem*. Closed semirings are algebraic structures that provide a unified approach to several seemingly unrelated problems of computer science and mathematics. Examples of closed semiring problems are graph theoretic path problems, such as computing the shortest or the most reliable path in a graph, data flow analysis problems in compiler technique, and inverting real matrices and solving systems of linear equations in mathematics [1, 10, 85, 119]. Similar algorithms can be used to solve all closed semiring problems, but algorithms that are specially designed for particular types of semirings are often more efficient. Therefore, we do not study transitive closure as a closed semiring problem.

Other generalizations of the transitive closure problem have been studied in the area of databases, e.g., [3, 30, 103, 113]. Typically, these generalizations extend the transitive closure to n-ary relations and allow path computations, selections, and aggregation of information.

2.3 Previous work

In this section, we describe the most important transitive closure algorithms presented in the literature. The descriptions are compact; for more details the reader should consult the original presentations. We also describe previous methods for representing the transitive closure.

2.3.1 Warshall's algorithm

The best-known transitive closure algorithm is Warshall's algorithm [129], which is presented in many textbooks, e.g., in [10, 11, 107]. Lehmann [85] points out that this algorithm was first described by Roy [104].

The idea in Warshall's algorithm is the following. If the graph contains paths $v \xrightarrow{*} w$ and $w \xrightarrow{*} u$ whose intermediate vertices come from a specific set S, then the graph also contains a path $v \xrightarrow{*} u$ such that its intermediate vertices come from the set $S \cup \{w\}$. Warshall's algorithm iterates from 1 to n and in the kth iteration studies paths whose intermediate vertices come from the set $\{v_1, \ldots, v_{k-1}\}$.

The algorithm is presented in Figure 2.5. The input to the algorithm is an adjacency matrix A representing the input graph G. The algorithm transforms A into an adjacency matrix A^+ representing the output graph G^+.

The worst-case execution time of Warshall's algorithm is $O(n^3)$ and the best-case execution time is $\Omega(n^2)$. This implies that Warshall's algorithm is not economical for a sparse input graph.

Warshall's algorithm examines the matrix position $A[i,k]$ column-by-column, but the matrix positions $A[i,j]$ and $A[k,j]$ row-by-row. Warren [128] noticed that we can change the

```
(1)   for k := 1 to n do
(2)       for i := 1 to n do
(3)           if i ≠ k and A[i,k] then
(4)               for j := 1 to n do A[i,j] := A[i,j] or A[k,j];
```

FIGURE 2.5: Warshall's algorithm

```
(1)   for i := 2 to n do
(2)       for k := 1 to i − 1 do
(3)           if A[i, k] then
(4)               for j := 1 to n do A[i, j] := A[i, j] or A[k, j];
(5)   for i := 1 to n − 1 do
(6)       for k := i + 1 to n do
(7)           if A[i, k] then
(8)               for j := 1 to n do A[i, j] := A[i, j] or A[k, j];
```

FIGURE 2.6: Warren's algorithm

algorithm to examine also the position $A[i,k]$ row-by-row if we split the algorithm into two passes. The first pass tests the positions below the main diagonal of the matrix and the second tests the positions above the main diagonal. We get the algorithm of Figure 2.6.

The worst-case and the best-case execution times of Warren's algorithm are $O(n^3)$ and $\Omega(n^2)$, just like in Warshall's algorithm. Both algorithms examine and set the same positions of the input matrix, but in a different order.

In the number of page faults generated, Warren's algorithm is better. Warren showed that if the adjacency matrix of a sparse graph does not fit into the main memory and if the LRU (least recently used) page replacement policy [117] is used, then his algorithm causes fewer page faults than Warshall's by a factor that is between one and $n/2$.

Warshall's and Warren's algorithms can easily be modified to solve the reflexive transitive closure problem or any other closed semiring problem. Lehmann [85] shows that Warshall's transitive closure algorithm, Floyd's algorithm for minimum-cost paths [42], Kleene's proof that every regular language can be defined by a regular expression [80], and the Gauss-Jordan method for inverting real matrices are different interpretations of the same basic program scheme.

2.3.2 Algorithms based on detecting the strong components

All vertices of a strong component have the same successor set. Purdom [101] presented the first transitive closure algorithm based on this fact. The algorithm consists of four parts:

1. Detect the strong components of the input graph G and build its condensation graph $\overline{G}$.

2. Using the partial order on the strong components induced by the edges of $\overline{G}$, sort the vertices of $\overline{G}$, i.e., the strong components of G, into a topological order.

3. Compute the transitive closure of $\overline{G}$. Start from the topologically greatest vertex of $\overline{G}$ and work back to the smallest vertex. To form the successor set of a vertex C of $\overline{G}$, combine the vertices adjacent from C and their successor sets. The topological order ensures that the successor sets of vertices adjacent from C are computed before the successor set of C.

4. Convert the transitive closure of $\overline{G}$ into the transitive closure of G. If vertex x is in component X and vertex y in component Y, insert y into $Succ(x)$ iff Y is in $Succ(X)$ in the transitive closure of $\overline{G}$.

The details of the algorithm are complicated: the listing is seven pages long. Purdom said that the first three parts can be combined.

Munro's algorithm [91] is also based on the detection of the strong components. It differs mainly from Purdom's algorithm in that Munro's algorithm uses matrix multiplication to construct the transitive closure of the condensation graph. We discuss Munro's algorithm in the next subsection below.

Purdom's and Munro's algorithms compute the strong components using virtually identical methods. Both traverse the graph in depth-first order. When a cycle is found, the vertices in the cycle are marked as being in the same strong component and the process is repeated. Two small strong components may be collapsed into a bigger one; the relabeling requires $O(n \log n)$ steps [118]. Thus, the worst-case execution time is $O(e+n \log n)$. Later, Tarjan [118] presented a strong component algorithm with $O(n + e)$ worst-case bound (and with smaller constant coefficient than in Purdom's and Munro's algorithms). If we use Tarjan's algorithm to detect the strong components, the worst-case execution time of Purdom's and Munro's algorithms is $O(n + e^+ + l(\overline{G}))$, where $l(\overline{G})$ is the time required to compute the transitive closure of condensation graph $\overline{G}$.

In Munro's algorithm $l(\overline{G})$ is $O(M(n))$, the time required to compute the product of two Boolean matrices (see section 2.3.3). In Purdom's algorithm $l(\overline{G})$ is $O(n^2 + n\overline{e}_r)$, where $\overline{e}_r$ is the number of edges in the transitive reduction of the condensation graph. Goralcikova and Koubek [45] presented an algorithm, which demonstrates that $l(\overline{G})$ is $O(n+e+n\overline{e}_r)$. The term $n\overline{e}_r$ represents the time needed to compute $\overline{e}_r$ unions of two successor sets having at most n elements. The term $n + e$ represents the time needed to topologically order the acyclic graph and its adjacency lists. Jaumard and Minoux [72] presented a slightly different version of the same algorithm and showed that $l(\overline{G}) = O(n+de^+)$ for acyclic graphs with in-degrees bounded by d.

Simon [111, 112] presented a transitive closure algorithm for acyclic graphs that is based on the *chain decomposition* of the topologically ordered graph. The chain decomposition is a division of the set of vertices V into a collection of sets C_i, each representing a path in the topologically ordered graph. A successor set is represented as a vector of length k, where k is the number of sets in the chain decomposition. In a vector representing the successor set S, the element at position i is the topologically smallest element of $S \cap C_i$. The chain decomposition and the topological order of the graph can be computed in $O(n + e)$ time. Also Simon's algorithm needs $\overline{e}_r$ unions of two successor sets. The union of two successor sets that are represented using the chain decomposition can be computed in $\Theta(k)$ time where k is the number of sets in the chain decomposition. Thus, Simon's algorithm shows that $l(\overline{G}) = O(n + e + k\overline{e}_r)$. We study the chain decomposition more thoroughly in Chapter 4.

Eve and Kurki-Suonio [40], Ebert [36], Schmitz [105], and Ioannidis et al. [62, 64] presented algorithms that use the strong components to compute the transitive closure without generating the condensation graph. All these algorithms are based on Tarjan's strong component algorithm. The worst-case execution time of these algorithms is typically $O(ne + n^2)$. We discuss these algorithms more thoroughly in Chapter 3 and compare them with our new transitive closure algorithms.

A weakness in transitive closure algorithms that are based on strong component detection is that these algorithms cannot solve other closed semiring problems. However, for transi-

tive closure computation these algorithms are in most cases faster and more practical than algorithms that can solve any closed semiring problem.

2.3.3 Algorithms based on matrix multiplication

Matrix multiplication and transitive closure computation are closely related. Let A be the $n \times n$ adjacency matrix of a graph $G = (V, E)$. Then $A^2 = A \wedge A$ is another $n \times n$ Boolean matrix such that $A^2[i,j] =$ **true** iff G contains a path $v_i \xrightarrow{*} v_j$ of length 2. More generally, matrix A^k is an $n \times n$ Boolean matrix such that $A^k[i,j] =$ **true** iff G contains a path $v_i \xrightarrow{*} v_j$ of length k. Only simple paths are needed in the construction of the transitive closure; thus A^+, the adjacency matrix of the transitive closure of G, can be written

$$A^+ = A \vee A^2 \vee \ldots \vee A^n \tag{2.2}$$

Furman [43] showed that A^+ can be computed in $O(\log n)$ iteration steps using the simple iteration formula

$$A_{k+1} = A_k \vee A_k^2 \tag{2.3}$$

$A \wedge B$ can be formed by representing **true** as one and **false** as zero and computing the matrix product of A and B over the ring of integers modulo $n+1$ and by normalizing non-zero entries to ones. Assuming that ordinary matrix multiplication takes $O(n^\alpha)$ time, Furman's algorithm computes the transitive closure in $O(n^\alpha \log n)$ time. Coppersmith and Winograd [29] showed that $\alpha \leq 2.376$.

An even closer connection between matrix multiplication and transitive closure computation was detected by Munro [91] and Fischer and Meyer [41], who proved the following result, which shows that matrix multiplication and transitive closure computation are computationally equivalent.

Theorem 2.1. Let $M(n)$ be a function satisfying $M(2n) \geq 4M(n)$ and $M(3n) \leq 27M(n)$. Then the transitive closure can be computed in $O(M(n))$ time iff the product of two arbitrary $n \times n$ matrices can be computed in $O(M(n))$ time.

Thus, the reflexive transitive closure can be computed in $O(n^\alpha)$ time where $\alpha \leq 2.376$. Munro [91] presented the following algorithm that manifests the theorem above.

1. Detect the strong components of the input graph G and build its condensation graph $\overline{G}$.
2. Using the partial order on the strong components induced by the edges of $\overline{G}$, sort the strong components into a topological order.
3. Build an adjacency matrix A corresponding to $\overline{G}$. Since $\overline{G}$ is topologically ordered, the matrix is in the upper triangular form.
4. Compute the transitive closure of A recursively using the following identity [41] where A (and hence A_{11} and A_{22}) are upper triangular:

$$A^* = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix}^* = \begin{pmatrix} A_{11}^* & A_{11}^* A_{12} A_{22}^* \\ 0 & A_{22}^* \end{pmatrix} \tag{2.4}$$

In step 4, we assume that A_{11}, A_{22}, and A_{12} are of order 2^i, which implies that A is of order 2^{i+1}. The algorithm can be generalized for an arbitrary matrix by padding it with zeros to make it of order 2^i. A_{11} and A_{22} represent two subgraphs S_1 and S_2 of $\overline{G}$, and A_{12} represents the connections between these subgraphs. The term $A_{11}^* A_{12} A_{22}^*$ above can be interpreted in the following way: a path $v \xrightarrow{*} u$, where v is in subgraph S_1 and u is in subgraph S_2, can be constructed by joining a path $v \xrightarrow{*} x$ that is entirely in S_1 with a path $y \xrightarrow{*} u$ that is entirely in S_2 by an edge (x, y).

Steps 1–3 take $O(n^2)$ time. Munro showed that step 4 requires $O(m^\alpha)$ operations, where m is the number of strong components and an ordinary matrix multiplication takes $O(m^\alpha)$ time. Thus, the total execution time is $O(n^\alpha)$ time in the worst case.

Arlazov et al. [14] presented a method (called the "four Russians" algorithm and apparently due to Kronrod [127]) for multiplying two Boolean matrices and computing the reflexive transitive closure in $O(n^3 / \log n)$ steps, both in the worst case and on the average [126]. O'Neill and O'Neill [97] gave another algorithm for this problem that runs in $O(n^2)$ expected time. Adleman et al. [2] presented a way to reduce the number of bit operations required in Boolean matrix multiplication and transitive closure.

Aho et al. generalized Theorem 2.1 by showing that when the scalars come from any closed semiring, the product of two matrices is computationally equal to the closure of one matrix (see Theorem 5.6, page 202, in [10]).

Although the transitive closure algorithms based on matrix multiplication have good asymptotic time bounds, they are not practical, since the constant factors are high and since they use the adjacency matrix representation, which is uneconomical for sparse graphs.

2.3.4 Transitive closure computation in databases

The relational data model [123] is currently the most popular data model in commercial database systems. Relational databases use relational query languages, which are based on relational algebra or, equivalently, on first-order relational calculus, both introduced by Codd [27, 28]. A relational query language consists of a small set of simple declarative operators on relation tables corresponding to mathematical relations. Complicated queries can be expressed by combining simple relational operators. An essential element in the success of relational query languages has been that the relational operators can be implemented efficiently.

Recently, it has become clear that the relational data model lacks the expressiveness that is needed in many modern applications, for instance in the areas of artificial intelligence, CAD, and software engineering. The relational model has two main shortcomings. First, the model is value oriented, whereas in many applications the natural way to represent data is object oriented. Object oriented data models [23, 123] have therefore become popular in recent years, and much research has been done in this area (see for instance [16, 23, 132, 133]). Second, relational query languages cannot express any kind of recursion. Even the simplest kind of recursive query, the transitive closure of a binary relation, cannot be expressed in relational algebra [13].

Several proposals for introducing recursion to relational languages have been presented. Zloof [134] suggested augmenting relational algebra with a transitive closure operator. Aho and Ullman [13] proposed augmenting relational algebra with a least fixed point operator that

enables a wider class of recursive queries than the transitive closure operator. Rosenthal et al. [103] presented what they call "traversal recursion," i.e., recursive queries that model traversal of a directed graph. Agrawal [3] proposed a so called *alpha* operator that enables generalized linear recursive queries. Sippu and Soisalon-Soininen [113] presented a generalized transitive closure operator based on a composition operator that is generalized to n-ary relations. Cruz and Norvell [30] presented "aggregative closure," i.e., generalized transitive closure with path computations, e.g., the computation of the shortest paths between vertices, and aggregation of information, e.g., computing averages or sums of fields of relations. Eder [37] and Dar and Agrawal [31, 32] extended the popular SQL query language with similar generalized transitive closure operators. The ANSI/SQL standards committee also proposed the inclusion of a closure operator (called "recursive union") into the query language SQL [39, 109, 110, 115].

The approaches to introducing general recursive queries have mostly been based on logic programming. The Datalog query language [24, 123] is the best-known example. Although much research has been done in the area of optimizing general recursive queries (see, e.g., [15, 17, 123]), it seems that general recursion cannot be implemented efficiently. On the other hand, it seems that most recursive queries that occur in practical applications are transitive [15, 68]. Further, Jagadish et al. [68] showed that all linear recursive queries can be expressed using transitive closure possibly preceded and followed by operations available in relational algebra. Therefore, the recent approaches to introduce recursion to relational algebra have concentrated on transitive closure in its various forms.

Several transitive closure algorithms for database environments have been proposed. These can be divided into *iterative*, *matrix-based*, *graph-based*, and *hybrid* algorithms [31].

Iterative algorithms

The iterative algorithms compute the transitive closure of a relation R by evaluating the least fixed point of the relational equation [13]:

$$R^+ = R^+ \circ R \cup R \tag{2.5}$$

A simple algorithm for computing the least fixed point is presented in Figure 2.7. The algorithm is called *semi-naive* in [15].

```
(1)   R+ := R;
(2)   Δ := R;
(3)   do
(4)       Δ := Δ ∘ R − R+
(5)       R+ := R+ ∪ Δ
(6)   while Δ ≠ ∅
```

FIGURE 2.7: The semi-naive algorithm.

Here R, R^+, and Δ are variables that contain relations. After the ith iteration, the variable Δ contains the new tuples that were generated in the ith iteration.

Lu [86] presented a version of the semi-naive algorithm that uses two strategies to speed up the computation. First, the algorithm reduces the input relation R dynamically after each iteration. It removes from R each tuple (x, y) with no tuple (z, x) in Δ, since no such tuple

(x, y) would yield any new tuples to R^+. Second, Lu's algorithm hashes the tuples of Δ on their second argument and the tuples of R on their first argument. The algorithm takes a pair of has buckets (one bucket of Δ and one bucket of R) that it can hold in the main memory simultaneously and generates all tuples derivable from those buckets.

If $R \subseteq V \times V$ and $|V| = n$, the semi-naive algorithms needs at most n iterations. The maximum number of iterations is needed, for instance, when the graph $G = (V, R)$ consists of a simple cycle of n elements. Ioannidis [61] and Valduriez and Boral [124] presented algorithms that need only a logarithmic number of iterations. The logarithmic algorithms can be derived from the following equation defining the transitive closure:

$$R^+ = \sum_{k=1}^{\infty} R^k = R \circ \prod_{k=1}^{\infty} (I \cup R^{2^k}) = R \circ (I \cup R) \circ (I \cup R^2) \circ (I \cup R^4) \ldots \tag{2.6}$$

If the domain of R contains n elements, the highest k in R^{2^k} that we need is $\log(n+1) - 1$. Thus, the logarithmic algorithm presented in Figure 2.8 computes the transitive closure in $\log(n+1) - 1$ iterations.

(1) $R^+ := R$;
(2) $\Delta := R$;
(3) $\delta := R$
(4) **do**
(5) $\quad \delta := \delta \circ \delta$;
(6) $\quad \Delta := R^+ \circ \delta$;
(7) $\quad R^+ := R^+ \cup \Delta \cup \delta$
(8) **while** $\Delta \neq \emptyset$

FIGURE 2.8: The logarithmic algorithm.

After the ith iteration, variable δ contains R^{2^i}, i.e, the tuples corresponding to paths of length 2^i in the graph $G = (V, R)$. Variable Δ contains tuples corresponding to paths of lengths $2^i + 1 \ldots 2^{i+1} - 1$.

It is interesting to note that the logarithmic algorithm uses similar ideas as Furman's algorithm [43] (see section 2.3.3) that is based on matrix multiplication.

Qadah et al. [102] presented iterative algorithms for solving weak multi-source transitive closure problems. The algorithms resemble the semi-naive algorithm described above.

A benefit of iterative algorithms is their generality. They can evaluate all kinds of recursive queries, not just transitive closures. They can solve single-source and multi-source transitive closure problems. They can also solve generalized transitive closure problems involving path computations and aggregation. On the other hand, the generality of iterative algorithms makes them slower than the matrix-based, graph-based, and hybrid algorithms that we describe below.

Matrix-based algorithms

The matrix-based transitive closure algorithms for database environments are usually developed from Warshall's or Warren's algorithm (see section 2.3.1). The term "matrix-based" refers to the approach of processing the input, which resembles Warshall's and Warren's algorithms, not to the data representation. Most of these algorithms use adjacency and successor lists or relation tables.

Lu et al. [87] presented an adaptation of Warren's algorithm that processes relation tables instead of adjacency matrices. The algorithm, shown in Figure 2.9, first sorts the tuples of relation R lexicographically by both their fields. Then the tuples are scanned in two passes that correspond to the two passes of Warren's algorithm.

```
(1)   T := R sorted by both its fields;
(2)   for (x, y) ∈ T in sorted order do
(3)       if x > y then
(4)           T := T ∪ {(x, y)} ∘ T;
(5)   for (x, y) ∈ T in sorted order do
(6)       if x < y then
(7)           T := T ∪ {(x, y)} ∘ T
```

FIGURE 2.9: An adaptation of Warren's algorithm by Lu et al. [87].

Agrawal et al. [5, 6] described what they call *Warshall-derived* algorithms. These algorithms do the same operations as Warshall's algorithm, but in a different order. Warren's algorithm is an example of a Warshall-derived algorithm. Agrawal et al. presented several Warshall-derived algorithms that are suitable for a database environment. These algorithms represent the input and the output by lists and use blocking to reduce I/O costs. Lists are read into the main memory and processed one block at a time. The way of dividing the lists into blocks and the order of processing the blocks differs in the algorithms.

Ullman and Yannakakis [122] presented another matrix-based algorithm, called the *Grid* algorithm in [8]. If s is the size of the main memory, the algorithm splits the $n \times n$ adjacency matrix A into n^2/s submatrices $A_{i,j}$ of size $\sqrt{s} \times \sqrt{s}$. Then the transitive closure is computed using the code in Figure 2.10.

```
(1)   for k := 1 to n/√s do begin
(2)       A_{k,k} := A*_{k,k};
(3)       for i := 1 to n/√s do
(4)           for j := 1 to n/√s do
(5)               A_{i,j} := A_{i,j} + A_{i,k} × A_{k,k} × A_{k,j};
```

FIGURE 2.10: The Grid algorithm.

Here $A^*_{k,k}$ is the reflexive transitive closure of the sub-matrix $A_{k,k}$. Since the algorithm is designed for dense graphs, it uses the adjacency matrix representation. Ullman and Yannakakis showed that if $s \leq n^2$, the algorithm needs $O(n^3/\sqrt{s})$ I/O operations in the worst case. The blocking Warshall-derived algorithms by Agrawal et al. [5, 6] need $O(n^4/s)$ I/O operations in the worst case; thus, the Grid algorithm is more efficient. The problem in the Grid algorithm is that all sub-matrices must be of the same size, which may lead to underutilization of the memory if the input is sparse.

Matrix-based algorithms are not as general as iterative algorithms. They can only be used for computing transitive closures, not general recursive queries. Matrix-based algorithms always compute the full closure and thus cannot be used to solve single-source and multi-source transitive closure problems efficiently. On the other hand, they can be used to solve generalized transitive closure problems involving path computations and aggregation.

Graph-based algorithms

The graph-based algorithms consider the input relation as a directed graph and use graph search to compute the transitive closure. Ioannidis et al. [62, 63, 64] presented two transitive closure algorithms called BTC and GDFTC that are based on detecting the strong components of the input graph and that are designed for a database environment. We discuss these algorithms in Chapter 3, and compare them with the new transitive closure algorithms that we have developed.

Jiang [73] presented an algorithm that uses a combination of depth-first and breadth-first traversal to compute single-source and multi-source transitive closures of database relations. The algorithm reduces the input graph heuristically during the traversal. If a vertex v is not in the set of source vertices S, whose successors we are computing, and only one vertex u adjacent to v exists, the successor set of v is not needed. Instead, vertex v can be reduced to a sink vertex by inserting an edge from u to each vertex adjacent from v and deleting all edges leaving v.

Toroslu and Qadah [121] presented a graph-based algorithm for computing strong multi-source transitive closures. Their algorithm uses a representation that stores the combined closure, i.e., all vertices reachable from the source vertices. Each element of the combined closure is tagged with the set of source vertices whose successor the vertex is. The algorithm uses a depth-first traversal for generating an initial representation of the combined closure. The depth-first traversal is followed by a backward and a forward propagation phase that compute the tags associated with the vertices of the combined closure. Toroslu and Qadah presented performance evaluations indicating that their algorithm is more efficient than the semi-naive algorithm and Jiang's algorithm [73] described above.

Graph-based algorithms cannot be used to solve generalized transitive closure problems involving path computations and aggregation when the input graphs are cyclic, since strong component detection loses path information. If the input graphs are acyclic, it is possible to adapt graph-based algorithms for generalized transitive closure problems [64]. Yannakakis [130] shows how linear recursive queries can be efficiently evaluated by graph traversal.

Hybrid algorithms

The hybrid transitive closure algorithms combine ideas from iterative, matrix-based, and graph-based algorithms.

Agrawal and Jagadish [8] presented a family of hybrid algorithms that compute the condensation graph of the input graph and sort it topologically in the first pass. In the second pass, the algorithms compute the transitive closure of the topologically sorted graph by a method resembling Warren's algorithm. Blocking is used in the second pass as in matrix-based algorithms. The second pass is a breadth-first algorithm, whereas the graph-based algorithms like BTC described above are depth-first algorithms. Blocking can be used more efficiently with breadth-first algorithms than with depth-first algorithms [74]. Note however, that computing the condensation graph in the first pass of the algorithm requires depth-first search if Tarjan's algorithm [118] is used or a combination of breadth-first and depth-first search if Jiang's algorithm [75] is used.

Jakobson [70, 71] and Dar and Jagadish [33] presented hybrid transitive closure algorithms

that use *successor trees* for representing the successor sets. In addition to the vertices of the successor set, a successor tree contains the paths that were used to obtain these vertices. Using the path information the algorithms detect multiple paths between a pair of vertices and avoid adding the same successors twice to a successor set. We discuss the successor tree representation more thoroughly in section 4.4. The algorithm that Jakobson presented in [70] is an adaptation of the semi-naive algorithm and the algorithm he presented in [71] is an adaptation of Warshall's algorithm. The algorithm that Dar and Jagadish presented in [33] is an adaptation of the hybrid algorithm by Agrawal and Jagadish [8].

These hybrid algorithms can be used to solve single-source and multi-source transitive closure problems as well as generalized transitive closure problems involving path computations and aggregation.

Parallel and distributed algorithms

Some articles discuss the computation of the transitive closure of a database relation in parallel and distributed environments, see [22, 25, 26, 48, 49, 51, 52, 53, 54, 55, 57, 76, 116, 125]. We do not describe these articles here, since we are only interested in centralized, sequential transitive closure computation in this thesis.

Comparisons between the algorithms

In the articles described above, the comparisons between the algorithms are mostly based on empirical performance measurements or simulations. We describe these performance studies below.

Ioannidis [61] compared the logarithmic algorithm and the semi-naive algorithm in computing the transitive closure of small lists and trees. The performance metrics were disk I/O in number of pages and CPU-time. The logarithmic algorithm was more efficient than the semi-naive algorithm in most inputs, both in the disk I/O and in the CPU-time. Valduriez and Boral [124] reported results indicating also that logarithmic algorithms are more efficient than the semi-naive algorithm.

In a more recent study, Kabler et al. [77] compared the semi-naive algorithm, the logarithmic algorithm and the Blocked Warren algorithm, which is one of the Warshall-derived algorithms [6], in computing the transitive closure of randomly generated trees and graphs. The performance metrics were disk I/O in number of pages and an aggregate CPU-time (a combination of the measured CPU-time plus a computed CPU-time for the disk operations). The Blocked Warren algorithm was usually superior to the other two algorithms. Contrary to the results of Ioannidis [61] and Valduriez and Boral [124], the semi-naive algorithm was usually superior to the logarithmic algorithm.

Lu [86] compared his improved semi-naive algorithm, the original semi-naive algorithm, and the logarithmic algorithm in computing the transitive closure of small lists and binary trees. The performance metric was disk I/O in number of tuples. The method of study was simulation, not direct measurement as in the other comparisons described above. The semi-naive algorithm augmented with the improvements proposed by Lu was more efficient than the other two algorithms.

Agrawal and Jagadish [6] compared three Warshall-derived algorithms and a logarithmic

algorithm in computing the transitive closure of randomly generated graphs. The performance metric was disk I/O in kilobytes. The Blocked Warren algorithm was the most efficient of the Warshall-derived algorithms and it was also superior to the logarithmic algorithm.

Agrawal et al. [5] compared two Warshall-derived algorithms and the semi-naive algorithm in computing the transitive closure, the bill of materials, and the shortest paths of randomly generated graphs. The performance metrics were disk I/O in number of pages and CPU-time. The Warshall-derived algorithms were roughly equally efficient, but the semi-naive algorithm was much slower.

Lu et al. [87] compared their adaptation of Warren's algorithm with two versions of the logarithmic algorithm in computing the transitive closure. The performance metric was the execution time. No measurements were done, instead the performance metrics were evaluated using analytic formulas. Inputs and outputs were characterized using the size of the source relation and the output relation, selectivity of join operations, and the number of iterations as parameters. The adapted Warren's algorithm was more efficient than the other algorithms when the main memory was not much smaller than the input relation and the path lengths in the inputs varied greatly. Otherwise, the logarithmic algorithms were more efficient.

Agrawal and Jagadish [8] compared their hybrid algorithm with a Warshall-derived algorithm, a graph-based algorithm presented in [62] and the Grid algorithm by Ullman and Yannakakis [122] in computing the bill of materials of randomly generated acyclic graphs. The performance metric was disk I/O in number of tuples. The hybrid algorithm was in all inputs more efficient than the other algorithms.

Ioannidis et al. [63, 64] compared their graph-based algorithms BTC and GDFTC with Schmitz's graph-based algorithm [105] in computing the transitive closure of randomly generated graphs. The performance metrics were disk I/O in kilobytes and an aggregated CPU-time. Algorithm BTC was in most inputs more efficient than the other two algorithms.

Dar and Jagadish [33] compared their spanning tree transitive closure algorithm with the hybrid algorithm by Agrawal and Jagadish [8] in computing the transitive closure of randomly generated acyclic graphs. The performance metric was disk I/O in number of tuples. The spanning tree algorithm was more efficient than the hybrid algorithm by Agrawal and Jagadish in all inputs.

Dar and Ramakrishnan [31, 34] compared several algorithms in computing full and partial transitive closures of randomly generated acyclic graphs. The algorithms were BTC by Ioannidis et al. [63, 64], the hybrid algorithm by Agrawal and Jagadish [8], Jiang's algorithm [73], the spanning tree algorithm by Dar and Jagadish [33], and two algorithms by Jakobson [71]. The main performance metric was disk I/O in number of pages. The hybrid algorithm and the spanning tree algorithm were less efficient than the other algorithms. The relative efficiency of the algorithms varied in different inputs and no algorithm was generally superior to the other algorithms. In addition to measuring disk I/O in number of pages, Dar and Ramakrishnan measured disk I/O in number of tuples and successor lists, and the total number of generated tuples. They detected that none of these other metrics reliably predicted disk I/O in pages.

To summarize, the performance studies discussed above do not give a clear picture of the relative efficiency of the algorithms. In most of the studies, the iterative algorithms were less efficient than the matrix-based algorithms, and the matrix-based algorithms were less efficient than graph-based and hybrid algorithms.

Note that all these studies are poor performance evaluation studies. The system that was evaluated and the goal of the study were not properly explained. All system parameters affecting the performance were not identified, and the most important system parameters, like the size of the main memory and the number of buffer pages, were in most studies not varied to find out their effect on the performance.

The most severe problems in these studies were in the selection of the inputs and in the analysis of the measurement data. In some studies, only trees and lists were used as inputs; in all studies, the inputs were small. The random nature of the inputs was not understood in any study. For each combination of input parameters, only a couple of random inputs were generated. The variability of the results that is unavoidable when random inputs are used was not considered in any of these studies. Thus, the studies gave no information on the accuracy of the results.

Although the input graphs usually had only a few hundred vertices, the number of disk I/O operations was large, which indicates that each page was moved several times between the main memory and the disk. The studies did not answer the question of what happens if real-life databases are used as inputs.

2.3.5 Representations and dynamic updates

Most of the transitive closure algorithms described above use adjacency lists, adjacency matrices, or relation tables to represent the closure. However, some algorithms that we described use special representations to speed up the computation. Simon's transitive closure algorithm [111, 112] for acyclic graphs obtains a small worst-case time by using the chain decomposition, which speeds up union operations. The hybrid algorithms by Dar and Jagadish [33] and Jakobson [70, 71] use the successor tree representation, which aims at avoiding unnecessary successor set operations. In this subsection, we study other special representations for transitive closure.

Special data representations for transitive closure computation have been designed for both main memory and a database environment. In main memory representations, the goals have been to speed up successor set operations and to avoid redundant operations. In database representations, the goals have been to speed up the traversal of the input graph by special file structures and to materialize the transitive closure, i.e., to store the computed closure so that queries about connectivity of vertices can be efficiently evaluated. Dynamic maintenance of the representation, when edges are inserted to the input graph or deleted from it, is associated with both main memory and database representations.

We study first how the transitive closure can be dynamically maintained in a main memory environment. Next, we study special file structures that speed up the traversal of the input graph in a database environment. Finally, we study transitive closure materialization in a database environment.

Dynamic updates in main memory

Ibaraki and Katoh [58] presented in their seminal article, algorithms for the dynamic maintenance of transitive closure when edges are inserted to the input graph or deleted from it. The algorithms use no special data structures; the transitive closure is represented by an adjacency matrix. The algorithm that updates the closure between edge insertions requires $O(n^2)$ time

in the worst case to process one edge insertion, but only $O(ne^+)$ time in the worst case to process e edge insertions. This is a typical property of algorithms for dynamic maintenance of transitive closures: the worst-case time to process one edge insertion or deletion is greater than the amortized time [120] over a sequence of several insertions or deletions. Ibaraki's and Katoh's algorithm that updates the closure between edge deletions requires $O((n+e)e^+)$ time in the worst case to process e deletions.

Italiano [65] showed that faster updating between edge insertions is possible if a specialized data structure is used. The data structure contains the successor sets organized as a tree of paths and an $n \times n$ matrix of pointers pointing to the tree. The data structure makes it possible to update the closure between e consecutive edge insertions in $O(ne)$ worst-case time. It is also possible to return a path connecting two vertices in time proportional to the length of the path. Buchsbaum et al. [21] extended Italiano's results to edge labeled graphs. In another paper, Italiano [66] showed that the closure of an acyclic graph can be updated between e edge deletions in $O(ne)$ worst-case time when a modified version of the data structure is used.

La Poutré and van Leeuwen [81] studied the dynamic maintenance of both the transitive reduction and the transitive closure of a graph between edge insertions or deletions. They presented algorithms and data structures that resemble those by Italiano. The algorithm that processes edge insertions has the same worst-case time $O(ne)$ as Italiano's insertion algorithm. The algorithm that processes edge deletions has the same worst-case bound as Italiano's deletion algorithm when the inputs are acyclic. Unlike Italiano's algorithm, La Poutré's and van Leeuwen's algorithm can also process deletions in cyclic graphs; the worst-case time of processing e deletions is slightly greater than $O(ne)$.

Yellin [131] presented algorithms and data structures similar to those by Italiano, La Poutré and van Leeuwen. Also Yellin's algorithms have the $O(ne)$ worst-case time bound. However, Yellin showed that if the outdegree of the graph is bounded by d, his algorithms have another worst-case bound $O(de^+)$, which is optimal for bounded degree graphs. The other algorithms [58, 65, 66, 81] do not have this bound.

Special representations for supporting graph traversal in databases

When the input graph resides in the disk, the paging caused by the traversal of the input is a bottleneck of the computation. The order in which the graph is stored in the disk affects the performance. A random order causes the read and write heads of the disk to skip randomly. If the graph does not fit into the main memory, each page may be moved many times from the disk to the main memory.

Larson and Deshpande [82] presented a simple file structure for supporting traversal recursion [103] in acyclic graphs. The vertices and the edges may have labels, permitting path computations. The representation consists of two files: a main file and an index file. The main file contains one record per each vertex. The record contains the label of the vertex and the successors and predecessors of the vertex. The successors and predecessors are represented as vertex-label pairs. The records are sorted in a topological order and stored in a B-tree. The topological order of the records enables finding all successors or predecessors of a set of vertices in a single scan over the main file. The index file maps the vertex names to their topological order keys, which are used to locate the records in the B-tree. Larson and Deshpande presented algorithms for updating the file structures when vertices or edges are inserted, deleted,

or modified. Adding an edge is the only operation that may render the topological order invalid and force a rearrangement of some records in the main file. The other maintenance operations are simple and fast.

The graph traversal algorithm that Larson and Deshpande presented uses a priority queue to manage the vertices to be visited. If the outdegrees of the vertices are large, the priority queue may be too large to fit into the main memory and the queue management may cause much disk I/O. Hua et al. [56] presented a similar file structure, called *connectivity index*, that also enables graph traversal during a single scan over the file. However, the traversal of the connectivity index is controlled by a simple FIFO (first in, first out) queue that is much more efficient in disk I/O than the priority queue.

Agrawal and Kiernan [9] presented a file structure, called the *traversal index*, that resembles those of Larson and Deshpande and Hua et al. except that it tolerates cyclic input graphs. The traversal index is kept in a topological order if the input graph is acyclic and in a relaxed topological order if the input graph is cyclic. Agrawal and Kiernan presented an algorithm for creating the traversal index and algorithms for maintaining the index when the input graph is changed and showed how the input graph can be traversed during a single scan over the traversal index.

Special representations for materializing transitive closure in databases

Jagadish [67] studied the chain decomposition as a way to materialize transitive closures. The difference from Simon's representation is that Jagadish represented a successor set S as a list of pairs (C_i, v_i), where C_i is a chain and v_i is the topologically smallest element of $S \cap C_i$. Simon represented S as a k element vector $[v_1, v_2, \ldots, v_k]$ where v_i is the topologically smallest element of $S \cap C_i$. Thus, the time needed to union two successor sets may be smaller than $\Omega(k)$ in Jagadish's representation, but inserting a vertex into a successor set and testing the presence of a vertex in a set require $\Omega(k)$ time in the worst case.

Jagadish showed that finding an optimal chain decomposition for an acyclic graph is a minimum flow problem. He then presented several heuristics for computing the decomposition and compared them empirically. The tests indicated that a decomposition method that is based on topological ordering yields the smallest number of chains. This is exactly the method that Simon [111, 112] proposed, but Jagadish did not directly refer to Simon's work. Jagadish also presented how to maintain the chain decomposition and the successor sets when the input graph is modified.

Agrawal et al. [4] presented another compact representation for materializing the transitive closure of an acyclic graph. This representation is based on intervals of integers. A successor set is a collection of intervals $\{I_1, I_2, \ldots, I_m\}$, where each interval $I = (i, j)$ represents integers $i, i+1, \ldots, j$. A successor set S can typically be represented using only a few intervals. Thus, unioning two successor sets, inserting vertices to successor sets, or checking the presence of vertices in successor sets is typically much faster than in a successor list representation. The worst-case bounds for these operations are, however, the same as in the successor list representation. The vertices can be numbered in several ways and different numberings require different numbers of intervals to represent the successor sets. Agrawal et al. presented an algorithm for computing an optimal numbering under certain conditions. They also showed how to maintain the representation when the input graph is modified.

We discuss both the interval representation and the chain representation more thoroughly in Chapter 4, where we present another interval representation and another chain representation, which can be efficiently used with cyclic input graphs.

The methods for materializing transitive closures by Jagadish [67] and Agrawal et al. [4] cannot be used for generalized transitive closures, since path information cannot be associated with the chain decomposition or with intervals. Full materialization of path information is infeasible, since the number of possible paths grows exponentially to the number of vertices in the graph.

In some papers, partial materialization of path information is studied. Agrawal and Jagadish [7] presented a representation for partially materializing path information between pairs of vertices. The representation associates to each vertex v a set of tuples of form (w, x, L), where w is a vertex that is reachable from v, x is the first vertex in the path leading to w, and L is the label associated to that path. Agrawal and Jagadish showed how the representation is created and maintained when the input graph is modified and how path computations can be efficiently evaluated using the representation.

Guh et al. [47, 48] presented another representation for partially materializing paths between pairs of vertices in an acyclic graph. This representation is based on tuples of form (v, w, G_{vw}, g_{vw}), where v and w are vertices connected by a path $v \xrightarrow{*} w$, G_{vw} is the number of different paths between v and w, and g_{vw} is **true** iff (v, w) is an edge of the input graph. Associated with each vertex v are all tuples containing v (either as the source or the target of the path). Guh et al. presented both sequential and parallel algorithms for creating and managing this representation.

The drawback of both the representation of Agrawal and Jagadish [7] and the representation by Guh et al. [47, 48] is their great size. The representation by Agrawal and Jagadish needs $\Omega(dn^2)$ tuples in the worst case, where d is the maximum outdegree of a vertex. This kind of memory requirement seems to be more a rule than an exception in this representation. The representation by Guh et al. needs $\Omega(n^2)$ tuples in the worst case, but even this is too much if n is large. Thus, it seems that even partial materialization of path information is not practical when the number of vertices is large.

Chapter 3

Transitive closure computation using strong components

Strong components are probably the most important source of redundant computations in transitive closure algorithms that do not detect them [44]. Some transitive closure algorithms presented in the literature are based on detecting the strong components of the input graph, see [36, 40, 62, 64, 91, 101, 105]. Most of these algorithms are based on Tarjan's strong component algorithm [118]. The problem with these algorithms is that they either generate one partial successor set for each vertex of the component or scan the whole input graph more than once.

In this chapter, we present new transitive closure algorithms that solve these problems. First, we review Tarjan's strong component algorithm and present two improved versions of it. Then, starting from a simple adaptation of Tarjan's algorithm to transitive closure computation, we develop two new efficient transitive closure algorithms. Finally, we compare the new algorithms with the previous ones presented in the literature.

3.1 Strong component detection – Tarjan's algorithm

Tarjan [118] presented an elegant algorithm that finds the strong components in $\Theta(n + e)$ time, where n is the number of vertices and e is the number of edges in the input graph. We review here the basic ideas of Tarjan's algorithm. Our aim is not simply to duplicate existing material, but to give a basis for describing and analyzing the transitive closure algorithms we have designed and the improvements on Tarjan's algorithm that we have made. We use a notation that differs from the original presentation [118], but that simplifies the description of the algorithm and its analysis.

Tarjan's algorithm contains two interleaved traversals of the graph. First, a depth-first search traverses all edges and constructs a depth-first spanning forest. Second, once a so called *root* of a strong component is found, all its descendants that are not elements of previously found components are marked as elements of this component. This second traversal is implemented by using a stack, where each vertex is stored when entered by the depth-first search. Before the root of a component is exited, all vertices down to the root are removed from the stack and they form the component in question.

Tarjan's algorithm is presented in Figure 3.1. It consists of a recursive procedure VISIT and a main program that applies VISIT to each vertex that has not already been visited. VISIT

```
(1)   procedure VISIT(v);
(2)   begin
(3)       Root(v) := v; Comp(v) := Nil;
(4)       PUSH(v, stack);
(5)       for each vertex w such that (v, w) ∈ E do begin
(6)           if w is not already visited then VISIT(w);
(7)           if Comp(w) = Nil then Root(v) := MIN(Root(v), Root(w))
(8)       end;
(9)       if Root(v) = v then begin
(10)          create a new component C;
(11)          repeat
(12)              w := POP(stack);
(13)              Comp(w) := C;
(14)              insert w into component C;
(15)          until w = v
(16)      end
(17)  end;
(18)  begin /* Main program */
(19)      stack := ∅;
(20)      for each vertex v ∈ V do
(21)          if v is not already visited then VISIT(v)
(22)  end.
```

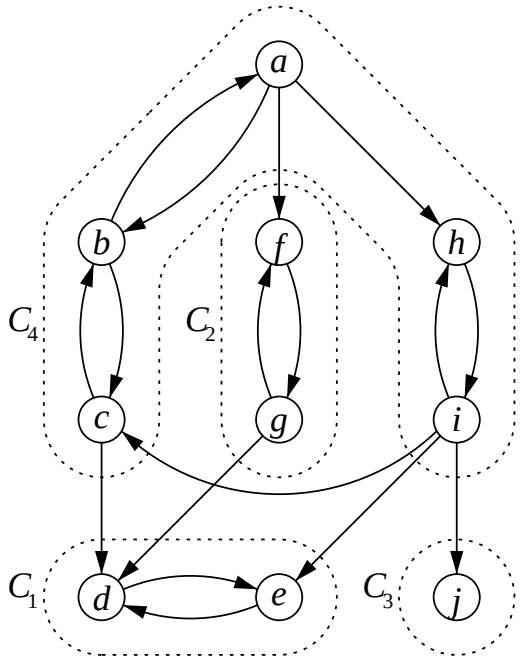
FIGURE 3.1: Tarjan's algorithm detects the strongly connected components of graph $G = (V, E)$.

enters the vertices of the graph in depth-first order. For each strong component C, the first vertex of C that VISIT enters is called the *root* of component C. An important task of the algorithm is to find the component roots. For this purpose, we define a variable $Root(v)$ for each vertex v. When VISIT is processing vertex v, $Root(v)$ contains a candidate vertex for the root of the component containing v.

Initially, at line 3, vertex v itself is the root candidate. When VISIT processes the edges leaving vertex v at lines 5–8, new root candidates are obtained from children vertices that belong to the same component as v. The MIN operation at line 7 compares the vertices using the order in which VISIT has entered them, i.e., $\text{MIN}(x, y) = x$ if VISIT entered vertex x before it entered vertex y, otherwise $\text{MIN}(x, y) = y$. A simple way to implement this is to use an array and a counter to assign a unique depth-first number to each vertex. When VISIT has processed all edges leaving v, $Root(v) = v$ iff v is the root of the component containing v (line 9). Note however, that if v is not a component root, we do not know whether $Root(v)$ is the right root of the component containing v.

To distinguish between vertices belonging to the same component as vertex v and vertices belonging to other components, a variable $Comp(w)$ is defined for each vertex w. Its initial value is *Nil*. When a component C is detected, VISIT sets $Comp(w) := C$ for each vertex w that belongs to C (line 13). An auxiliary stack is used for this purpose. Each vertex is stored onto the stack in the beginning of VISIT. When the component is detected, the vertices belonging to it are on top of the stack. VISIT removes them from the stack, sets their $Comp(w)$

(a)



(b)

Vertex	Adjacency list
a	(b, f, h)
b	(c, a)
c	(d, b)
d	(e)
e	(d)
f	(g)
g	(f, d)
h	(i)
i	(h, j, e, c)
j	$()$

FIGURE 3.2: **(a)** Graph G with four strong components. **(b)** The adjacency lists of G.

variables, and inserts them into component C.

Example 3.1. Consider the graph in Figure 3.2(a). It consists of four strong components $C_1 = \{d, e\}$, $C_2 = \{f, g\}$, $C_3 = \{j\}$, and $C_4 = \{a, b, c, h, i\}$, which we have encircled. In Figure 3.3, we present the trace of one possible application of Tarjan's algorithm to the graph. We assume that the execution starts at vertex a. Tarjan's algorithm processes the vertices adjacent from a vertex v in the order in which they appear in the adjacency list of v. The adjacency lists are presented in Figure 3.2(b). The trace lists the operations that Tarjan's algorithm does, i.e., entering and exiting vertices (lines 2 and 17 of Tarjan's algorithm), modifying the values of the *Root* and *Comp* variables (lines 3, 7, and 13 of Tarjan's algorithm), creating new components (line 10 of Tarjan's algorithm), and storing vertices onto and removing them from *stack* (lines 4 and 12 of Tarjan's algorithm). We also present the current state of *stack* at any given moment. □

Besides Tarjan's algorithm, another linear time strong component algorithm is presented in many textbooks. The algorithm is attributed in [11] to R.Kosaraju and published in [108]. This algorithm is inferior to Tarjan's algorithm, since it requires one depth-first traversal of the input graph and another traversal of the graph obtained by reversing the edges of the input graph.

3.1.1 Analysis

We prove now the correctness of Tarjan's algorithm. The proof is rather detailed and differs from the original presentation in [118]. It serves as a basis for the correctness proofs of the new algorithms that we present later in this chapter. First, we give some definitions that

Operation	Stack	Operation	Stack
enter(a)		POP$(stack)$	(f,c,b,a)
Root$(a) := a$		*Comp*$(g) := C_2$	(f,c,b,a)
PUSH$(a, stack)$	(a)	POP$(stack)$	(c,b,a)
enter(b)	(a)	*Comp*$(f) := C_2$	(c,b,a)
Root$(b) := b$	(a)	*exit*(f)	(c,b,a)
PUSH$(b, stack)$	(b,a)	*enter*(h)	(c,b,a)
enter(c)	(b,a)	*Root*$(h) := h$	(c,b,a)
Root$(c) := c$	(b,a)	PUSH$(h, stack)$	(h,c,b,a)
PUSH$(c, stack)$	(c,b,a)	*enter*(i)	(h,c,b,a)
enter(d)	(c,b,a)	*Root*$(i) := i$	(h,c,b,a)
Root$(d) := d$	(c,b,a)	PUSH$(i, stack)$	(i,h,c,b,a)
PUSH$(d, stack)$	(d,c,b,a)	*Root*$(i) := h$	(i,h,c,b,a)
enter(e)	(d,c,b,a)	*enter*(j)	(i,h,c,b,a)
Root$(e) := e$	(d,c,b,a)	*Root*$(j) := j$	(i,h,c,b,a)
PUSH$(e, stack)$	(e,d,c,b,a)	PUSH$(j, stack)$	(j,i,h,c,b,a)
Root$(e) := d$	(e,d,c,b,a)	create C_3	(j,i,h,c,b,a)
exit(e)	(e,d,c,b,a)	POP$(stack)$	(i,h,c,b,a)
create C_1	(e,d,c,b,a)	*Comp*$(j) := C_3$	(i,h,c,b,a)
POP$(stack)$	(d,c,b,a)	*exit*(j)	(i,h,c,b,a)
Comp$(e) := C_1$	(d,c,b,a)	*Root*$(i) := b$	(i,h,c,b,a)
POP$(stack)$	(c,b,a)	*exit*(i)	(i,h,c,b,a)
Comp$(d) := C_1$	(c,b,a)	*Root*$(h) := b$	(i,h,c,b,a)
exit(d)	(c,b,a)	*exit*(h)	(i,h,c,b,a)
Root$(c) := b$	(c,b,a)	create C_4	(i,h,c,b,a)
exit(c)	(c,b,a)	POP$(stack)$	(h,c,b,a)
Root$(b) := a$	(c,b,a)	*Comp*$(i) := C_4$	(h,c,b,a)
exit(b)	(c,b,a)	POP$(stack)$	(c,b,a)
enter(f)	(c,b,a)	*Comp*$(h) := C_4$	(c,b,a)
Root$(f) := f$	(c,b,a)	POP$(stack)$	(b,a)
PUSH$(f, stack)$	(f,c,b,a)	*Comp*$(c) := C_4$	(b,a)
enter(g)	(f,c,b,a)	POP$(stack)$	(a)
Root$(g) := g$	(f,c,b,a)	*Comp*$(b) := C_4$	(a)
PUSH$(g, stack)$	(g,f,c,b,a)	POP$(stack)$	$()$
Root$(g) := f$	(g,f,c,b,a)	*Comp*$(a) := C_4$	$()$
exit(g)	(g,f,c,b,a)	*exit*(a)	$()$
create C_2	(g,f,c,b,a)		

FIGURE 3.3: The trace of Tarjan's algorithm applied to the graph of Figure 3.2.

are related to the depth-first search that Tarjan's algorithm uses to traverse the input graph. Similar definitions can be given for any algorithm that is based on a depth-first search.

Definition. Let $G = (V, E)$ be a graph. A *depth-first spanning forest of G induced by an execution of Tarjan's algorithm* is a spanning forest $F = (V, E')$ of G such that E' contains an edge (v, w) of E iff procedure VISIT entered w via edge (v, w) at line 6 of Tarjan's algorithm.

By this definition, the execution of VISIT(v) contains the execution of VISIT(w) iff a path $v \xrightarrow{*} w$ exists in the depth-first spanning forest. A graph usually has many possible depth-first spanning forests, since the order of scanning the vertices at line 20 and the order of scanning the edges at line 5 of Tarjan's algorithm is not fixed.

Definition. A *depth-first order of graph $G = (V, E)$ induced by an execution of Tarjan's algorithm*, denoted $\leq_\tau$, is a total order on the vertex set V such that $v <_\tau w$ iff procedure VISIT entered v before w in the execution of Tarjan's algorithm.

Similarly to the depth-first spanning forests, a graph usually has many possible depth-first orders. Note that if a depth-first spanning forest F contains a path $v \xrightarrow{*} w$, then $v \leq_\tau w$. The converse is not necessarily true, since vertex v may be both entered and exited before w is entered.

Definition. Given a graph $G = (V, E)$ together with a depth-first spanning forest $F = (V, E')$ and a depth-first order $\leq_\tau$ of G induced by an execution of Tarjan's algorithm, the edge set E is partitioned into four groups as follows. Let (v, w) be in E and let T_w be the subtree of F rooted at w.

1. If (v, w) is in E', then (v, w) is a *tree edge*.
2. If $v <_\tau w$ and (v, w) is not in E', then (v, w) is a *forward edge*.
3. If $w \leq_\tau v$ and v is in T_w, then (v, w) is a *back edge*.
4. If $w <_\tau v$ and v is not in T_w, then (v, w) is a *cross edge*.

Note that for each forward edge (v, w), the graph contains a path from v to w consisting solely of tree edges and vertex w is in the subtree T_v of F rooted at v. Note also that a back edge indicates a cycle in the graph.

Definition. An edge (v, w) such that v and w are in the same component C is called an *intracomponent edge*. An edge (v, w) such that v and w are in two different components C_1 and C_2 is called an *intercomponent edge*.

We divide intracomponent edges further into intracomponent tree edges, intracomponent forward edges, intracomponent back edges, and intracomponent cross edges. Similarly, we divide intercomponent edges into intercomponent tree edges, intercomponent forward edges, and intercomponent cross edges. Note that no intercomponent back edges exist, since a back edge is always inside a component.

Definition. Given a depth-first order $\leq_\tau$ of graph G and a strong component C of G, the *root* of C is the smallest vertex of C in $\leq_\tau$.

The root of a strong component may be different in different executions of Tarjan's algorithm. The following lemma explains the name "root."

Lemma 3.1. Let C be a strong component of a graph G, let F be a depth-first spanning forest of G induced by an execution of Tarjan's algorithm, and let r be the root of C in the same execution. Then each vertex v of component C is in the subtree T_r of F rooted at r.

Proof. Suppose that C contains a vertex v that is not in T_r. Obviously $v \neq r$. Since r is the root, VISIT has not entered v when it enters r, and since v is not in T_r, v is not entered during the execution of VISIT(r). But neither can any vertex adjacent to v be in T_r, since then v would be entered during the execution VISIT(u) for some vertex u adjacent to v. The same argument can repeatedly be applied to all predecessors of v. Hence no predecessor of v is in T_r. But this is a contradiction, since r is in T_r and r is a predecessor of v. □

Definition. Let $\overline{G}$ be the condensation graph induced by the strong components of G. A *leaf of* $\overline{G}$ is a strong component with no outgoing edges. The *level of a strong component* C is the length of the longest path from C to a leaf in $\overline{G}$.

We prove the correctness of Tarjan's algorithm (and later the correctness of other algorithms) by induction on the level of the strong component.

Lemma 3.2. For every vertex v, $Root(v) \leq_\tau v$.

Proof. At line 3 of VISIT, v is assigned to $Root(v)$. At line 7, the minimum of $Root(v)$ and $Root(w)$ is assigned to $Root(v)$. Since $Root(v)$ is not modified elsewhere, $Root(v) \leq_\tau v$. □

Definition. The *final candidate root of vertex* x, denoted $Fcr(x)$, is the vertex y such that $Root(x) = y$ at line 9 of Tarjan's algorithm when procedure VISIT has processed all edges leaving vertex x.

Theorem 3.3. Tarjan's algorithm, presented in Figure 3.1, correctly detects the strong components of the input graph.

Proof. A component C with root r is correctly detected iff $Comp(x) = C$ for each vertex x of C when VISIT exits the root r, and the contents of the auxiliary stack is the same as it was before VISIT entered the root r. The proof consists of four parts **(a)**–**(d)**. Given a component C with root r, part **(a)** shows that for each vertex x in C, $Fcr(x)$ is in C. Part **(b)** shows that $Fcr(r) = r$. Part **(c)** shows that for each nonroot vertex x of C, $Fcr(x) <_\tau x$. Finally, part **(d)** shows that parts **(b)** and **(c)** imply the correct detection of component C. In each part of the proof, we use induction on the level of the component C.

(a) For each vertex v of a component C at level zero, $Fcr(v)$ must be in C, since no edge leaves C. Let the level of C be $l > 0$. Let v be any vertex in component C. When VISIT processes an edge (v, w) and vertex w is in another component C', C' must be at a level below l. If VISIT has not already entered w, it enters w and (by the induction hypothesis) correctly detects C'. If w is already visited, then (again by the induction hypothesis) VISIT has correctly detected C'. In both cases, $Comp(w) \neq Nil$ at line 7 in VISIT(v), and w is not used to update $Root(v)$. Hence $Fcr(v)$ is in C.

(b) Suppose, on the contrary, that $Fcr(r) <_\tau r$; by Lemma 3.2, $Fcr(r)$ cannot be greater than r. By **(a)**, this implies that C contains a vertex $x <_\tau r$. Thus, x was entered before r, and r cannot be the root of C, a contradiction.

(c) Suppose, on the contrary, that $Fcr(x) = x$ for some vertex $x \neq r$ in C. Let x be the first vertex of C satisfying the test $Root(x) = x$ at line 9 of VISIT during the execution of Tarjan's algorithm. Since both x and r are in C, one or more non-null paths $x \stackrel{+}{\rightarrow} r$ are

inside component C. Let T_x be the depth-first spanning tree rooted at x. Since $r <_\tau x$, no path $x \xrightarrow{+} r$ completely lies in T_x. Let (v, w) be the first edge considered by VISIT such that (v, w) is on a path $x \xrightarrow{+} r$ and v is in T_x, but w is not in T_x. Since w is not in T_x, VISIT has entered it before x and inserted it onto the stack. Since x is the first vertex of C satisfying the test $Root(x) = x$, w is still on the stack, and $Comp(w) = Nil$. Thus, VISIT uses $Root(w)$ in updating $Root(v)$ and (by Lemma 3.2) $Root(v) \leq_\tau w$ after (v, w) is processed. Similarly, in each vertex u on the path $x \xrightarrow{*} v$, VISIT sets $Root(u) \leq_\tau w$. In particular, VISIT sets $Root(x) \leq_\tau w$. Since w was entered before x, $w <_\tau x$. Thus, $Fcr(x) \leq_\tau Root(x) \leq_\tau w <_\tau x$, a contradiction.

(d) Let C be a component with root r. By Lemma 3.1, all vertices of C are visited during the execution of VISIT(r) and are inserted onto the stack in the order they are entered. If C is at level $l > 0$, each component other than C visited during the execution of VISIT(r) is at a level below l. By the induction hypothesis, these components are correctly detected. If C is at level zero, no other component is visited during the execution of VISIT(r). Thus, the processing of other components does not remove vertices of C from the stack. Each vertex of C remains on the stack until the condition $Root(v) = v$ is satisfied for some vertex v of C at line 9 of VISIT(v). By **(b)** and **(c)**, the condition is satisfied only for the root r. Since all other components processed during VISIT(r) are correctly detected, the vertices of C are on top of the stack. Since the vertices are inserted onto the stack in the order they are entered, root r is the bottom-most vertex of C on the stack. Thus, at lines 11–15 in VISIT(r), each vertex w of C is removed from the stack and $Comp(w)$ is set to C. When VISIT exits r, the contents of the stack is the same as it was when r was entered. Thus, C is correctly detected. □

Theorem 3.4. Tarjan's algorithm runs in $\Theta(n + e)$ time, where n is the number of vertices and e the number of edges in the input graph.

Proof. The tests at lines 6 and 21 guarantee that VISIT enters each vertex at most once. The for-loop at line 20 in the main program considers each vertex once. Thus, VISIT enters each vertex exactly once. The for-loop at lines 5–8 scans each edge leaving the vertex v once. Since VISIT is applied once for each vertex v, all e edges are considered exactly once. Each vertex is stored onto the stack at line 4. The repeat-loop at lines 11–15 removes each vertex of a component from the stack. Since each vertex belongs to a single component, the repeat-loop runs altogether n times. These are the major costs of the algorithm; other operations can be done in constant time. Adding these costs together yields the $\Theta(n + e)$ bound. □

Finally, we present a theorem that forms a basis for many algorithms that are based on Tarjan's algorithm. For instance, the transitive closure algorithms that we study below are based on this theorem.

Theorem 3.5. The order in which Tarjan's algorithm detects the strong components of the input graph G is a reverse topological order of the condensation graph $\overline{G}$ induced by the strong components of G.

Proof. Suppose, on the contrary, that the components are not detected in a reverse topological order, i.e., the graph contains two distinct components X and Y such that X is detected before

Y, and at least one path p goes from X to Y in the condensation graph $\overline{G}$. Let Z be the last component in path p before Y. If Y is not entered before or during the detection of X, then Z cannot be entered either; otherwise Y would be entered via some edge leading from Z to Y. By a similar argument, we can show that no component preceding Y in path p can be entered before or during the detection of X. But at least X itself is entered before X is detected, a contradiction. Thus Y is entered either before or during the detection of X. But if Tarjan's algorithm entered Y before X is detected, Y would be detected first, since no path leads from Y back to component X, a contradiction. □

3.1.2 Improvements

Although Tarjan's algorithm is asymptotically optimal, it does some unnecessary work. If the input graph is acyclic, each strong component consists of a single vertex. Thus, the second traversal that marks the elements of a component is useless and the auxiliary stack is unnecessary. Also cyclic graphs may contain such trivial components. We study next how the second traversal can be eliminated when it is not needed.

Tarjan's algorithm has the following property: a new strong component is detected when processing its root vertex. During the second traversal that marks the vertices of the component, we would have access to the root vertex even if it were not stored onto the stack. Our first improved version of Tarjan's algorithm, called NEWSCC1 and presented in Figure 3.4 is

```
(1)   procedure VISIT1(v);
(2)   begin
(3)       Root(v) := v; Comp(v) := Nil;
(4)       for each vertex w such that (v, w) ∈ E do begin
(5)           if w is not already visited then VISIT1(w);
(6)           if Comp(w) = Nil then Root(v) := MIN(Root(v), Root(w))
(7)       end;
(8)       if Root(v) = v then begin
(9)           create a new component C;
(10)          Comp(v) := C;
(11)          insert v into component C;
(12)          while TOP(stack) > v do begin
(13)              w := POP(stack);
(14)              Comp(w) := C;
(15)              insert w into component C
(16)          end
(17)      end else PUSH(v, stack);
(18)  end;
(19)  begin /* Main program */
(20)      Initialize stack to contain a value < any vertex in V;
(21)      for each vertex v ∈ V do
(22)          if v is not already visited then VISIT1(v)
(23)  end.
```

FIGURE 3.4: Algorithm NEWSCC1 stores only nonroot vertices on the stack.

based on this observation. The recursive procedure VISIT1 in algorithm NEWSCC1 stores a vertex v onto the stack only after it has processed all edges leaving v and knows that v is not a component root (at line 17). Since the root vertex is not on the stack when the component is detected, the stack is processed slightly differently in NEWSCC1 compared to Tarjan's algorithm. Each nonroot vertex of a component is greater than the root in the depth-first order. Therefore (at lines 12–16), we remove vertices from the stack as long as the topmost vertex is greater than the root vertex (in the depth-first order) and assign C to $Comp(w)$ for each nonroot vertex w of the component. To prevent stack underflow, the stack is initialized to contain a sentinel value smaller than any vertex of the input graph. The *Comp* variable of the root vertex is set at line 9.

Example 3.2. Consider again the graph in Figure 3.2(a). In Figure 3.5, we present the trace of one possible application of NEWSCC1 to the graph. We assume that the execution starts at vertex a, and that the vertices adjacent from a vertex v are processed in the order in which they appear in the adjacency list of v. The adjacency lists are presented in Figure 3.2(b). As we see, NEWSCC1 stores only six vertices onto *stack*, whereas Tarjan's algorithm stores all ten vertices on *stack*. □

Theorem 3.6. Algorithm NEWSCC1, presented in Figure 3.4, correctly detects the strong components of the input graph.

Proof. The proof is similar to the correctness proof of Tarjan's algorithm above. We can show that given a component C with root r: **(a)** for each vertex x in C, $Fcr(x)$ is in C, **(b)** $Fcr(r) = r$, and **(c)** for each nonroot vertex x of C, $Fcr(x) <_\tau x$. The only difference is in showing how **(b)** and **(c)** imply **(d)**, the correct detection of component C.

(d) Let C be a component with root r. Similarly to Lemma 3.1 we can show that all vertices of C are visited during the execution of VISIT1(r). By **(c)**, the condition $Root(v) = v$ at line 8 fails for each nonroot vertex v. Thus, each nonroot vertex is inserted onto the stack at line 17. If C is at level $l > 0$, each component other than C visited during the execution of VISIT1(r) is at a level below l. By the induction hypothesis, these components are correctly detected. If C is at level zero, no other component is visited during the execution of VISIT1(r). Thus, the processing of other components does not remove vertices of C from the stack. Each vertex of C inserted onto the stack remains there until the condition $Root(v) = v$ is satisfied for some vertex v of C at line 8 in VISIT1(v). By **(b)** and **(c)**, the condition is satisfied only for the root r. Since all other components processed during VISIT(r) are correctly detected, the vertices of C are on top of the stack. Since $r <_\tau w$ for each nonroot vertex w, each nonroot vertex w of C is removed from the stack at lines 12–16 in VISIT1(r) and $Comp(w)$ is set to C. $Comp(r)$ is set to C at line 10. When VISIT1 exits r, the contents of the stack is the same as it was when r was entered. Thus, C is correctly detected. □

Theorem 3.7. Algorithm NEWSCC1 runs in $\Theta(n + e)$ time, where n and e are the number of vertices and the number of edges in the input graph, respectively.

Proof. Similar to the proof of Theorem 3.4. □

Operation	Stack	Operation	Stack
enter(a)	()	*Comp*(f) := C_2	(g, b, c)
Root(a) := a	()	POP(*stack*)	(b, c)
enter(b)	()	*Comp*(g) := C_2	(b, c)
Root(b) := b	()	*exit*(f)	(b, c)
enter(c)	()	*enter*(h)	(b, c)
Root(c) := c	()	*Root*(h) := h	(b, c)
enter(d)	()	*enter*(i)	(b, c)
Root(d) := d	()	*Root*(i) := i	(b, c)
enter(e)	()	*Root*(i) := h	(b, c)
Root(e) := e	()	*enter*(j)	(b, c)
Root(e) := d	()	*Root*(j) := j	(b, c)
PUSH(e, *stack*)	(e)	create C_3	(b, c)
exit(e)	(e)	*Comp*(j) := C_3	(b, c)
create C_1	(e)	*exit*(j)	(b, c)
Comp(d) := C_1	(e)	*Root*(i) := b	(b, c)
POP(*stack*)	()	PUSH(i, *stack*)	(b, c)
Comp(e) := C_1	()	*exit*(i)	(i, b, c)
exit(d)	()	*Root*(h) := b	(i, b, c)
Root(c) := b	()	PUSH(h, *stack*)	(h, i, b, c)
PUSH(c, *stack*)	(c)	*exit*(h)	(h, i, b, c)
exit(c)	(c)	create C_4	(h, i, b, c)
Root(b) := a	(c)	*Comp*(a) := C_4	(h, i, b, c)
PUSH(b, *stack*)	(b, c)	POP(*stack*)	(i, b, c)
exit(b)	(b, c)	*Comp*(h) := C_4	(i, b, c)
enter(f)	(b, c)	POP(*stack*)	(b, c)
Root(f) := f	(b, c)	*Comp*(i) := C_4	(b, c)
enter(g)	(b, c)	POP(*stack*)	(c)
Root(g) := g	(b, c)	*Comp*(b) := C_4	(c)
Root(g) := f	(b, c)	POP(*stack*)	()
PUSH(g, *stack*)	(g, b, c)	*Comp*(c) := C_4	()
exit(g)	(g, b, c)	*exit*(a)	()
create C_2	(g, b, c)		

FIGURE 3.5: The trace of NEWSCC1 applied to the graph of Figure 3.2.

We examine now the possibility to further reduce the second traversal in Tarjan's algorithm. Obviously, if we have to output the components, we need an access to each vertex of the component. But if we only want to detect the component roots, for instance, to compute the number of strong components, we can do better than in NEWSCC1.

Examine line 7 in Figure 3.1. This is the only place where we test whether the child vertex w belongs to the same component as vertex v. Note that w belongs to the same component as v iff *Root*(w) belongs to the same component as v. Tests *Comp*(w) = *Nil* and *Comp*(*Root*(w)) = *Nil* always yield the same result. If we replace the test *Comp*(w) = *Nil* by the test *Comp*(*Root*(w)) = *Nil*, we only have to access and modify the *Comp* values of the final candidate roots. When a component C is detected, it suffices to set *Comp*(w) = C for each final candidate root w in component C.

```
(1)   procedure VISIT2(v);
(2)   begin
(3)       Root(v) := v; Comp(v) := Nil;
(4)       for each vertex w such that (v, w) ∈ E do begin
(5)           if w is not already visited then VISIT2(w);
(6)           if Comp(Root(w)) = Nil then Root(v) := MIN(Root(v), Root(w))
(7)       end;
(8)       if Root(v) = v then begin
(9)           create a new component C;
(10)          if TOP(stack) ≥ v then
(11)              repeat
(12)                  w := POP(stack);
(13)                  Comp(w) := C
(14)              until TOP(stack) < v;
(15)          else Comp(v) := C;
(16)      end else if Root(v) is not on stack then PUSH(Root(v), stack);
(17)  end;
(18)  begin /* Main program */
(19)      Initialize stack to contain a value < any vertex in V;
(20)      for each vertex v ∈ V do
(21)          if v is not already visited then VISIT2(v)
(22)  end.
```

FIGURE 3.6: Algorithm NEWSCC2 stores only final candidate roots of nontrivial components on the stack.

Our second improved version of Tarjan's algorithm, called NEWSCC2 and presented in Figure 3.6, is based on this idea. Procedure VISIT2 stores each final candidate root of a nontrivial component onto the stack at line 16. When a nontrivial component is detected, its final candidate root vertices (at least one exists) are on top of the stack. VISIT2 removes vertices from the stack until the topmost vertex is smaller than the actual root vertex (in the depth-first order) and assigns C to their *Comp* variables at lines 10–14. If the component is trivial, the algorithm only sets $Comp(v) = C$ (at line 15). To prevent stack underflow, the stack is initialized to contain a sentinel value smaller than any vertex of the input graph.

Example 3.3. Consider again the graph in Figure 3.2(a). In Figure 3.7, we present the trace of one possible application of NEWSCC2 on the graph. We assume that the execution starts at vertex a, and that the vertices adjacent from a vertex v are processed in the order in which they appear in the adjacency list of v. The adjacency lists are presented in Figure 3.2(b). As we see, NEWSCC2 stores only four vertices onto *stack*, whereas Tarjan's algorithm stores all ten vertices and NEWSCC1 six vertices onto *stack*. The four vertices stored onto *stack* are the roots of the nontrivial vertices and vertex b, which is the final candidate root of vertices c, h, and i. Note that NEWSCC2 changes the *Comp* variables of only five vertices. These are vertex b and the roots of the components. □

Operation	Stack	Operation	Stack
enter(a)	()	*Root*(g) := f	(a, b)
Root(a) := a	()	PUSH(f, *stack*)	(f, a, b)
enter(b)	()	*exit*(g)	(f, a, b)
Root(b) := b	()	create C_2	(f, a, b)
enter(c)	()	POP(*stack*)	(a, b)
Root(c) := c	()	*Comp*(f) := C_2	(a, b)
enter(d)	()	*exit*(f)	(a, b)
Root(d) := d	()	*enter*(h)	(a, b)
enter(e)	()	*Root*(h) := h	(a, b)
Root(e) := e	()	*enter*(i)	(a, b)
Root(e) := d	()	*Root*(i) := i	(a, b)
PUSH(d, *stack*)	(d)	*Root*(i) := h	(a, b)
exit(e)	(d)	*enter*(j)	(a, b)
create C_1	(d)	*Root*(j) := j	(a, b)
POP(*stack*)	()	create C_3	(a, b)
Comp(d) := C_1	()	*Comp*(j) := C_3	(a, b)
exit(d)	()	*exit*(j)	(a, b)
Root(c) := b	()	*Root*(i) := b	(a, b)
PUSH(b, *stack*)	(b)	*exit*(i)	(a, b)
exit(c)	(b)	*Root*(h) := b	(a, b)
Root(b) := a	(b)	*exit*(h)	(a, b)
PUSH(a, *stack*)	(a, b)	create C_4	(a, b)
exit(b)	(a, b)	POP(*stack*)	(b)
enter(f)	(a, b)	*Comp*(a) := C_4	(b)
Root(f) := f	(a, b)	POP(*stack*)	()
enter(g)	(a, b)	*Comp*(b) := C_4	()
Root(g) := g	(a, b)	*exit*(a)	()

FIGURE 3.7: The trace of NEWSCC2 applied to the graph of Figure 3.2.

Theorem 3.8. Algorithm NEWSCC2, presented in Figure 3.6, correctly detects (the roots of) the strong components of the input graph.

Proof. Here the correct detection of a component C with root r means that when VISIT2 exits r, the contents of the stack is the same as it was when r was entered, and $Comp(w) = C$ for each final candidate root of C. The proof resembles the correctness proofs of Tarjan's algorithm and NEWSCC1 above. We can show, as in the proof of Theorem 3.3, that given a component C with root r: **(a)** for each vertex x in C, $Fcr(x)$ is in C, **(b)** $Fcr(r) = r$, and **(c)** $Fcr(x) <_\tau x$ for each nonroot vertex x of C. The only difference is in showing how **(b)** and **(c)** imply **(d)**, the correct detection of component C.

(d) Let C be a component with root r. Similarly to Lemma 3.1 we can show that all vertices of C are visited during the execution of VISIT2(r). By **(c)**, the condition $Root(v) = v$ fails for each nonroot vertex v at line 8 in VISIT2(v). Hence the final candidate roots of all nonroot vertices are inserted once onto the stack at line 16. If C is at level $l > 0$, each component other than C visited during the execution of VISIT2(r) is at a level below l. By the induction hypothesis, these components are correctly detected. If C is at level zero,

no other component is visited during the execution of VISIT2(r). Thus, the detection of other components during VISIT2(r) does not remove vertices of C from the stack. The vertices of C that are inserted onto the stack remain there until the condition $Root(v) = v$ is satisfied for some vertex v of C at line 8 in VISIT2(v). By **(b)** and **(c)**, the condition is satisfied only for the root r. If C is a trivial component, i.e., $C = \{r\}$, r is not on the stack and all vertices on the stack must be smaller than r. Hence the test at line 10 fails. Assigning C to $Comp(r)$ at line 15 implies the correct detection of C. If C is nontrivial, at least one vertex of C is on the stack. Since all other components that are processed during VISIT2(r) are correctly detected, the vertices of C on the stack are the topmost vertices. The test at line 10 succeeds, and each vertex w of C on the stack is removed from the stack at lines 11–14, and $Comp(w)$ is set to C. When VISIT2 exits r, the contents of the stack is the same as it was when r was entered. Thus, C is correctly detected. □

Theorem 3.9. Algorithm NEWSCC2 runs in $\Theta(n + e)$ time, where n and e are the number of vertices and the number of edges in the input graph, respectively.

Proof. The proof is similar to the proof of Theorem 3.4. We can check if a vertex is on the stack in constant time if we keep this information in a Boolean vector or other appropriate data structure. □

We conjecture that the second traversal cannot be completely removed, at least without changing the first traversal. If we remove the second traversal, we can access only the component root r when we detect a new component. Thus, only the variable $Comp(r)$ can be set to C. After a nonroot vertex w has been processed, we do not necessarily have a fixed length access path from w to the root of the component containing w. Thus, testing if w belongs to an already detected component cannot be done in constant time, which slows the first traversal.

We analyze now how the number of vertices stored onto the stack differs in the three algorithms. Let P_T, P_1, and P_2 be the number of vertices stored onto the stack by Tarjan's algorithm and by algorithms NEWSCC1 and NEWSCC2, respectively, when applied to a graph G. Tarjan's algorithm stores all n vertices onto the stack. Thus, $P_T = n$. NEWSCC1 stores a vertex onto the stack unless it is a component root. Thus, $P_1 = n - s$, where s is the number of strong components in the input graph. Let $p(C)$ be the number of vertices stored onto the stack by NEWSCC2 when processing a component C. If C is trivial, no vertices are stored onto the stack. If C is nontrivial, at least one vertex is not a final candidate root vertex and thus not stored onto the stack. Thus, $0 \leq p(C) \leq |C| - 1$, where $|C|$ is the number of vertices in component C. $P_2 = \sum_{C \in \Pi} p(C)$, where Π is the set of all strong components in the input graph. Using the inequality for $p(C)$, we get $0 \leq P_2 \leq n - s = P_1 < P_T$. Thus, Tarjan's algorithm always stores more vertices onto the stack than NEWSCC1, and NEWSCC2 stores at most as many vertices onto the stack as NEWSCC1.

3.2 Adapting Tarjan's algorithm to transitive closure computation

Two main strategies exist for employing strong component detection in computing the transitive closure. The first is the strategy used in Purdom's algorithm [101], which we described in section 2.3.2. In this strategy, the strong components are detected and the transitive closure is computed for the condensation graph induced by the components. The transitive closure of the condensation graph is then converted to the transitive closure of the original graph. The weakness in this strategy is that it requires several passes over the graph. This makes the constant costs high.

The second strategy is based on a more direct adaptation of Tarjan's algorithm to transitive closure computation. The transitive closure is computed during a single pass over the input graph by interleaving the detection of the strong components with the computation of the successor sets. The problem with this strategy is that we do not have all the information about the structure of the graph when we are traversing it. This may lead to redundant operations in the successor set computation.

We begin our study using the second strategy. We introduce a simple transitive closure algorithm, originally presented in [114], analyze its behavior and point out its weaknesses. In the next subsections, we present new transitive closure algorithms that overcome the problems in the simple algorithm and in other transitive closure algorithms that are based on strong component detection.

The simple transitive closure algorithm, called SIMPLE_TC and presented in Figure 3.8, is a straightforward modification of Tarjan's algorithm. Note that we could equally well have used NEWSCC1 as a basis for the algorithm. For transitive closure computation, we have added lines 5, 9, and 17 to Tarjan's algorithm.

To compute the transitive closure, we define a variable $Succ(v)$ for each vertex v. It contains the (partially computed) successor set of vertex v. Initially (at line 5), $Succ(v)$ contains only the vertices adjacent from v. At line 9, the successors of a child vertex w computed so far are inserted into $Succ(v)$. When a strong component is detected, the successor set of the root vertex is correctly computed. Other vertices of the component may have incomplete successor sets. At line 17, the successor set of the component root is distributed to the other members of the component.

Example 3.4. In Figure 3.9(a), we present again graph G and in Figure 3.9(b) the successor sets of the vertices of G. As we see, the vertices of a strong component have the same successor set. Examine what happens when we apply SIMPLE_TC on graph G. Assume that the execution starts at vertex a and that the vertices adjacent from a vertex v are processed in the adjacency list order, presented in Figure 3.10(a). Each vertex except j gets a non-empty partial successor set. We present these sets in Figure 3.10(b). The total size of the partial successor sets is 62. We need 17 union operations for computing the successor sets, one per each edge of the graph. If the graph is traversed in some other order, the size of the partial successor sets may be slightly different. □

Theorem 3.10. Algorithm SIMPLE_TC, presented in Figure 3.8, correctly computes the transitive closure of the input graph G.

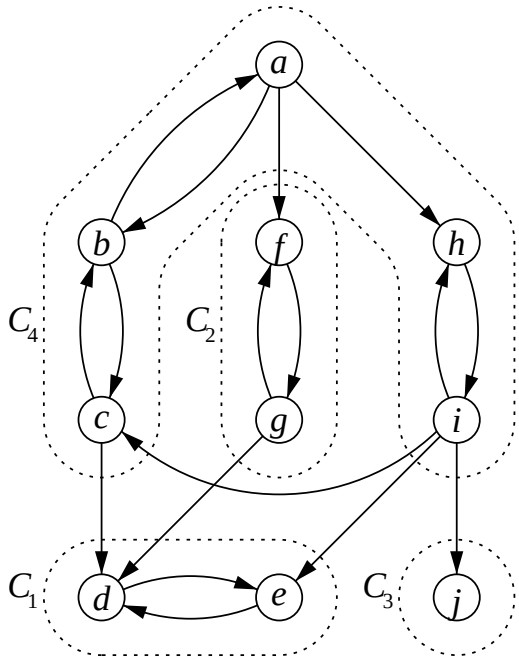
```
(1)   procedure SIMPLE_TC(v);
(2)   begin
(3)       Root(v) := v; Comp(v) := Nil;
(4)       PUSH(v, stack);
(5)       Succ(v) := {w | (v, w) ∈ E};
(6)       for each vertex w such that (v, w) ∈ E do begin
(7)           if w is not already visited then SIMPLE_TC(w);
(8)           if Comp(w) = Nil then Root(v) := MIN(Root(v), Root(w))
(9)           Succ(v) := Succ(v) ∪ Succ(w);
(10)      end;
(11)      if Root(v) = v then begin
(12)          create a new component C;
(13)          repeat
(14)              w := POP(stack);
(15)              Comp(w) := C;
(16)              insert w into component C;
(17)              Succ(w) := Succ(v);    /* Pointer assignment, not a copy */
(18)          until w = v
(19)      end
(20)  end;
(21)  begin /* Main program */
(22)      stack := ∅;
(23)      for each vertex v ∈ V do
(24)          if v is not already visited then SIMPLE_TC(v)
(25)  end.
```

FIGURE 3.8: Algorithm SIMPLE_TC: Tarjan's algorithm adapted to transitive closure computation.

Proof. The strong components are detected as in Tarjan's algorithm. We only have to show that the successor sets are correctly computed. Let C be a strong component and r its root. Since $Succ(r)$ is distributed to the other members of C at line 17, we have to show that when all edges leaving r are processed, $Succ(r)$ contains a vertex v iff G contains a non-null path $r \xrightarrow{+} v$. The only-if part is obvious, since SIMPLE_TC adds new successors to a successor set $Succ(v)$ only by employing edges leaving v. We show the if part by induction on the level of component C.

(i) Let $C = \{r\}$ be a trivial component at level zero. At line 5 in SIMPLE_TC(r), r is inserted into $Succ(r)$ if G has an edge (r, r). Thus, $Succ(r)$ is correctly computed. Let C be a nontrivial component at level zero and v a vertex of C. Since C is nontrivial, it has a vertex u such that G contains edge (u, v). Thus, each vertex v of C is inserted into the set $Succ(u)$ of some vertex u in C at line 5 in SIMPLE_TC(u). We show that the elements of $Succ(u)$ are added into $Succ(r)$. For each vertex $u \neq r$ in C, the depth-first spanning forest F induced on the execution of SIMPLE_TC contains a non-null path $p = (v_0, v_1), \ldots, (v_{k-1}, v_k)$, where $v_0 = r$ and $v_k = u$. Thus, the execution of SIMPLE_TC(v_{i-1}) contains the execution of SIMPLE_TC(v_i) for $1 \leq i \leq k$. When VISIT returns from vertex v_i to vertex v_{i-1}, $Succ(v_i)$ is added to $Succ(v_{i-1})$ in the union operation at line 9 of SIMPLE_TC(v_{i-1}). Thus, when

(a)



(b)

Vertex	Successors
a	$(a, b, c, d, e, f, g, h, i, j)$
b	$(a, b, c, d, e, f, g, h, i, j)$
c	$(a, b, c, d, e, f, g, h, i, j)$
d	(d, e)
e	(d, e)
f	(d, e, f, g)
g	(d, e, f, g)
h	$(a, b, c, d, e, f, g, h, i, j)$
i	$(a, b, c, d, e, f, g, h, i, j)$
j	$()$

FIGURE 3.9: **(a)** Graph G with four strong components. **(b)** The successor sets of the vertices of G.

(a)

Vertex	Adjacency list
a	(b, f, h)
b	(c, a)
c	(d, b)
d	(e)
e	(d)
f	(g)
g	(f, d)
h	(i)
i	(h, j, e, c)
j	$()$

(b)

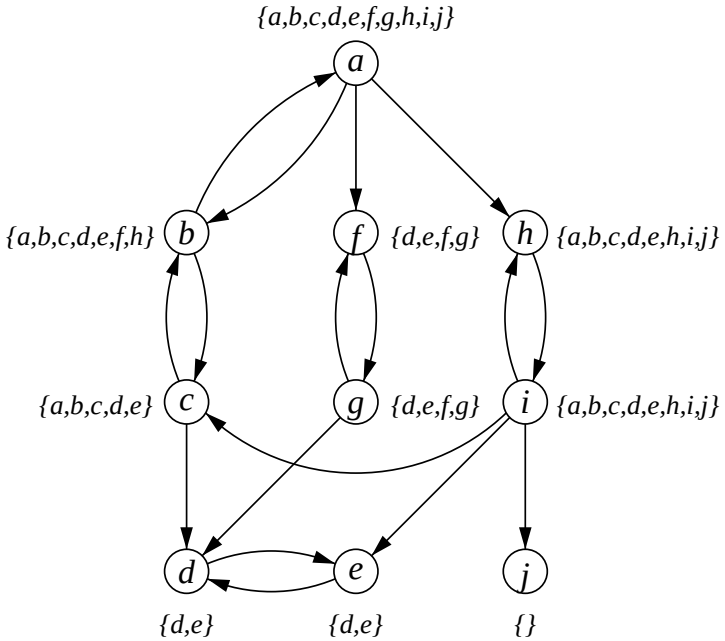


FIGURE 3.10: **(a)** The adjacency lists of graph G of Figure 3.9(a). **(b)** The partial successor sets of the vertices when SIMPLE_TC is applied to G.

edge (v_0, v_1) has been processed, $Succ(r)$ contains $Succ(u)$.

(ii) Let C be a component at level $l > 0$. The correct insertion of each vertex v of C into $Succ(r)$ can be shown as in the induction basis. We show now that if vertex v is not in C and a non-null path $r \xrightarrow{+} v$ exists, then v is inserted into $Succ(r)$. The path $r \xrightarrow{+} v$ can be divided into three parts: a possibly null path $r \xrightarrow{*} x$ inside C, an edge (x, y), and a possibly null path $y \xrightarrow{*} v$ outside C. If $y \xrightarrow{*} v$ is null, then $y = v$. Thus, v is adjacent from x and is inserted into $Succ(x)$ at line 5 in SIMPLE_TC(x). If $y \xrightarrow{*} v$ is non-null, then v is a successor of y. Component C' that contains y is at a level below l. By the induction hypothesis, the successors of component C' are correctly detected. Thus, $Succ(y)$ contains all successors of y before the union at line 9 in SIMPLE_TC(x), and v is inserted into $Succ(x)$ when the union at line 9 is executed. Without loss of generality, we can assume that path $r \xrightarrow{*} x$ is in the depth-first spanning forest F induced by the execution of SIMPLE_TC. If $x \neq r$, the inclusion of $Succ(x)$ into $Succ(r)$ can be shown as in the induction basis. Thus, v is inserted into $Succ(r)$. □

The most time-consuming operations in SIMPLE_TC are the successor set operations. The time required by these operations depends on the data structures that are used to implement the successor sets.

Common set data structures, which can be used to implement the successor sets, are bit matrices and bit vectors, ordered and unordered lists, and ordered binary search trees, which may or may not be balanced. Another possible data structure is an unordered list augmented with a bit vector; the contents of the list are duplicated in the bit vector, i.e., for each vertex v_i in the list, the corresponding position i of the bit vector holds one and other positions hold zeros.

In computing the worst-case execution times, we use the following worst-case bounds for the different set operations. In the bit matrix and bit vector representations, the initialization of an empty set takes $O(n)$ time. The membership test and the insertion operation take constant time, and the union operation takes $O(n)$ time. In the list representation (both ordered and unordered), the initialization of an empty set takes constant time. The membership test and the insertion and union operations take $O(n)$ time. (In the union operation of unordered lists, we need a bit vector to speed up the duplicate elimination.) In the ordered binary search tree representation, the initialization of an empty set takes constant time. The membership test and the insertion operations take $O(n)$ time if the tree is not balanced and $O(\log n)$ time if the tree is balanced. The union operation takes $O(n)$ time. In the list representation augmented with bit vectors, the initialization of an empty set takes $O(n)$ time. The membership test and the insertion operation take constant time, and the union operation takes $O(n)$ time.

We give two kinds of worst-case execution time bounds. A worst-case bound of the first kind describes the execution time independent of the successor set implementation, i.e., using the number of initialization, membership test, insertion, and union operations needed. A worst-case bound of the second kind describes the execution time when the successor set implementation is fixed. To describe the execution time independent of the successor set implementation, we use the following notation:

$Z(n)$ The maximum time of creating an empty set that can hold n elements.

$F(n)$ The maximum time of a membership test in a set of at most n elements.

$I(n)$ The maximum time of inserting an element into a set of at most n elements.

$U(n)$ The maximum time of unioning two sets of at most n elements.

We use n and e to denote the number of vertices and edges in the input graph, respectively. We omit the term $O(n+e)$ representing the scanning of the input graph and the strong component detection, although it sometimes, e.g., when $e = 0$, is the major cost.

Theorem 3.11 gives the implementation independent worst-case bound of SIMPLE_TC.

Theorem 3.11. Algorithm SIMPLE_TC runs in $O(nZ(n) + eI(n) + eU(n))$ time in the worst case.

Proof. A partial successor set is initialized at line 5 by creating an empty successor set, scanning the edges leaving v, and inserting the heads of the edges into the successor set. Since e edges and n successor sets exist, this takes $O(nZ(n) + eI(n))$ total time. Line 9 is executed e times, once for each edge. Hence the unions take $O(eU(n))$ time in the worst case. The pointer assignments at line 17 take $\Theta(n)$ time. Summing these terms yields the limit $O(nZ(n) + eI(n) + eU(n))$. □

Corollary 3.12. Algorithm SIMPLE_TC runs in $O(ne)$ time in the worst case when the successor sets are implemented as ordered lists or ordered binary trees.

This is the best worst-case bound that can be achieved with SIMPLE_TC when the usual successor set data structures discussed above are used.

Although SIMPLE_TC traverses the input graph exactly once, it is often inefficient. As Example 3.4 showed, the total size of the partial successor sets and the number of union operations needed to create these sets is often high. The next example reveals a severe problem in SIMPLE_TC.

Example 3.5. Figure 3.11 presents a cycle of n vertices. Assuming that SIMPLE_TC starts at vertex v_1, it enters the vertices recursively in order $v_1, v_2, \ldots, v_n$. For each vertex v_i, SIMPLE_TC initializes the successor set $Succ(v_i)$ to $\{v_{i+1}\}$ at line 5. $Succ(v_n)$ is initialized to $\{v_1\}$. SIMPLE_TC does not traverse (v_n, v_1), since v_1 is already visited. SIMPLE_TC adds the contents of $Succ(v_1)$ into $Succ(v_n)$. After this, $Succ(v_n) = \{v_1, v_2\}$. Then SIMPLE_TC exits the vertices in order $v_n, v_{n-1}, \ldots, v_1$. When SIMPLE_TC exits vertex v_{i+1}, it adds $Succ(v_{i+1})$ into $Succ(v_i)$. This way, each vertex receives a partial successor set that is larger than the successor set of its child vertex. The total memory requirement for these sets is $\Omega(n^2)$, and the execution takes $\Omega(n^2)$ time. This is inefficient, since the graph only has n edges and only contains one strong component. □

Example 3.5 is not an exception, but rather the rule. SIMPLE_TC does not avoid the redundant operations caused by strong components.

To design a better algorithm, we analyze the deficiencies in SIMPLE_TC that lead to large partial successor sets and unnecessary successor set operations. Some of these deficiencies are avoided in previous transitive closure algorithms that compute the successor sets during the detection of strong components, namely Eve's and Kurki-Suonio's algorithm [40], Ebert's algorithm [36], and the algorithm GDFTC by Ioannidis et al. [64]. However, considerable improvements are still possible.

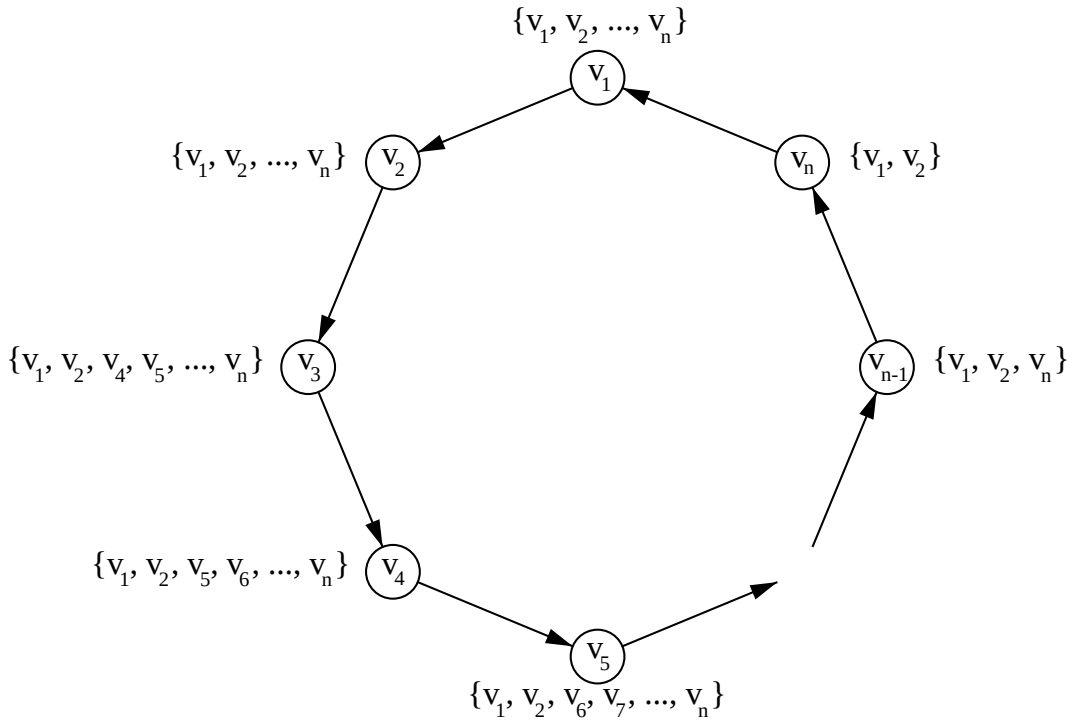


FIGURE 3.11: A cycle with partially computed successor sets.

A deficiency in SIMPLE_TC is that the algorithm inserts the members of a component C into $Succ(C)$ exactly as it inserts the other successors into $Succ(C)$. This produces large partial successor sets even when most successors of a component are inside that component. To get a better strategy, we need the conditions for inserting a component member into the successor set of that component. First, if the component has more than one member, all component members are successors of each other. Second, if the component has only one member v and an edge (v, v) exists, then v is its own successor. The better strategy is to record self-loop edges (v, v) and ignore all other intracomponent edges when initializing the partial successor sets at line 5 in SIMPLE_TC. When a component is detected, we insert the component members into the successor set of the component iff the component has more than one vertex or only a single vertex v and a self-loop (v, v) exists. This way we can insert the component members into the successor set of the component in $O(n+e)$ time. The processing of the example graph in Figure 3.11 would take $O(n)$ time instead of $O(n^2)$ time.

Separating the processing of component members from the processing of other successors makes another optimization possible. Instead of building the successor sets from vertices, we can build them from strong components and save much space. Thus, we actually compute the transitive closure of the condensation graph induced by the strong components as in Purdom's seminal algorithm [101]. Here we benefit from the property of Tarjan's algorithm that the components are detected in a reverse topological order, i.e., if a path from a component C to a different component C' exists, then C' is detected first. Unlike in Purdom's algorithm, no separate topological sorting of the components is needed. Storing strong components instead of vertices in the successor sets saves much memory space. The members of the strong components and the successor sets that contain strong components bear the same information as the successor sets that contain vertices. Answering a query like "is vertex u a successor of vertex v?" can be implemented by checking if $Comp(u)$ is contained in $Succ(Comp(v))$.

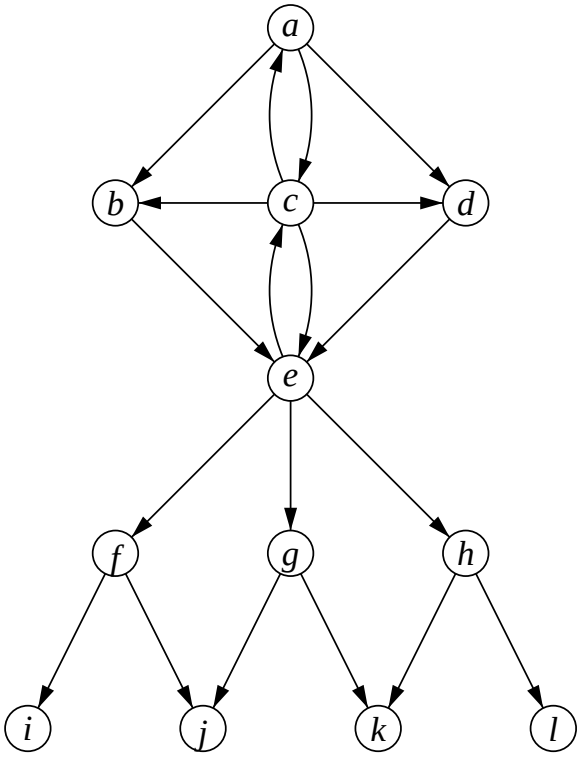


FIGURE 3.12: An example graph that leads to redundant operations in SIMPLE_TC.

All successors of a vertex v can be enumerated by listing the members of the components in $Succ(Comp(v))$. The computational complexity of answering these queries is here not greater than if the successor sets contained vertices instead of components. The previous algorithms [36, 40, 64] do not use either of these optimizations.

Another deficiency in SIMPLE_TC is that it uses all edges in computing the successor sets. It adds the successor set of the head of each edge to the successor set of the tail of that edge. In Example 3.5, this results in n union operations of partial successor sets of size $O(n)$. Another example is given below.

Example 3.6. Figure 3.12 shows a graph with a strong component $C_1 = \{a, b, c, d, e\}$ that is connected to an acyclic subgraph. If SIMPLE_TC traverses the graph starting at vertex a, it propagates the vertices of the acyclic subgraph from vertex e to root vertex a via all intracomponent edges, requiring 10 union operations. □

A more careful analysis shows that only a subset of all edges is needed to correctly propagate the successor sets. All forward edges can be ignored, since for each forward edge (v, w), the input graph contains a path $v \xrightarrow{*} w$ consisting solely of tree edges. The successors of w are added into the successors of v via this path. All back edges and intracomponent cross edges can be ignored, since they produce no new successors to the successor set of the root vertex of the component containing the edge. Ebert's algorithm [36] and the algorithm GDFTC by Ioannidis et al. [64] avoid forward edges, back edges, and intracomponent cross edges. These algorithms only use tree edges and intercomponent cross edges to propagate successor sets. Eve's and Kurki-Suonio's algorithm does not use any intracomponent edges to propagate the successor sets. Instead, when the component is detected, the algorithm unions the partial successor sets. The number of union operations needed to combine the partial successor sets is the same as in

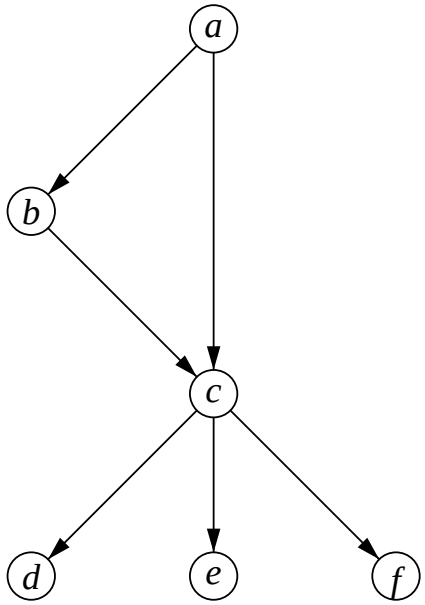


FIGURE 3.13: An example graph that causes SIMPLE_TC to add $Succ(c)$ twice to $Succ(a)$.

Ebert's algorithm and algorithm GDFTC. The weakness in Eve's and Kurki-Suonio's algorithm is that it propagates successor sets via intercomponent forward edges.

Further, when we are processing an edge (v, w), we should not add $Succ(w)$ again to $Succ(v)$ if $Succ(w)$ is already a subset of $Succ(v)$. For instance, if SIMPLE_TC is applied to the graph in Figure 3.13, the successor set $Succ(c)$ is added twice to $Succ(a)$, once via edge (a, c) and once via path $(a, b), (b, c)$. In general, this happens always when SIMPLE_TC is applied to any graph having pairs of vertices that are connected via multiple paths. Detecting all situations when a successor set $Succ(w)$ is a subset of the target set $Succ(v)$ requires a general set containment test and is probably too expensive, but we can detect many such situations if we use the following observation: a completely computed successor set $Succ(w)$ is a proper subset of another completely computed successor set $Succ(v)$ iff $Succ(v)$ contains w. Thus, if we insert a vertex w and its successor set $Succ(w)$ into another successor set $Succ(v)$ always at the same time, we can avoid inserting $Succ(w)$ into $Succ(v)$ again simply by checking if w is a member of $Succ(v)$. Obviously, all successor sets should be initialized to empty sets. This strategy is used in some previous transitive closure algorithms that compute the successor sets after detecting the components [45, 64, 101], but we can use it also with an algorithm that computes the successor sets during the detection of the strong components.

Yet another deficiency in SIMPLE_TC, which is also present in the previous algorithms [36, 40, 64], is that SIMPLE_TC uses $Succ(v)$ as the target of the union operation when it is processing the edges leaving v. A better strategy would be to add the successor set $Succ(w)$ of the head of an edge (v, w) directly into the successor set of the root of $Comp(v)$, but this is not possible in an algorithm like SIMPLE_TC. During the detection of a component C, we have no single location associated with C, where we could insert all its successors. Fortunately, we can do better than in SIMPLE_TC and in the previous algorithms by using only some partial successor sets as targets of unions. We can use the following heuristic. When we are processing an edge (v, w), we insert $Succ(w)$ into $Succ(Root(v))$, the partial successor set of the current candidate root vertex. This reduces the number and the total size of the partial successor sets

that are targets of insertions. This strategy is used in none of the previous algorithms.

3.3 New algorithm CR_TC

We describe now a new transitive closure algorithm, called CR_TC and presented in Figure 3.14, that uses the optimizations described above. CR_TC scans the input graph only once and builds the successor sets from strong components instead of vertices. Like SIMPLE_TC, CR_TC uses Tarjan's algorithm to detect the strong components. We could have based CR_TC on NEWSCC1 equally well, and even on NEWSCC2 if we only wanted to compute the transitive closure of the condensation graph.

When entering a vertex v, CR_TC initializes $Succ(v)$ to an empty set at line 5. During the detection of a strong component, CR_TC records the self-loop edges (v, v) (at line 7), but does

```
(1)   procedure CR_TC(v);
(2)   begin
(3)       Root(v) := v; Comp(v) := Nil;
(4)       PUSH(v, stack);
(5)       Succ(v) := {}; SelfLoop(v) := false;
(6)       for each vertex w such that (v, w) ∈ E do begin
(7)           if w = v then SelfLoop(v) := true
(8)           else begin
(9)               if w is not already visited then CR_TC(w);
(10)              if Comp(w) = Nil then Root(v) := MIN(Root(v), Root(w))
(11)              else if (v, w) is not a forward edge and Comp(w) ∉ Succ(Root(v)) then
(12)                  Succ(Root(v)) := Succ(Root(v)) ∪ {Comp(w)} ∪ Succ(Comp(w));
(13)          end
(14)      end;
(15)      if Root(v) = v then begin
(16)          create a new component C;
(17)          if TOP(stack) ≠ v or SelfLoop(v) then Succ(C) := Succ(v) ∪ {C}
(18)          else Succ(C) := Succ(v);
(19)          repeat
(20)              w := POP(stack);
(21)              Comp(w) := C;
(22)              insert w into component C;
(23)              if w ≠ v and Succ(w) ≠ ∅ then Succ(C) := Succ(C) ∪ Succ(w);
(24)          until w = v
(25)      end
(26)  end;
(27)  begin /* Main program */
(28)      stack := ∅;
(29)      for each vertex v ∈ V do
(30)          if v is not already visited then CR_TC(v)
(31)  end.
```

FIGURE 3.14: Algorithm CR_TC, the "candidate root" transitive closure algorithm.

not use other intracomponent edges for successor set generation. Only intercomponent tree and cross edges are used for this purpose. When processing an edge (v, w), CR_TC does not automatically add $Succ(w)$ into $Succ(v)$ as SIMPLE_TC does. Instead, CR_TC checks at line 11 that (v, w) is not a forward edge, i.e., it is an intercomponent tree or cross edge, and that $Comp(w)$, the component containing w, is not already in $Succ(Root(v))$, the partial successor set of the current candidate root vertex of v. If these tests are satisfied, CR_TC adds $Comp(w)$ and $Succ(Comp(w))$ into $Succ(Root(v))$ at line 12. The use of $Root(v)$ instead of v here gives the algorithm the name "candidate root transitive closure algorithm," CR_TC, for short. Vertex v can be seen as a mediator between the component $Comp(w)$ and the candidate root vertex $Root(v)$. Note that since $Root(v)$ is updated during the processing of the edges leaving v, the successor sets of different vertices adjacent from v may be inserted into the partial successor sets of different candidate root vertices. When C is detected, CR_TC inserts C into $Succ(C)$ (at line 17) if C contains more than one vertex or if CR_TC has detected a self-loop (r, r). Obviously, C contains more than one vertex iff the topmost vertex on the vertex stack is different from the root vertex r. To get the full successor set, CR_TC unions the nonempty partial successor sets of the component members at line 23.

The strategy used in CR_TC usually decreases the number and the total size of the partial successor sets compared to SIMPLE_TC. Decreasing the number of different partial successor sets increases the probability that component $Comp(w)$ and $Succ(Comp(w))$ are already in a partial successor set S where we try to add them, in which case we can omit inserting them again.

Example 3.7. Consider again the graph G, now presented in Figure 3.15. If we apply CR_TC to G starting at vertex a, and the adjacency lists are the same as in our previous examples (see Figure 3.10(a)), we get four non-empty partial successor sets and their total size is six. To create these sets, CR_TC needs seven union operations. Five of these unions are needed to propagate successor sets via the intercomponent edges that are drawn solid in Figure 3.15 and two are needed to combine the partial successor sets of nonroot vertices. Compare this to Example 3.4 where we applied SIMPLE_TC to G. There we had nine non-empty partial successor sets, the total size of the sets was 62, and SIMPLE_TC needed 17 union operations to compute them.

The number of non-empty partial successor sets that CR_TC creates, the total size of these sets, and the number of union operations needed depend on the order in which the graph is traversed. For instance, if we change the order of vertices in the adjacency lists of vertex b and c to (a, c) and (b, d), respectively, the number of non-empty partial successor sets and the total size of the sets both decrease by one. □

Theorem 3.13. Algorithm CR_TC, presented in Figure 3.14, correctly computes the transitive closure of the input graph G.

Proof. The strong components are detected as in Tarjan's algorithm. We have to show that the successor sets are correctly computed. We show that after the execution of CR_TC, the successor set $Succ(C)$ of a component C contains a component X iff G contains a non-null path $v \xrightarrow{+} w$ such that vertex v is in C and vertex w is in X.

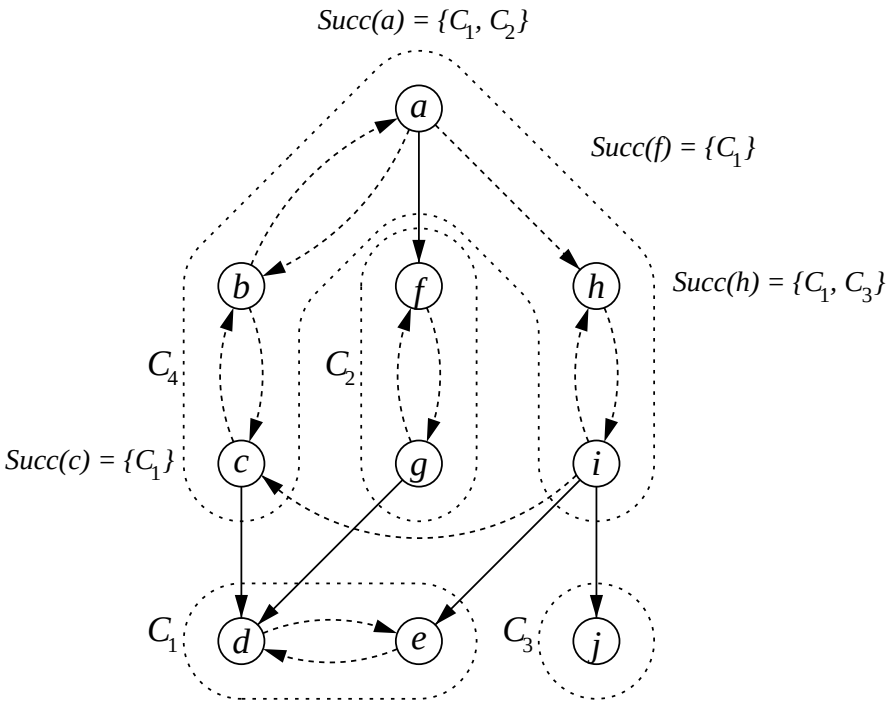


FIGURE 3.15: Graph G and the non-empty partial successor sets of the vertices when CR_TC is applied to G.

We can prove the only-if part by showing that the following invariant holds after any number of union operations in CR_TC: a successor set $Succ(C)$ or a partial successor set $Succ(u)$, $u \in C$, contains component X only if G contains a non-null path $v \xrightarrow{+} w$ such that v is in C and w is in X.

We prove the if part by induction on the level of component.

(i) Let $C = \{r\}$ be a trivial component at level zero. At line 5 CR_TC assigns **false** to $SelfLoop(v)$. If G contains an edge (r, r), CR_TC assigns **true** to $SelfLoop(v)$ at line 7. Since no edge leaves component C, $Succ(r)$ is empty when CR_TC has processed all edges leaving r. When component C is detected, the topmost element of the stack is r, and the test TOP($stack$) $\neq v$ at line 17 fails. Thus, CR_TC correctly assigns $\{C\}$ to $Succ(C)$ if the input graph contains a self-loop (r, r) and otherwise assigns $\emptyset$ to $Succ(C)$. Let C be a nontrivial component at level zero. Since no edge leaves component C, the successor sets of the members of C are empty when CR_TC has detected C. Since C is non-empty and since CR_TC entered the root r of C before the other vertices of C, the topmost element of the stack is not r. Thus, CR_TC correctly assigns $\{C\}$ to $Succ(C)$ at line 17.

(ii) Let C be a component at level $l > 0$. The correct insertion of C into $Succ(C)$ can be shown as in the induction basis. We show that CR_TC correctly inserts the successor components other than C into $Succ(C)$. Since $Succ(C)$ is constructed by unioning the non-empty partial successor sets of the vertices of C, we show that CR_TC inserts each successor component other than C into the partial successor set of some vertex of C. Each successor component that is not an immediate successor of C is a successor of some immediate successor of C. Each immediate successor component of C is at a level below l and, by the induction hypothesis, has its successor set correctly constructed. Hence each

successor component of C that is not an immediate successor is in the successor set of some immediate successor of C. It suffices to show that all immediate successors and their successor sets are correctly added into the partial successor set of some vertex v in C. Let X be an immediate successor of C and let (v, w) be an edge leading from C to X. When CR_TC is processing edge (v, w), it is either a tree edge, a cross edge, or a forward edge. If (v, w) is a tree edge or a cross edge, CR_TC adds X and $Succ(X)$ into $Succ(Root(v))$ at line 12. $Root(v)$ is in C, and we are done. If (v, w) is a forward edge, a path $v \xrightarrow{+} w$ consisting solely of tree edges exists. This path can be split into three parts: a path $v \xrightarrow{*} a$, an edge (a, b), and a path $b \xrightarrow{*} u$, where a is in C and b is in a component Y adjacent from C. If $Y = X$, CR_TC adds X and $Succ(X)$ into $Succ(Root(a))$ when processing edge (a, b). $Root(a)$ is in C, and we are done. If $Y \neq X$, then X is a successor component of Y and by the induction hypothesis has correctly been inserted into $Succ(Y)$. When CR_TC processes edge (a, b), it adds $Succ(Y)$ that contains X into $Succ(Root(a))$. $Root(a)$ is in C, and we are done. □

If we express the worst-case execution times of SIMPLE_TC and CR_TC using n and e as the only parameters, we get the same worst-case bound, namely $O(ne)$. However, as our examples have shown, CR_TC is more efficient in some situations. To make the difference between the execution times of SIMPLE_TC and CR_TC explicit, we have to use more fine-grained parameters than n and e. We introduce parameters that represent the sizes of some subsets of V, the set of vertices, and E, the set of edges. We express the worst-case execution time of CR_TC using these parameters and show that CR_TC has a better worst-case bound than SIMPLE_TC.

Theorem 3.14. Algorithm CR_TC runs in $O(nZ(n) + e_{oct}F(n) + (e_{oct1} + s_{cyc})I(n) + (n_1 + e_{oct1})U(n))$ time in the worst case. Here e_{oct} is the number of intercomponent tree and cross edges in the input graph, and e_{oct1} is the number of intercomponent tree and cross edges (v, w) such that $Comp(w)$ is not in $Succ(Root(v))$ at line 11 of CR_TC(v). s_{cyc} is the number of cyclic components, i.e., nontrivial components and trivial components with a self-loop edge. n_1 is the number of nonroot vertices with a non-empty partial successor set.

Proof. Initializing the partial successor sets at line 5 takes $O(nZ(n))$ total time. The self-loop detection at line 7 and the forward edge detection at line 11 take constant time per each edge, i.e., $O(e)$ total time in the worst case. The set membership lookup at line 11 is executed once per each intercomponent tree or cross edge. This takes $O(e_{oct}F(n))$ total time in the worst case. The insertion and union operations at line 12 are executed once per each intercomponent tree or cross edge (v, w) such that $Comp(w)$ is not already in $Succ(Root(v))$. This takes $O(e_{oct1}(I(n) + U(n)))$ total time in the worst case. The insertion operation at line 17 is executed once per each cyclic component. This takes $O(s_{cyc}I(n))$ total time in the worst case. The union at line 23 is executed once per each nonroot vertex having a nonempty partial successor set. This takes $O(n_1U(n))$ total time in the worst case. When we sum these terms, we get $O(nZ(n) + e_{oct}F(n) + (e_{oct1} + s_{cyc})I(n) + (n_1 + e_{oct1})U(n))$. □

Note that the number of cross edges in the graph depends on the order in which the algorithm happens to scan the vertices and the adjacency lists. Thus, the values of e_{oct} and e_{oct1} are not fixed for a given graph. We know, however, that $e_{oct1} \leq e_{oct} \leq e_o \leq e$, where e_o is the number of intercomponent edges. Further, we know that $s_{cyc} \leq s$, and $n_1 \leq n - s$, where s is the number of strong components in the input graph.

We present next the worst-case bounds that we get with different successor set representations.

Corollary 3.15. Algorithm CR_TC runs in $O(n^2+ne_{oct1})$ time in the worst case when bit vectors or lists augmented with bit vectors are used to implement the successor sets. Algorithm CR_TC runs in $O(n(n_1+s_{cyc}+e_{oct}))$ time in the worst case when lists are used to implement successor sets. Algorithm CR_TC runs in $O((s_{cyc}+e_{oct})\log n+n(n_1+e_{oct1}))$ time in the worst case when balanced ordered binary trees are used to implement the successor sets.

Assume that lists are used to represent the successor sets in CR_TC. Thus, the worst-case bound is $O(n(n_1 + s_{cyc} + e_{oct}))$. For each cyclic component, a graph contains at least one edge. For each nonroot vertex, the graph contains additionally one edge. Thus, $n_1 + s_{cyc} \leq (n - s) + s_{cyc} \leq e$, and $O(n(n_1 + s_{cyc} + e_{oct})) = O(n(e + e_{oct})) = O(ne)$. Thus, the worst-case bound of CR_TC is at most as great as the worst-case bound of SIMPLE_TC.

Assume that bit vectors or lists augmented with bit vectors are used to represent the successor sets in CR_TC. Thus, the worst-case bound is $O(n^2 + ne_{oct1})$. ne_{oct1} is at most as great as ne. An infinite set of graphs (and their traversals) exists, for which ne_{oct1} is negligible compared to ne (for instance, all graphs consisting of a single strong component). When $e > n$, ne is greater than n^2; when the graph is dense, n^2 is negligible compared to ne. Thus, when $e > n$, CR_TC has a better worst-case bound than SIMPLE_TC.

3.4 New algorithm STACK_TC

Compared to SIMPLE_TC, CR_TC reduces the number and the total size of the partial successor sets generated for nonroot vertices and the number of set operations. However, some inputs cause CR_TC to compute partial successor sets that are not needed after the computation. Combining the partial successor sets requires several union operations, which are the most expensive operations in transitive closure computation. We would like to eliminate all partial successor sets generated for nonroot vertices and thus avoid the expensive union operations required to combine them. The algorithm that we seek should generate exactly one successor set per each strong component. The problem is that no single location is associated with an incomplete strong component, where we could insert its successors. Thus, if we construct the successor set of a strong component during the detection of that component, we cannot avoid building partial successor sets and unioning them. The other strategy that is used in transitive closure algorithms presented in the literature [64, 91, 101, 105] is to construct the successor set of a component only after the component is detected or only after all components are detected. The problem with the previous algorithms is that they scan the input graph at least twice.

In this section, we present a new transitive closure algorithm that generates exactly one successor set per each strong component without scanning the input graph twice. The new algorithm delays the construction of the successor set $Succ(C)$ of component C until C is detected, thus avoiding the creation of partial successor sets for nonroot vertices. The algorithm avoids scanning the input graph twice, since during the detection of component C, the algorithm collects components adjacent from C that are later needed in constructing $Succ(C)$. An auxiliary stack, resembling the vertex stack of Tarjan's algorithm, is used for this purpose. When a component C is detected, the components that were stored onto the stack during the

detection of C are removed from the stack and the successor set of the component is computed by unioning the set of these components and their successor sets. To minimize the number of union operations needed, the algorithm sorts the components on the stack in a topological order before computing the successor set.

The new algorithm, called STACK_TC[1], is presented in Figure 3.16. Like CR_TC, STACK_TC stores strong components instead of vertices into the successors sets. The insertion of component C into its own successor set $Succ(C)$ is handled as in CR_TC. STACK_TC processes an edge (v, w) leaving a vertex v like CR_TC, except when (v, w) is an intercomponent tree or cross edge. Instead of adding $Comp(w)$ and $Succ(Comp(w))$ into $Succ(Root(v))$, STACK_TC stores $Comp(w)$ in the auxiliary stack *cstack*. This is done at line 13 in STACK_TC. When a component C is detected, STACK_TC creates a new successor set $Succ(C)$. If C is nontrivial or if $C = \{r\}$ and the input graph contains a self-loop (r, r), STACK_TC inserts C into $Succ(C)$. At lines 20–21, STACK_TC sorts the components that were stored onto *cstack* during the detection of C into a topological order and eliminates duplicates. To know the number of components that are stored onto *cstack* during the detection of component C, STACK_TC stores the current height of *cstack* into a local variable $SavedHeight(v)$ at line 5. After sorting the components into a topological order, STACK_TC removes the components from *cstack* at lines 22–25. For each component X removed from *cstack*, STACK_TC checks if X is already in $Succ(C)$. If X is not in $Succ(C)$, STACK_TC adds X and $Succ(X)$ into $Succ(C)$. If X is already in $Succ(C)$, every component Y in $Succ(X)$ also is in $Succ(C)$, and STACK_TC ignores X.

The details of the sorting are not presented in Figure 3.16. It can be done efficiently in the following way. Let r be the root of component C. Scan the components on *cstack* between the top and $SavedHeight(r)$ and use a bit vector to record the components that are present, removing duplicates. If the number of unique components remaining on *cstack* above $SavedHeight(r)$ is small, i.e., a number x such that $x \log x$ is smaller than n, sort them into the topological order using some common sorting algorithm. Otherwise, scan the bit vector to obtain the components directly in the topological order.

Note that *cstack* and *vstack* could be merged into a single stack.

Example 3.8. Consider again graph G presented in Figure 3.15(a). In Figure 3.17, we present the condensation graph $\overline{G}$ induced by the strong components of G. If we apply STACK_TC to G starting at vertex a and the adjacency lists are the same as in Figure 3.15(b), we need only three union operations to compute the transitive closure. The edges of the condensation graph that cause these unions are drawn solid whereas other edges are drawn dashed in Figure 3.17. In Example 3.7, where we applied CR_TC to graph G, we needed seven union operations and in Example 3.4, where we applied SIMPLE_TC to graph G, we needed 17 union operations. □

Theorem 3.16. Algorithm STACK_TC correctly computes the transitive closure of the input graph G.

Proof. The strong components are detected as in Tarjan's algorithm. We only have to show that the successor sets are correctly computed. We show that after the execution of STACK_TC, the successor set $Succ(C)$ of a component C contains a component X iff G contains a non-null path $v \xrightarrow{+} w$ such that vertex v is in C and vertex w is in X.

[1] In [92] we called this algorithm COMP_TC.

```
(1)   procedure STACK_TC(v);
(2)   begin
(3)       Root(v) := v; Comp(v) := Nil;
(4)       PUSH(v, vstack);
(5)       SavedHeight(v) := HEIGHT(cstack);
(6)       SelfLoop(v) := false;
(7)       for each vertex w such that (v, w) ∈ E do begin
(8)           if w = v then SelfLoop(v) := true
(9)           else begin
(10)              if w is not already visited then STACK_TC(w);
(11)              if Comp(w) = Nil then Root(v) := MIN(Root(v), Root(w))
(12)              else if (v, w) is not a forward edge then
(13)                  PUSH(Comp(w), cstack);
(14)          end
(15)      end;
(16)      if Root(v) = v then begin
(17)          create a new component C;
(18)          if TOP(vstack) ≠ v or SelfLoop(v) then Succ(C) := {C}
(19)          else Succ(C) := ∅;
(20)          sort the components in cstack between SavedHeight(v) and HEIGHT(cstack)
(21)          into a topological order and eliminate duplicates;
(22)          while HEIGHT(cstack) ≠ SavedHeight(v) do begin
(23)              X := POP(cstack);
(24)              if X ∉ Succ(C) then Succ(C) := Succ(C) ∪ {X} ∪ Succ(X);
(25)          end;
(26)          repeat
(27)              w := POP(vstack);
(28)              Comp(w) := C;
(29)              insert w into component C;
(30)          until w = v
(31)      end
(32)  end;
(33)  begin /* Main program */
(34)      vstack := ∅; cstack := ∅;
(35)      for each vertex v ∈ V do
(36)          if v is not already visited then STACK_TC(v)
(37)  end.
```

FIGURE 3.16: Algorithm STACK_TC.

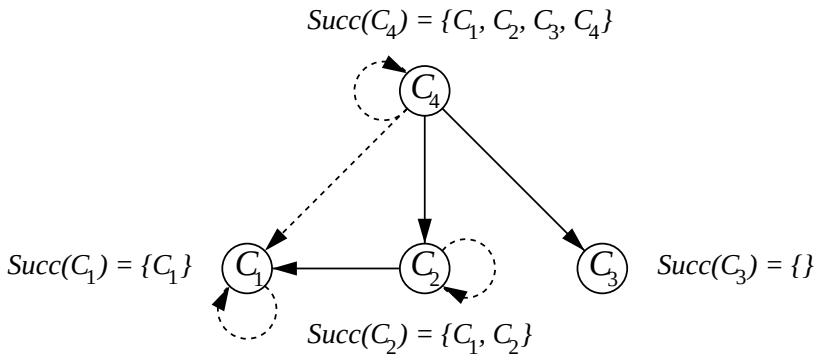


FIGURE 3.17: The condensation graph $\overline{G}$ of graph G presented in Figure 3.15(a) and the corresponding successor sets.

We prove the if part and the only-if part at the same time using induction on the level of the strong component C. We also show that all components that are stored on *cstack* during the detection of C are removed from *cstack* when $Succ(C)$ is constructed.

(i) Let C be a component at level zero. Since no edge leaves component C, no components are stored onto *cstack* during the detection of C, and no other component than C is inserted into $Succ(C)$. When all edges leaving the root r are processed, the top-most element of *vstack* is r iff C is trivial. The initial value of $SelfLoop(r)$ is **false**. STACK_TC assigns **true** to $SelfLoop(r)$ at line 8 iff the input graph has an edge (r, r). Thus, STACK_TC assigns $\{C\}$ to $Succ(C)$ at line 18 iff C is either non-trivial or $C = \{r\}$ and a self-loop edge (r, r) exists. Otherwise, STACK_TC assigns $\emptyset$ to $Succ(C)$. Hence $Succ(C)$ is correctly computed.

(ii) Let C be a component at level $l > 0$ with root vertex r. The correct insertion of C into $Succ(C)$ can be shown as in the induction basis. We only need to show that each component X different from C is inserted to $Succ(C)$ iff X is a successor of C.

Only if: When STACK_TC is processing an edge (v, w), where v is in C, it inserts $Comp(w)$ into *cstack* iff (v, w) is an intercomponent tree or cross edge. Thus, each component inserted into *cstack* during the processing of edges leaving C is adjacent from C. All components that are visited during the detection of C must be at some level below l, and are hence correctly detected. Thus, when all edges leaving the root r are processed, *cstack* contains no other components above $SavedHeight(r)$. Sorting *cstack* between $SavedHeight(r)$ and HEIGHT(*cstack*) into a topological order and eliminating duplicates does not change the situation. Thus, each component that is removed from *cstack* at line 23 is adjacent from C, and each component that is added into $Succ(C)$ at line 24 is a successor of C.

If: Let $X \neq C$ be a successor of C. If a tree or cross edge (v, w) leads from C to X, STACK_TC inserts X into *cstack* above $SavedHeight(r)$ at line 13. When STACK_TC has sorted *cstack* and eliminated duplicates, *cstack* contains one occurrence of X above $SavedHeight(r)$. Thus, STACK_TC removes X at line 23 and inserts it into $Succ(C)$. If no tree or cross edge (v, w) leads from C to X, another component Y exists such that a tree of cross edge (u, y) leads from C to Y, and X is a successor of Y. Since Y is at a level below l, $Succ(Y)$ is correctly computed and therefore contains X. When STACK_TC processes edge (u, y), it inserts Y onto *cstack*. When C is detected, STACK_TC removes Y from *cstack* and adds $Succ(Y)$, which contains X, into $Succ(C)$. □

Theorem 3.17. Algorithm STACK_TC runs in $O(sZ(n) + \overline{e}F(n) + \overline{e}_r(I(n) + U(n)) + \min(ns, e_{oct} \log n))$ time in the worst case. Here n, e, and s are the number of vertices, the number of edges, and the number of strong components in the input graph G, respectively, e_{oct} is the number of intercomponent tree and cross edges in G, $\overline{e}$ is the number of edges in the condensation graph $\overline{G}$ induced by the components of G (with the self-loop edges removed), and $\overline{e}_r$ is the number of edges in the transitive reduction of $\overline{G}$ (with the self-loop edges removed).

Proof. Sorting x components on *cstack* in the way we described above takes $O(\min(x + n, x \log x))$ time in the worst case. Let Π be the set of strong components in G. Let x_C be the number of components on *cstack* between the top and *SavedHeight*(r) when a component C with root r is detected. This is also the number of intercomponent tree and cross edges leaving the members of component C. Thus, $\sum_{C \in \Pi} x_C = e_{oct}$. The total time needed for sorting is

$$\begin{aligned} T_{\text{sort}} &= O(\sum_{C \in \Pi} \min(x_C + n, x_C \log x_C)) \\ &= O(\min(\sum_{C \in \Pi} (x_C + n), \sum_{C \in \Pi} x_C \log x_C)) \\ &= O(\min(e_{oct} + sn, \sum_{C \in \Pi} x_C \log x_C)) \end{aligned}$$

Since $x_C \leq e$ and $\log e \leq \log n^2 = 2 \log n$, we get

$$\begin{aligned} T_{\text{sort}} &= O(e_{oct} + sn, \sum_{C \in \Pi} x_C \log e) \\ &= O(\min(e_{oct} + sn, e_{oct} \log n)) \end{aligned}$$

The sorting removes all duplicates from *cstack*. The total number of components that remain is at most $\overline{e}$, since each remaining component is adjacent from C. These components are scanned at lines 22–25 and the presence of the components in *Succ*(C) is tested. This takes $O(\overline{e}F(n))$ total time. Since the components are scanned in a topological order, each component X that satisfies the test $X \notin Succ(C)$ at line 24 is the target of an edge in the transitive reduction of the condensation graph $\overline{G}$. To see this, suppose on the contrary that a component X satisfies the test and edge (C, X) is not in the transitive reduction of $\overline{G}$. By the definition of transitive reduction, edge (C, X) can be removed from $\overline{G}$ without changing the transitive closure of $\overline{G}$. Thus, $\overline{G}$ contains another path $p : C \xrightarrow{+} X$, which consists of at least two edges. Since all paths in $\overline{G}$ are topologically ordered, the first edge of p leads to a component Y that is topologically smaller than X. This implies that Y is removed from *cstack*, and Y and *Succ*(Y) are added into *Succ*(C) before X is removed from *cstack*. But since X is in *Succ*(Y), X is already in *Succ*(C) when X is removed from *cstack*, and the test $X \notin Succ(C)$ fails, which is a contradiction. Thus, the number of components that satisfy the test at line 24, and therefore the number of insertion and union operations, is at most $\overline{e}_r$. The insertion and union operations take $O(\overline{e}_r(I(n) + U(n)))$ total time. □

Corollary 3.18. If the successor sets are implemented as unordered lists augmented with a bit vector for membership lookups, algorithm STACK_TC runs in $O(n\overline{e}_r + \min(ns, e_{oct} \log n))$ time in the worst case.

In the unordered list representation augmented with a bit vector, we need only one bit vector, since at most one successor set is under construction at any moment.

Note that sorting the components on *cstack* slows the execution only by a term $\min(e_{oct} + sn, e_{oct} \log n)$, which is usually negligible compared to the other terms in the worst-case bound. Note also that $\min(ns, e_{oct} \log n) \leq n^2$.

3.5 Comparisons with previous algorithms

We have mentioned some properties of the previous transitive closure algorithms that are based on strong component detection [36, 40, 62, 64, 91, 101, 105]. Now we describe these algorithms more thoroughly and compare them analytically with our new algorithms. In Chapter 5, we compare the previous algorithms with our new algorithms experimentally.

We start by describing those algorithms that compute the successor sets during the strong component detection [36, 40, 64]. These algorithms generate a partial successor set for each vertex that is the tail of an edge. A common weakness in these algorithms compared to our new algorithms is that they build the successor sets from vertices and not from strong components. Our algorithm CR_TC, which also constructs the successor sets during the detection of the components, usually generates fewer partial successor sets than the previous algorithms, since CR_TC uses $Succ(Root(v))$ instead of $Succ(v)$ as the target of the insertions.

The oldest of these algorithms is Eve's and Kurki-Suonio's algorithm [40], which we here call EKS, for short. In EKS, the partial successor set of a vertex v contains the heads of the edges leaving v and the successors of the heads of the intercomponent edges leaving v. EKS does not propagate the partial successor sets via intracomponent edges towards the root vertex; instead, when EKS has detected a component C, it unions the partial successor sets of the members of C to get $Succ(C)$. A weakness in EKS is that it does not ignore intercomponent forward edges and does not check if a successor set S_1 is already present in the successor set S_2, where EKS is adding S_1. This leads to unnecessary union operations. Inserting the heads of all edges into the partial successor sets of their tails yields unnecessary insertions.

Ebert's algorithm [36], which we here call EBERT, resembles SIMPLE_TC. The improvement in EBERT compared to SIMPLE_TC and EKS is that EBERT does not use intercomponent forward edges to propagate successor sets. Apart from this optimization, EBERT suffers from the same weaknesses as EKS. Propagating the partial successor sets via intracomponent tree edges to the root vertex produces the same result as unioning all partial successor sets of component members in EKS.

The most recent algorithm that computes the successor sets during the strong component detection is the algorithm GDFTC by Ioannidis, Ramakrishnan, and Winger [64]. The algorithm is based on a complicated stack mechanism that contains two kinds of data: intracomponent successor lists and intercomponent successor lists. Like EBERT, GDFTC adds $Succ(w)$ into $Succ(v)$ if (v, w) is an intercomponent tree or cross edge. If (v, w) is a back edge, GDFTC stores a new stack frame on top of the stack. Successors are later added into the new frame. If (v, w) is an intracomponent tree edge and v is the target of a previously processed back edge, GDFTC merges the two topmost stack frames. If (v, w) is an intracomponent tree edge and v is not the target of a previously processed back edge, GDFTC adds $Succ(v)$ into the intercomponent successor list of the topmost stack frame and v into the intracomponent successor list of the

topmost stack frame. Unlike EBERT, GDFTC does not insert the head of each edge to the successor set of the tail of the edge. This seems to be the only improvement. Unfortunately, the stack mechanism is expensive. In the experiments by Ioannidis et al. [64], GDFTC was slower than Schmitz's algorithm [105] and algorithm BTC by Ioannidis et al.

The second group of the previous algorithms computes the successor set of a component only after detecting the component [105] or after detecting all components [64, 91, 101]. Except for the algorithm BTC [64], these algorithms build the successor sets from strong components instead of vertices.

The oldest of these algorithms is Purdom's algorithm [101], called PURDOM here, which we described already in section 2.3.2. Although the worst-case bound of PURDOM is good, $O(n\overline{e}_r + n^2)$, the algorithm has two weaknesses. First, the constant costs are high. The algorithm is seven pages of Algol-code and consists of four different phases. The graph is scanned several times. Second, the best-case execution time is $O(n^2)$. This is due to the underlying Boolean matrix representation.

Munro's algorithm [91], called MUNRO here, differs from the other algorithms in that it uses matrix multiplication to compute the transitive closure of the condensation graph. This leads to the same worst-case execution time $O(n^\alpha)$ as that of the multiplication of two matrices. When the input graph is acyclic and dense, this is the best worst-case bound for transitive closure computation that is known. Unfortunately, the constant costs are high.

Schmitz's algorithm [105], called SCHMITZ here, computes the successor set of a strong component immediately after detecting the component. The successor set contains strong components represented by their root vertices. When a component C with the root r is detected, SCHMITZ scans again all edges leaving the vertices of the component. For each edge (v, w), the algorithm checks if the root vertex r' associated with w equals r or if r' is already in $Succ(r)$. If the test fails, $Succ(r')$ is added into $Succ(r)$. The algorithm always inserts r' into $Succ(r)$. Thus, SCHMITZ needs e insertions whereas STACK_TC needs only $\overline{e}_r$ insertions. The other deficiency in SCHMITZ is that it requires two passes over the graph.

SCHMITZ usually requires more union operations than STACK_TC. Schmitz presented a variant of his algorithm, which minimizes the number of union operations by computing an *edge basis* before computing the successor sets. The edge basis corresponds to the transitive reduction of the condensation graph. The edge basis is computed in the following way: when a component is detected, the algorithm scans the edges leaving the component members and constructs a queue containing a subset of the roots of the edge heads. For an edge (v, w) such that w is in another component, the algorithm tests whether the queue already contains $Root(w)$ or another vertex x such that $Root(w)$ is in $Succ(x)$. If the test fails, the algorithm inserts $Root(w)$ into the queue. When the queue has been constructed, the algorithm scans it again and for each (root) vertex x in the queue checks if x is already in $Succ(r)$, the successor set that is being constructed. If x is not in $Succ(r)$, the algorithm adds $Succ(x)$ into $Succ(r)$. This variant needs only $\overline{e}_r$ union operations. Unfortunately, computing the edge basis often costs more than it saves.

Ioannidis et al. [64] presented an algorithm, called BTC, that computes the successor sets after all strong components have been detected. Strangely, BTC constructs the successor sets from vertices instead of strong components. BTC first uses a modified Tarjan's algorithm to detect the strong components and to assign exit numbers to the vertices. Then BTC processes

the vertices in the exit order and computes the successor sets. Since the exit order is a reverse topological order if we omit the back edges, the successor set of a component can be constructed by unioning the successor sets of the adjacent components, which are already constructed. Ioannidis et al. [64] described several optimizations that improve the performance of BTC in a paging environment. For instance, the adjacency lists can be topologically ordered during the first phase. The weaknesses in BTC are the high constant costs and the use of vertices instead of strong components as the building blocks of successor sets.

In Tables 3.1 and 3.2, we present the worst-case execution times of the previous algorithms and our new algorithms. The first table contains the implementation independent worst-case times and the second table the corresponding worst-case times when bit vectors, AVL-trees, or lists augmented with a bit vector are used to implement the successor sets. We use the following notations in the tables:

n	The number of vertices.
e	The number of edges; $e \leq n^2$.
s	The number of strong components; $s \leq n$.
s_{cyc}	The number of cyclic strong components; $s_{cyc} \leq s$.
$\overline{e}$	The number of edges in the condensation graph without the self-loop edges; $\overline{e} \leq e$.
$\overline{e}_r$	The number of edges in the transitive reduction of the condensation graph without the self-loop edges; $\overline{e}_r \leq \overline{e}$.
$\overline{e}^+$	The number of edges in the transitive closure of the condensation graph; $\overline{e} \leq \overline{e}^+$.
e_i	The number of intracomponent edges; $e_i \leq e$.
e_o	The number of intercomponent edges; $e_o = e - e_i \leq e$.
e_{i1}	The number of intracomponent tree edges (v, w) such that w is not in $Succ(Root(v))$ when BTC is checking the edge; $e_{i1} \leq e_i$.
e_{it}	The number of intracomponent tree edges; $e_{it} = n - s$ and $e_{it} \leq e_i$.
e_{o1}	The number of intercomponent tree edges (v, w) such that w is not in $Succ(Root(v))$ when BTC is checking the edge; $\overline{e}_r \leq e_{o1} \leq e_o$.
e_{o2}	The number of intercomponent edges (v, w) such that $Root(w)$ is not in $Succ(Root(v))$ when SCHMITZ checks the edge; $\overline{e}_r \leq e_{o2} \leq e_o$.
e_{oct}	The number of intercomponent tree and cross edges; $\overline{e}_r \leq e_{oct} \leq e_o$.
e_{oct1}	The number of intercomponent tree and cross edges (v, w) such that $Comp(w)$ is not in $Succ(Root(v))$ at line 11 in CR_TC; $\overline{e}_r \leq e_{oct1} \leq e_{oct}$.
n_1	The number of nonroot vertices with a non-empty partial successor set in CR_TC; $n_1 \leq n - s = e_{it}$.
σ	$\sum_{C \in \Pi} \sum_{v \in C} Outdeg(v) Outdeg(C)$, the time needed to compute the edge basis in Schmitz's variant algorithm. Here Π is the set of strong components and $Outdeg(C)$ is the outdegree of component C in the condensation graph (with the self-loop edges removed).
μ	$\min(ns, e_{oct} \log n)$, the time needed to sort the components in STACK_TC.
τ	The time needed to sort the adjacency lists of all vertices in a reverse topological order in BTC. The sorting method is not described in [64].
n^α	The complexity of matrix multiplication; $\alpha \leq 2.376$.
$Z(n)$	The maximum time of creating an empty set that can hold n elements.
$F(n)$	The maximum time of a membership test in a set of at most n elements.

Algorithm	Worst case bound $O()$
SIMPLE_TC	$nZ(n) + eI(n) + eU(n)$
EKS	$nZ(n) + eI(n) + (n - s + e_o)U(n)$
EBERT	$nZ(n) + eI(n) + (e_{it} + e_{oct})U(n)$
GDFTC	$nZ(n) + (e_{it} + e_{oct})(I(n) + U(n))$
CR_TC	$nZ(n) + e_{oct}F(n) + (s_{cyc} + e_{oct1})I(n) + (n_1 + e_{oct1})U(n)$
SCHMITZ	$sZ(n) + e_oF(n) + eI(n) + e_{o2}U(n)$
SCHMITZ with edge basis	$sZ(n) + (\overline{e}_r + e_i)I(n) + \overline{e}_rU(n) + \sigma F(n)$
BTC	$nZ(n) + eF(n) + (e_{i1} + e_{o1})I(n) + e_{o1}U(n) + \tau$
STACK_TC	$sZ(n) + \overline{e}F(n) + \overline{e}_r(I(n) + U(n)) + \mu$

TABLE 3.1: A summary of the implementation independent worst case times.

$I(n)$ The maximum time of inserting an element into a set of at most n elements.

$U(n)$ The maximum time of unioning of two sets of at most n elements.

As we see, the worst-case times of SIMPLE_TC, EKS, and EBERT are similar. The only difference is in the number of union operations. Obviously, SIMPLE_TC does at least as many unions as EKS and EBERT. In a graph, exactly one intracomponent tree edge leads to each nonroot vertex. Thus, $n - s = e_{it}$. Since $e_{oct} \leq e_o$, EBERT does always at most as many unions as EKS. GDFTC differs from EBERT only by doing fewer insertion operations. Note that the term $e \log n$ is present in the worst-case time of SIMPLE_TC with AVL-trees, but it is dominated by the term ne. Similarly, the term $(e_{oct} + e_{it}) \log n$ is present in the worst-case time of GDFTC with AVL-trees, but is dominated by the term $n(e_{oct} + e_{it})$.

Algorithm	Bit vector $O()$	AVL-tree $O()$	List&bit vector $O()$
SIMPLE_TC	$ne + n^2$	ne	$ne + n^2$
EKS	$ne_o + n^2$	$n(e_o + n - s) + e \log n$	$ne_o + n^2$
EBERT	$n(e_{oct} + e_{it}) + n^2$	$n(e_{oct} + e_{it}) + e \log n$	$n(e_{oct} + e_{it}) + n^2$
GDFTC	$n(e_{oct} + e_{it}) + n^2$	$n(e_{oct} + e_{it})$	$n(e_{oct} + e_{it}) + n^2$
CR_TC	$n(e_{oct1} + n_1) + n^2$	$n(e_{oct1} + n_1)$	$n(e_{oct1} + n_1) + n^2$
PURDOM	$n\overline{e}_r + n^2$	–	–
MUNRO	n^α	–	–
SCHMITZ	$ne_{o2} + ns$	$ne_{o2} + e \log n$	$ne_{o2} + \overline{e}^+$
SCHMITZ with edge basis	$n\overline{e}_r + ns + \sigma$	$n\overline{e}_r + (\overline{e}_r + e_i) \log n + \sigma \log n$	$n\overline{e}_r + \overline{e}^+ + \sigma$
BTC	$ne_{o1} + n^2 + \tau$	$ne_{o1} + e \log n + \tau$	$ne_{o1} + n^2 + \tau$
STACK_TC	$n\overline{e}_r + ns + \mu$	$n\overline{e}_r + \overline{e} \log n + \mu$	$n\overline{e}_r + \overline{e}^+ + \mu$

TABLE 3.2: A summary of the smallest implementation dependent worst case times.

The number of unions in CR_TC is $n_1 + e_{oct1}$. Since $n_1 \leq n - s = e_{it}$ and $e_{oct1} \leq e_{oct}$, CR_TC does at most as many unions as GDFTC and EBERT, and usually fewer. CR_TC needs e_{oct} membership tests to reduce the number of unions. The number of insertions in CR_TC and in GDFTC cannot, in general, be compared, since we do not know which one is greater, s_{cyc} or e_{it}. The best worst-case bound of CR_TC is $O(n(e_{oct1} + n_1))$ and is reached with AVL-trees. This is better than the best worst-case bound of GDFTC, namely $O(n(e_{oct} + e_{it}))$. Thus, CR_TC has the best worst-case bound of those algorithms that compute the successor sets during the detection of the strong components. Remember also that CR_TC constructs the successor sets from strong components instead of vertices. Therefore, the successor sets constructed by CR_TC are in practice much smaller and can be constructed much faster than the successor sets constructed by EKS, EBERT, and GDFTC.

Examine now the worst-case times of the algorithms that construct the successor sets after the components are detected. Note that no implementation independent worst-case times are presented for PURDOM and MUNRO, since the algorithms depend on the underlying bit matrix data structure. As we pointed out, when the input graph is acyclic and dense, MUNRO has the best worst-case bound of all transitive closure algorithms, but in a wide class of input graphs other algorithms have better worst-case bounds.

BTC and SCHMITZ need more unions than the other algorithms of this group. Which one of these two algorithms needs more unions depends on the input.

Schmitz's variant algorithm that uses the edge basis and our algorithm STACK_TC both need $\overline{e}_r$ unions. PURDOM effectively does the same number of unions, although the unions are open coded into bit matrix operations. Each of these algorithms does nontrivial computations to avoid the unnecessary unions. PURDOM explicitly builds the condensation graph and sorts it topologically, Schmitz's variant algorithm computes the edge basis, and STACK_TC sorts the adjacent components on *cstack* before constructing a successor set. Building the condensation graph takes $\Theta(n^2)$ time in PURDOM. Sorting the adjacent components in STACK_TC takes $O(\min(ns, e_{oct} \log n)$ time. $\mathrm{Min}(ns, e_{oct} \log n)$ is never greater than n^2 and in an infinite set of graphs it is negligible compared to n^2. Further, in an infinite set of graphs $n\overline{e}_r$ is negligible compared to n^2. Thus, STACK_TC has a better worst-case bound than PURDOM.

Computing the edge basis in Schmitz's variant algorithm takes in the worst case $O(\sum_{C \in \Pi} \sum_{v \in C} \mathit{Outdeg}(v)\mathit{Outdeg}(C))$ time. This sum cannot be expressed in a closed form, but the following example shows that the edge basis computation may take $\Omega(n^3)$ time even when the unions take only $O(n^2)$ time.

Example 3.9. Consider a complete DAG $G = (V, E)$ of n vertices 1,2, ..., n such that for each i and j, $1 \leq i, j \leq n$, G has an edge (i, j) iff $i < j$. Thus, $e = n(n-1)/2$. Since the graph is acyclic, the set of strong components $\Pi = \{\{i\} \mid i \in V\}$ and $\mathit{Outdeg}(i) = \mathit{Outdeg}(\{i\})$. When Schmitz's variant algorithm is constructing the edge basis for a component $\{i\}$, it scans again all edges leaving vertex i. Assume that the adjacency lists are in a reverse topological order. Thus, whenever the algorithm is checking an edge (i, j), j is not in the queue and neither is there any vertex k such that j is in $\mathit{Succ}(k)$. Hence the algorithm has to scan the whole queue and after that insert vertex j in front of the queue. The head of each edge leaving i is inserted into the queue this way. The number of queue positions that have to be checked when

constructing the queue for component $\{i\}$ is

$$P_i = \sum_{l=1}^{n-i-1} l = (n-i)((n-i)-1)/2 \tag{3.1}$$

The total number of queue positions checked during the computation is

$$P_{tot} = \sum_{i=1}^{n}(n-i)((n-i)-1)/2 = n(n-1)(n-2)/6 = \Omega(n^3) \tag{3.2}$$

$\overline{G}_r$, the transitive reduction of the condensation graph of G is a graph of n vertices and $n-1$ edges $(i, i+1)$, $1 \leq i < n$. Computing the transitive closure of $\overline{G}_r$ requires $n-1$ unions, one per each edge. Assuming that the union of two successor sets takes $O(n)$ time, the total time for the unions is $O(n^2)$. Since the time needed for sorting the adjacent components in STACK_TC is $O(\min(ns, e_{oct}))$, which is $O(n^2)$ in this example, STACK_TC needs only $O(n^2)$ time to compute the transitive closure of G, whereas Schmitz's variant algorithm needs $\Omega(n^3)$ time. □

Our conclusion is that STACK_TC has a better worst-case bound than the previous transitive closure algorithms that are based on strong component detection except MUNRO, which has a better worst-case bound with dense inputs. With sparse inputs STACK_TC has a better worst-case bound. Note also that MUNRO has high constant costs. In Chapter 5, we present experimental results showing that STACK_TC is in practice considerably faster than the previous algorithms.

Chapter 4

Representing successor sets

The transitive closure algorithms discussed in the previous chapter spend most of their execution time in constructing the successor sets. The successor set construction time is determined by the numbers of the different successor set operations needed and the execution times of the different successor set operations, i.e., membership lookups, insertions, and unions. In the previous chapter, we mainly concentrated on reducing the number of successor set operations. We reduced the time taken by the different successor set operations only by using strong components as the building blocks of the successor sets, which reduces the size of the successor sets and hence the execution times of set operations.

In this chapter, we study how the time needed to do the different successor set operations can be reduced by selecting an appropriate representation for the successor sets. We present two new successor set representations that require less memory space than common set representations such as bit vectors, lists, or ordered binary trees, and enable faster successor set operations. The first representation is based on intervals of reverse topological numbers of strong components and the second is based on decomposing the set of strong components into chains. Both representations are based on methods that have originally been developed for the compact representation of the transitive closure of acyclic graphs [4, 112]. We have modified these representations so that they can be used with our new transitive closure algorithms and also when the input graphs are cyclic. We explain how the representations reduce the redundant computations caused by multiple paths between pairs of vertices.

4.1 Properties of common set representations

Before presenting the representations, we examine some important properties of common set representations such as such as bit vectors, lists, or ordered binary trees.

When common set representations are used, a successor set S takes $\Omega(|S|)$ memory space. The size of the union of two sets S_1 and S_2 is $\Omega(\max(|S_1|, |S_2|))$. The transitive closure G^+ of a graph G has at least as many edges as G. The number of edges e^+ in G^+ may be considerably larger than the number of edges e in G, e.g., n^2 versus n when G is a simple cycle on n vertices and edges. When common set representations are used, the successor sets take $\Omega(e^+)$ space, and the best-case execution time is $\Omega(e^+)$.

The weakness in common set representations is that they do not take into account the structure of the transitive closure or the input graph to reduce the memory requirements.

4.2 Interval representation

We present now a new compact representation for successor sets that can be used in our transitive closure algorithms. Contrary to the set data structures discussed above, this representation often enables a set A to be stored in memory space smaller than $\Omega(|A|)$. Further, when this representation is used, the union of two sets A and B may be smaller than either of its arguments. The transitive closure can usually be stored in much smaller than $O(e^+)$ space and the best-case execution time is not bounded by $\Omega(e^+)$. The representation is based on a method for compressing the transitive closure of an acyclic graph [4].

The representation consists of two parts: a method for storing sets of integers compactly and a method for mapping the strong components into integers.

We describe first the method for storing sets of integers compactly. Consider a set $S \subseteq \{1, 2, \ldots, n\}$. If S consists of one or more sequences of consecutive integers $i, i+1, \ldots, i+m$, where $m > 2$, a compact way to represent S is to store only the endpoints of each maximal consecutive sequence. We call such a representation the *interval representation*. An interval $[i, j]$ represents the set $\{i, i+1, \ldots, j\}$, and a collection of intervals $\{I_1, I_2, \ldots, I_r\}$ represents the union of the sets that the intervals represent. When new intervals are added to a set, overlapping intervals are merged together. Also, if two intervals $[i, j]$ and $[j+1, k]$ exist, they are combined to an interval $[i, k]$.

Example 4.1. 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 $[0, 10]$. □

It is easy to see that the maximum number of intervals that we need to store a set of integers $S \subseteq \{1, 2, \ldots, n\}$ is $\lceil n/2 \rceil$. This upper bound is reached when $S = \{1, 3, 5, \ldots, 2\lfloor (n-1)/2 \rfloor + 1\}$. The expected number of intervals needed is harder to define and it depends on the distribution of the integers in S. We present next the expected number of intervals needed when the distribution of the integers is even.

Theorem 4.1. Let $S \subseteq \{1, 2, \ldots, n\}$ and $|S| = k$. Assuming that each integer $i \in \{1, 2, \ldots, n\}$ has the same probability of being a member of S, the expected number of intervals needed to store S is $k - k(k-1)/n$.

Proof. To produce set S, we draw k different elements from $\{1, 2, \ldots, n\}$ with each element having the same probability of being drawn. Let $I_{n,k}$ denote the expected number of intervals thus created. To get a recursion formula for $I_{n,k}$, we examine the case that element n is in S and the case that element n is not in S. The probability of the former case is k/n and the probability of the latter case is $(n-k)/n$. Assume first that n is in S. The expected number of intervals needed is $I_{n-1,k-1} + x$, where $I_{n-1,k-1}$ is the expected number of intervals needed to store a $k-1$ element set $S' \subseteq \{1, 2, \ldots, n-1\}$ and x is the expected number of additional intervals needed to store element n. If element $n-1$ is in S', no additional intervals are needed, but if $n-1$ is not in S', one additional interval is needed. Thus, x is the probability that element $n-1$ is not in S', which is $(n-1-(k-1))/(n-1) = (n-k)/(n-1)$. Assume now that n is not in S. The expected number of intervals needed is $I_{n-1,k}$, the expected number

of intervals needed to store a k element set $S'' \subseteq \{1, 2, \ldots, n-1\}$. Finally, if $n = 0$, $k = 0$, or $k > n$, no intervals are needed, and if $n > 0$ and $k = 1$, one interval is needed. We get the following recursion formula:

$$I_{n,k} = \begin{cases} 0 & n = 0 \text{ or } k = 0 \text{ or } k > n; \\ 1 & n > 0 \text{ and } k = 1; \\ \frac{k}{n}\left(I_{n-1,k-1} + \frac{n-k}{n-1}\right) + \frac{n-k}{n} I_{n-1,k} & \text{otherwise.} \end{cases} \tag{4.1}$$

Examining $I_{n,k}$ with small values of n and k yields:

$$\begin{array}{lcll} I_{n,1} & = & 1 & 1 \le n \le 10 \\ I_{n,2} & = & 2 - 2/n & 2 \le n \le 10 \\ I_{n,3} & = & 3 - 6/n & 3 \le n \le 10 \\ I_{n,4} & = & 4 - 12/n & 4 \le n \le 10 \\ I_{n,5} & = & 5 - 20/n & 5 \le n \le 10 \end{array} \tag{4.2}$$

Thus, it seems that $I_{n,k} = k - k(k-1)/n$, when $n > 0$ and $0 < k \le n$ and $I_{n,k} = 0$ otherwise. This can be checked using the recursion formula. When $n > 0$ and $k = 1$, we get:

$$I_{n,1} = 1 = 1 - 1 \cdot 0/n \tag{4.3}$$

When $n > 0$ and $0 < k \le n$, we get

$$\begin{aligned} I_{n,k} &= \frac{k}{n}\left(I_{n-1,k-1} + \frac{n-k}{n-1}\right) + \frac{n-k}{n} I_{n-1,k} \\ &= \frac{k}{n}\left(k - 1 - \frac{(k-1)(k-2)}{n-1} + \frac{n-k}{n-1}\right) + \frac{n-k}{n}\left(k - \frac{k(k-1)}{n-1}\right) \\ &= k - \frac{k(k-1)}{n} \quad \square \end{aligned} \tag{4.4}$$

The maximum value of $I_{n,k} = k - k(k-1)/n$ is

$$\max(I_{n,k}) = \begin{cases} I_{n,\frac{n+1}{2}} = \frac{n}{4} + \frac{2n+1}{4n} & n \text{ is odd;} \\ I_{n,\frac{n}{2}} = \frac{n}{4} + \frac{1}{2} & n \text{ is even.} \end{cases} \tag{4.5}$$

Thus, with some choices of k, we need $\Omega(n)$ space to store a k element set $S \subseteq \{1, 2, \ldots, n\}$ if each element of $\{1, 2, \ldots, n\}$ has the same probability of being a member of S. If the probability distribution is not even, the expected number of intervals is different. Note that if the integers in set S are likely to be near some value x, the number of intervals needed is probably small. The expected number of intervals needed is then much smaller than $k - k(k-1)/n$.

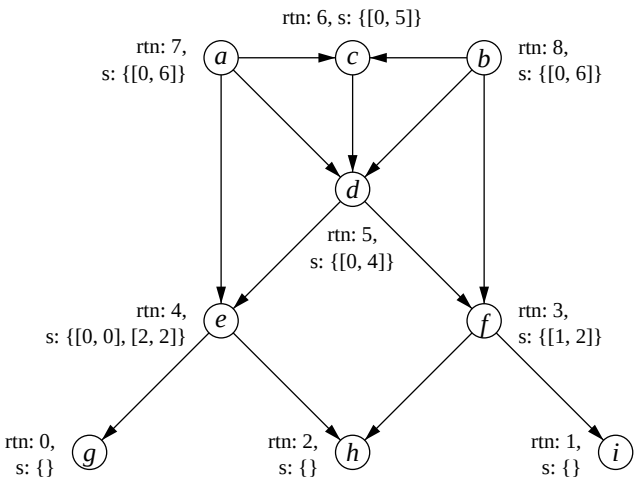


FIGURE 4.1: A DAG and the successor sets of its vertices represented as intervals.

We examine next the second component of our new representation, a method to map the strong components into integers in a way that produces compact interval collections. The mapping that we use is simple: the integers that the intervals contain are reverse topological numbers of the strong components. The strong components are numbered increasingly in the order they are detected. With this numbering, we usually need much fewer intervals to represent the successor sets than with a random numbering of components. The reasons for this are twofold. First, all successor components of a component C have numbers smaller than C. Hence the elements in $Succ(C)$ come from $\{1, 2, \ldots, k\}$, where k is the reverse topological number of C, not from $\{1, 2, \ldots, n\}$. Second, a subtree in the spanning tree induced by the depth-first search is represented as a single interval. Note that the numbering depends on the order in which the components are detected, and many orders are possible. Different numberings yield different numbers of intervals to represent the successor sets.

Example 4.2. In Figure 4.1, we present an acyclic graph and the successor sets of its vertices as sets of intervals produced by one possible traversal of the graph, namely g, f, i, h, c, d, e, a, b. For each vertex, the figure presents the reverse topological number (denoted by "rtn:") and the successor set as a collection of intervals (denoted by "s:"). In this traversal, the maximum number of intervals needed to represent a successor set is 2, and the total number of intervals needed is 7. □

The next example shows that with this numbering method also, we may need $\Omega(n^2)$ space to represent the successor sets of a graph.

Example 4.3. In Figure 4.2, we present an example graph of $n = 2m + 1$ vertices. An edge leads from vertex 0 to edges $1, \ldots, m$. An edge leads from each vertex $m + 1, \ldots, 2m$ to the vertices $1, 3, 5, \ldots, m$ (we assume that m is odd). Examine what happens when we compute the transitive closure of this graph using the interval representation and a transitive closure algorithm that is based on Tarjan's algorithm. Assume that the execution starts at vertex 0, and that the adjacency list of 0 is in order $1, 2, \ldots, m$. Each vertex i, $1 \leq i \leq m$, is in its own strong component, and the reverse topological number of the component is i. The successor set of 0 consists of a single interval $[1, m]$. When the algorithm later constructs the successor set for a vertex j, $m + 1 \leq j \leq 2m$, the successor set consists of $(m + 1)/2$ intervals. Thus, we need $\Omega(m^2) = \Omega(n^2)$ intervals. □

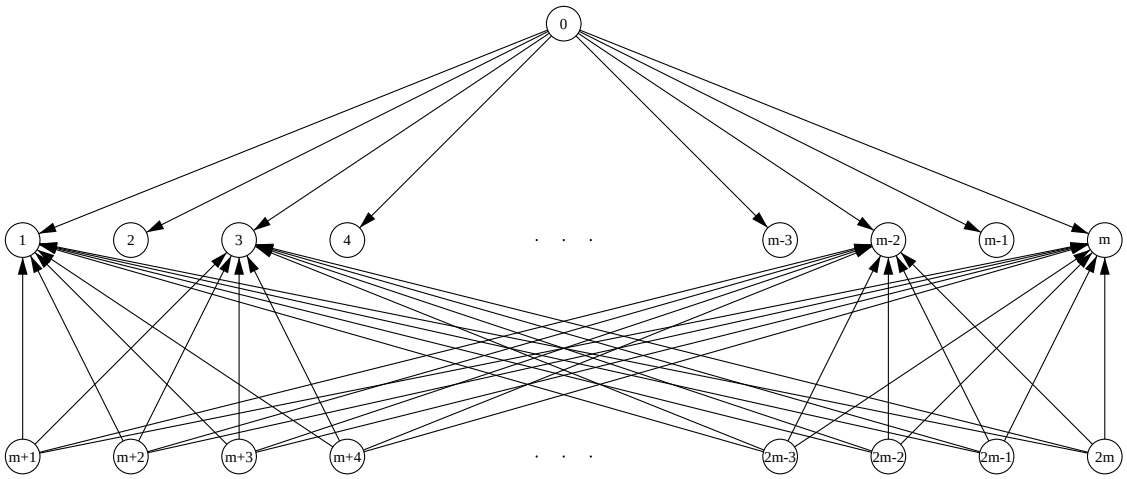


FIGURE 4.2: A graph the successor sets of which may require $\Omega(n^2)$ space.

The average number of intervals that is needed to represent the successor sets of a graph is difficult to analyze, since the number of intervals depends on the topology of the graph and not only on the number of vertices and edges of the graph. Further, different traversals of the same graph yield different numbers of intervals. Another problem is choosing an appropriate model of inputs. In section 5.2, we examine empirically the average size of the representation.

A collection of intervals representing a successor set 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 of intervals 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. Let S, S_1, and S_2 be successor sets that are represented as collection of intervals I, I_1, and I_2, respectively. The union of S_1 and S_2 takes $O(|I_1|+|I_2|)$ time. Testing the presence of an element x in set S takes $O(\log(|I|))$ time in the worst case if the intervals are stored as a balanced binary tree or as 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 $O(\log(|I|))$ in the worst case if the intervals are stored as a balanced binary tree and $O(|I|)$ time if the intervals are stored as an unbalanced binary tree or an ordered array. The binary tree representation requires twice as much memory space as the ordered array representation. The memory allocation is simpler for the array representation. In some transitive closure algorithms, e.g., in our algorithm STACK_TC, at most one successor set is under construction at any moment. In these algorithms, it is a good idea to use the array representation at least for all completely constructed successor sets. The successor set that is under construction can be stored as a binary tree to speed up insertion operations. If the number of intervals needed to represent a successor set is usually low, it is probably wise to store all successor sets as ordered arrays of intervals.

The idea of the interval representation came from [4], where Agrawal et al. studied how the transitive closure of an acyclic graph can be stored in a compressed form. The stored transitive closure can be used to directly answer queries about successors of a vertex without the need for computing the successors on-the-fly. Another goal they had was to incrementally propagate changes to the stored closure when the base relation changes. Since their goals were different from ours, their strategy also was different. Our goal here is the efficient computation of the full transitive closure, and a compact representation is a means to reach this goal. Agrawal et

al. [4], on the other hand, tried to represent the transitive closure as compactly as possible, and the efficiency of the computation of the transitive closure was not important. Another difference is that their inputs were restricted to acyclic graphs whereas our inputs may have cycles. Our method overcomes this problem by detecting the strong components during the traversal.

In the method by Agrawal et al. [4], an optimal spanning tree is first computed. The input graph is sorted topologically and then the vertices are considered in the topological order. For a vertex v, that incoming edge (u, v) whose source vertex u has the largest number of predecessors, is included into the spanning tree. The resulting spanning tree yields a representation having the smallest number of intervals in their setting.

Next, the spanning tree is traversed in postorder, and an interval cover is computed. A subtree T_r rooted at vertex r is represented as an interval $[p_v, p_r]$, where p_v is the smallest postorder number in T_r, and p_r is the postorder number of r.

Finally, the vertices of the input graph are processed in the reverse topological order, and the sets of intervals are propagated via cross edges from the target of the edge to the source of the edge. When an interval $[i_1, j_1]$ is added to a set S containing an interval $[i_2, j_2]$, such that $i_1 \leq i_2$ and $j_2 \leq j_1$, interval $[i_2, j_2]$ is removed. Similarly, if $i_2 \leq i_1$ and $j_1 \leq j_2$, interval $[i_1, j_1]$ is omitted. On the other hand, adjacent intervals $[i, j]$ and $[j + 1, k]$ are not merged. Agrawal et al. [4] told that the additional compression obtained by merging usually was less than 5%. Since no adjacent intervals are merged, no pairs of overlapping intervals $[i_1, j_1]$ and $[i_2, j_2]$, such that $i_1 < i_2 < j_1 < j_2$, exist. Note that in our method the merging of adjacent and overlapping intervals is essential, since our single traversal strategy does not necessarily yield an optimal spanning tree.

Example 4.4. Consider the acyclic graph presented in Figure 4.1. In Figure 4.3, we present this graph and its transitive closure again, but now the successor sets are computed using the method of Agrawal et al. [4]. For each vertex, the figure presents the interval that covers the vertex in the spanning tree (denoted by "i:") and the successor set as a collection of intervals (denoted by "s:"). The edges that are in the spanning tree generated by the algorithm are drawn solid and the other edges (cross edges) dashed. □

The method by Agrawal et al. [4] needs several passes over the input graph, whereas our method requires just one traversal. According to our experience, computing the optimal spanning tree is not necessary. Our method usually yields an equally small representation by merging the adjacent intervals.

4.3 Chain representation

The chain representation is another compact way to represent the successor sets of a transitive closure. Originally, this representation was presented for acyclic graphs [67, 90, 111, 112]. An important property of the chain representation is that it allows a better worst-case bound for the transitive closure computation of an acyclic graph than the traditional successor set representations. We show here how this representation can be used together with our transitive closure algorithms and also when the inputs are cyclic.

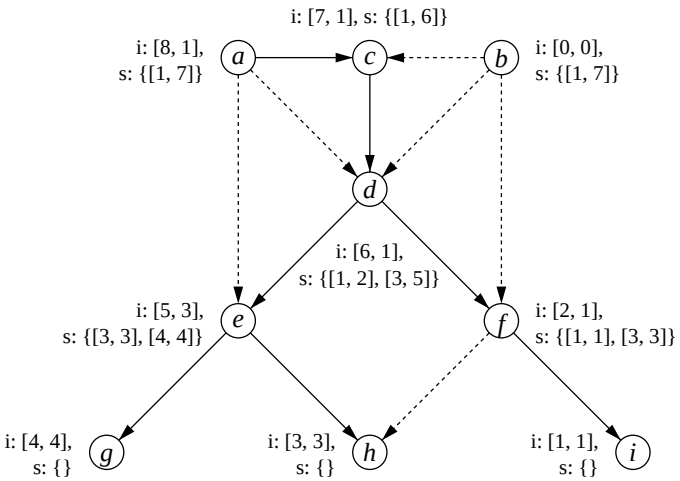


FIGURE 4.3: The interval representation of the graph in Figure 4.1 when the method of [4] is used.

The chain representation is an example of a representation that is based on partitioning the vertices of the input graph into a collection of disjoint subsets. Let $\Pi = \{C_i \mid C_i \subseteq V\}$ be a partition of the vertex set V, $\bigcup_{C_i \in \Pi} C_i = V$. Let S be a successor set. We can 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 [45] were the first to use this partitioning approach. In their algorithm, the partitioning is based on *topological levels* of the acyclic input graph. If a vertex v has no predecessors, it is at level zero. If a vertex v has predecessors and if l is the highest level where some of its immediate predecessors is located, then v is at level $l + 1$. A successor set is represented as a collection of intersections $S \cap L_i$, one intersection per each topological level L_i. Unfortunately, no compact way to store the intersections exists. An intersection is represented as a list of its member vertices. We need a partition that permits a more compact representation of the intersections.

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

Obviously, an acyclic graph can be partitioned into chains in many different ways. The number of chains varies in different partitions. Simon [111, 112] used the topological ordering of the input graph to compute the chains. The vertices and the adjacency lists are first topologically ordered. A new chain C_i is computed by first choosing the topologically smallest unprocessed vertex v and adding it to C_i. Then the topologically smallest unprocessed child of v is entered recursively. This continues until no unprocessed child is found. The topological ordering and the computation of the chains take $\Theta(n + e)$ time. If the partition contains k chains, a successor set is represented as a vector of k elements. The union of two successor sets takes $\Theta(k)$ time. The union of sets S_1 and S_2 is computed by scanning their vectors and by taking for each chain C_i the topological minimum of the vertices representing the intersections $S_1 \cap C_i$ and $S_2 \cap C_i$. Checking the presence of a vertex v in a successor set S takes constant

time. This is done in the following way: find the chain C containing v and the vertex w that represents the intersection $C \cap S$. Vertex v is in S iff v is w or v is topologically greater than w. Inserting an element into a successor set is done in a similar way and also takes a constant time.

Simon [111, 112] presented an algorithm that computes the transitive closure in a way that resembles the third part of Purdom's algorithm [101]. The vertices are considered in a reverse topological order. The children of a vertex v are processed in a topological order and if a child u is not already in $Succ(v)$, then u and $Succ(u)$ are added to $Succ(v)$. In this way, we need one union per each edge in the transitive reduction of the input graph. Since one union takes $\Theta(k)$ time, where k is the number of chains, the worst-case execution time of the algorithm is $\Theta(ke_r)$. Simon [111, 112] also showed that the average number of chains is $O(\ln(np)/p)$ in a randomly generated acyclic graph $G(n,p)$, where n is the number of vertices, and each edge (i,j), $i<j$, exists with probability p independently of other edges.

Jagadish [67] presented several ways to construct the chains. He showed that a minimal partition can be computed by the minimum flow algorithm. Unfortunately, this takes $O(n^3)$ time, i.e., it is usually slower than the transitive closure computation. Jagadish presented a collection of heuristics that can be used to construct the chains. His experiments indicated that the topological ordering approach by Simon [111, 112] usually yields a smaller partition than the other heuristics. In Jagadish's representation, a successor set S is represented as a collection of pairs (C,v), where C is a chain and v is the topologically smallest element of C that is in S. According to our experience, this usually requires much less memory space than the k element vector representation by Simon [111, 112].

Our approach to constructing the chains is the following. The elements in the chains are the strong components that are detected in a reverse topological order in our transitive closure algorithms. When a component C is detected, all components adjacent from it are already assigned to a chain. The procedure in Figure 4.4 finds a chain for component C.

```
(1)   procedure ASSIGN_TO_CHAIN(C);
(2)   begin
(3)       for each component X adjacent from C in topological order do
(4)           if TOP(CHAIN(X)) = X then begin
(5)               PUSH(C, CHAIN(X));
(6)               return
(7)           end
(8)       ch := create a new chain;
(9)       PUSH(C, ch)
(10)  end
```

FIGURE 4.4: Procedure ASSIGN_TO_CHAIN finds a chain for component C.

The components adjacent from C are processed in the reverse 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 storing C on top of the chain. If no such component is found, i.e., no component adjacent from C is the topmost element of the chain containing it, we create a new chain and store C on top of that chain.

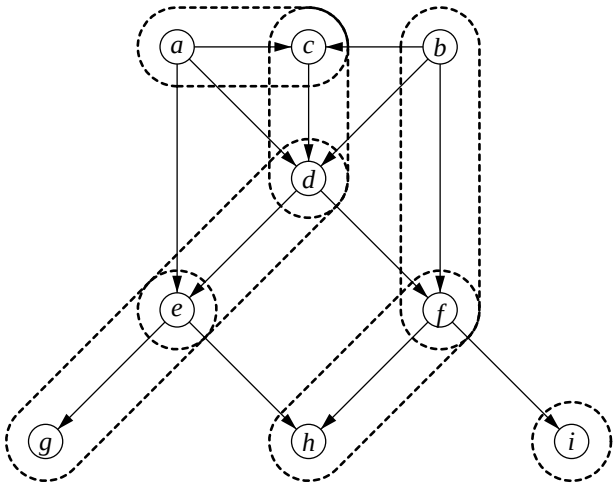


FIGURE 4.5: The chain partition of a graph when Simon's method [111, 112] is used.

Like Jagadish [67], we represent a successor set S as pairs (C, v), where C is a chain and v is the topologically smallest element of C that is in S. We cannot use the k element vector representation by Simon [111, 112], since the chains are constructed simultaneously with the successor sets, and we do not know the total number of chains required. We could use variable length vectors to represent the sets, but as we have pointed out, it seems that the pair representation saves much space compared to the vector representation.

Example 4.5. In Figure 4.5, we present the graph of Figure 4.1 and the chain partition of its vertex set computed by Simon's topological ordering method [111, 112]. In Figure 4.6, we present the chain partition computed by our reverse topological ordering method. We assume that the vertices are traversed in the order g, f, i, h, c, d, e, a, b. In this example, both methods need three chains, which is the optimal number of chains for this graph. □

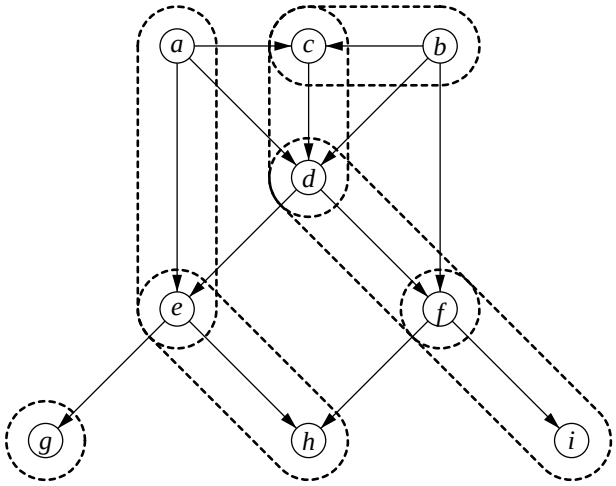


FIGURE 4.6: The chain partition of a graph when our method is used.

4.4 Avoiding multiple paths

In section 1.1, we pointed out two sources of redundancy that should be avoided when computing the transitive closure. So far, we have only discussed the redundancy caused by strong components. We consider now the redundancy caused by multiple paths between two vertices and show how the interval representation efficiently copes with this kind of redundancy.

Assume that a transitive closure algorithm that is based on Tarjan's algorithm is applied to the graph in Figure 4.7. Two paths go from a to f, namely a, b, d, f and a, c, e, f. The depth-first traversal uses both paths, but vertex f and the subtree rooted at f are entered only once. Thus, it seems that the algorithm avoids well the redundancy caused by multiple paths. Unfortunately, the successor set of f is propagated from f to a via both paths and is twice added to the successor set of a.

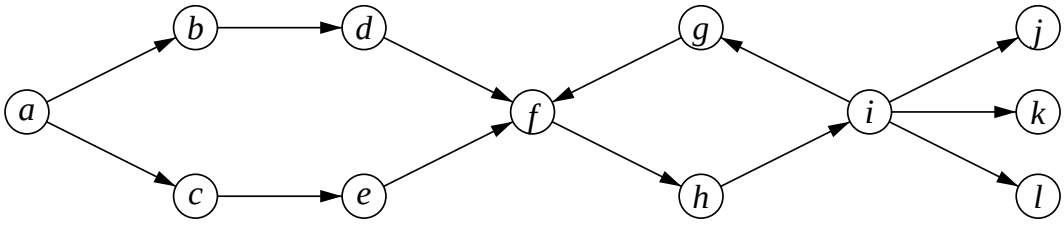


FIGURE 4.7: A directed graph with multiple paths between vertices.

Jakobson [70] presented a transitive closure algorithm that avoids this kind of redundancy. The idea is to use successor trees instead of successor sets. A successor tree contains the same vertices as a successor set, but in addition it contains the paths that were used to obtain these vertices. Multiple paths are detected and the same vertices are not added again to the successor tree. The cost that has to be paid is the space required by the successor trees and the time needed to construct them. Another problem is that the method does not detect strong components. Dar and Jagadish [33] presented a transitive closure algorithm that uses a similar representation.

When common set data structures such as ordered lists or relation tables are used to represent the successor sets, taking the union of two sets is the most time consuming operation. Adding the same successor set twice to another set should be avoided. But when we use the interval representation, a union operation usually means processing only a few intervals. Adding a few intervals twice to a successor set is not expensive. Interval merging often further reduces the costs.

The following example is taken from Jakobson's paper [70]. The graph in Figure 4.8 has $n = m^2$ vertices that are ordered as a square of m columns of m vertices. An edge leads from each vertex in column k to each vertex in column $k + 1$. Thus, each vertex in column k has all vertices in columns $k + 1, k + 2, \ldots, m$ as its successors. The size of the successor set is $\Theta(m^2)$ for most vertices.

Assume that we apply CR_TC to this graph. Since the graph has no cycles, each vertex has an own component. To compute the successor set of a component x in column k, we must insert the components in column $k + 1$ into $Succ(x)$. Further, we must add the successor sets of the components in column $k + 1$ into $Succ(x)$. Although these sets are equal, we must union

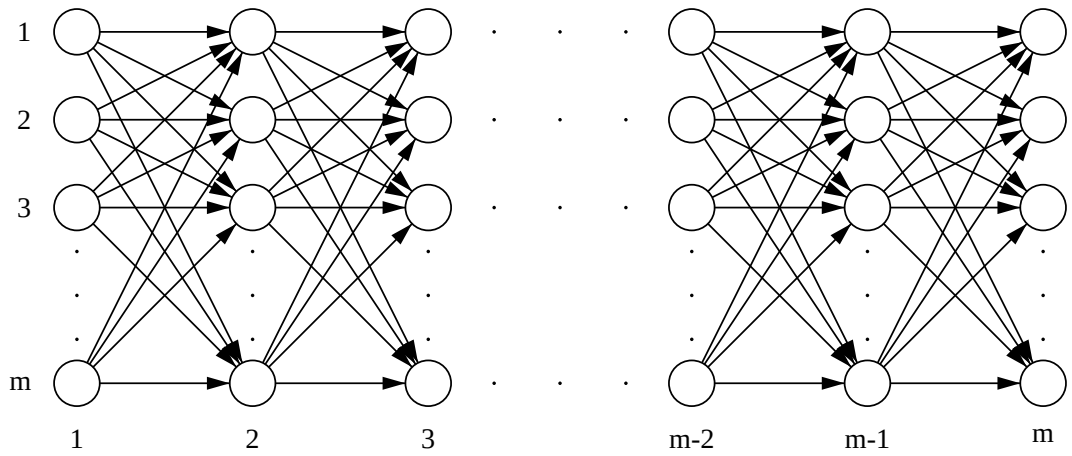


FIGURE 4.8: An example from Jakobson's paper[70].

m sets of $\Theta(m^2)$ components. This takes $\Omega(m^3)$ time when common set data structures are used. Since we have m^2 components, the computation takes $\Omega(m^5) = \Omega(n^2\sqrt{n})$ time.

Jakobson's algorithm requires only $O(n^2)$ time, since it uses only one path from a vertex x in column k to a vertex y in column $k+2$.

Assume next that CR_TC is used together with our interval representation. When we traverse the graph in any depth-first order, the components in column m get the reverse topological numbers $1, 2, \ldots, m$. Similarly, the components in column k get the numbers $m(m-k)+1, m(m-k)+2, \ldots, m(m-k+1)$.

The successor set of a component in column k is the single interval $[1, m(m-k)]$. To construct the successor set of a component C in column $k-1$, we add the reverse topological numbers and the successor sets of the components in column k into $Succ(C)$. Thus, we insert m numbers $m(m-k)+1, m(m-k)+2, \ldots, m(m-k+1)$ into $Succ(C)$ and we insert m times the interval $[1, m(m-k)]$ into $Succ(C)$.

If the intervals are stored in ordered balanced binary trees, the insertions require $O(m \log m)$ time. Since the graph has m^2 vertices, the total time required is $O(m^3 \log m)$ which is $O(n\sqrt{n} \log n)$. Thus, the interval representation is here more efficient than Jakobson's successor tree method.

Assume finally that CR_TC is used together with our chain representation. Our algorithm creates m different chains of m components, each containing one component from each column. The successor set of a component C in column $k-1$ is represented as m topological numbers of the m components in column k. To construct $Succ(C)$, we must compute $m-1$ unions of successor sets, each of size m, and insert m components into the set. This takes $O(m^2)$ time. Since the graph has m^2 components, we need the total of $O(m^4) = O(n^2)$ time to compute the transitive closure, which is the same time as with Jakobson's method.

Chapter 5

Simulations

In this chapter, we present computer simulations that we did to study the average-case performance of transitive closure algorithms and representations. First, in section 5.1 we describe the method that we used in our experiments. The presentation of the actual simulation experiments is divided into three sections: in section 5.2, we study memory requirements of different transitive closure representations, in section 5.3 the amount of work needed to construct the successor sets in different algorithms and representations, and in section section 5.4 the execution times of different transitive closure algorithms.

5.1 Method

In this section, we describe the method that we used in our performance experiments. We first explain why we decided to use computer simulations instead of mathematical analysis as the performance evaluation technique. Then we discuss the issues that have to be considered in computer simulations and describe the inputs that we used in the simulations. We end the description of the method by discussing the overall design of performance evaluation studies.

5.1.1 Mathematical algorithm analysis

Mathematical analysis of the worst-case execution time is the most common method for studying the performance of algorithms. It is usually easy, since we only have to determine the set of possible inputs, decide which inputs cause the algorithm to spend most time, and compute the corresponding execution time. The problem with the worst-case analysis is that it often gives overly pessimistic results. The algorithm may run much faster with typical inputs than in the worst case. Comparing the efficiency of two algorithms only by their worst-case execution times may yield misleading conclusions. For instance, many transitive closure algorithms have approximately the same worst-case time, but these algorithms behave differently with typical inputs.

To get a more realistic picture of the performance of an algorithm with typical inputs, we can analyze its average-case performance. Unfortunately, mathematical average-case analysis is much more difficult than worst-case analysis. We have to create a more sophisticated model of inputs. Determining the set of possible inputs is not enough: we have to assign each input a probability of occurrence. It is not obvious what these probabilities should be. To make

the processing of probabilities easier, we may have to choose an overly simplified model of inputs that may differ greatly from the inputs in practical applications. Given an algorithm and an input model, computing the average-case execution time is usually much more difficult than computing the worst-case execution time. As a general rule, average-case analysis using realistic models of input is beyond the limits of the current analytical methods [88]. The average-case analysis is often easier for probabilistic algorithms, i.e., algorithms that use the distribution of the underlying input model in some way in their behavior. If the inputs of the algorithms are simple combinatorial structures like binary trees or sequences of integers, a divide-and-conquer strategy may be possible.

In analyzing the average-case performance of transitive closure algorithms, we face all these problems. The selection of the input model is not obvious. We discuss this problem in the next section. The algorithms that we have presented 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 performance of the algorithms.

The average-case execution time of a transitive closure algorithm is analyzed in only a few previous articles. Bloniarz et al. [18] analyzed the average-case execution time of a simple transitive closure algorithm that traverses the input graph n times starting from each vertex in turn. The average-case bound of this algorithm is $O(n^2 \log n)$. Schnorr [106] presented a probabilistic algorithm that computes the transitive closure in $O(n + e^+)$ average-case time, where e^+ is the number of edges in the transitive closure. The algorithm has high constant costs, since it traverses the input graph several times to obtain both the vertices adjacent from and the vertices adjacent to each vertex and to sort these vertices lexically. Karp [79] presented another probabilistic transitive closure algorithm. When the expected outdegree of the graph is slightly greater than one, the asymptotic average-case execution time of the algorithm is $O((n \log n)^{4/3} \omega(n))$, where $\omega(n)$ is any non-decreasing function. Otherwise, the average-case execution time is $O(n)$. Also Karp's algorithm has high constant costs, since it consists of three separate algorithms that are executed simultaneously step-by-step until one of them completes. The algorithm needs a constant time access both to the vertices adjacent from and the vertices adjacent to each vertex. If only the vertices adjacent from each vertex are available, as we generally have expected in this thesis, Karp's algorithm needs an additional $\Theta(n + e)$ time to compute the vertices adjacent to each vertex. Both Schnorr's and Karp's algorithms depend heavily on the distribution of the underlying input model. It is not clear how well the algorithms behave with other kinds of inputs.

5.1.2 Performance evaluation by simulation

Computer simulation is another method for studying the average-case performance of an algorithm [88]. The technique seems to be straight-forward: we implement the algorithm augmented with commands to collect performance measures, then we apply the program on randomly generated inputs, and finally we compute the average of the performance measures. The benefit of this approach is that it can be used even when mathematical average-case analysis fails. It also yields more accurate results than mathematical analysis. The deficiency of this approach is that the results are valid only in that area of the space of inputs from which we

have selected the inputs. Simulations give no asymptotic results. After a simulation study, we cannot say that the run time of the algorithm is $O(f(n))$ in the average case. Note also that a simulation study usually takes more time than mathematical analysis (when the latter is possible).

In selecting the input model for a simulation study, we face partially the same problems as in mathematical average-case analysis. We must choose the set of possible inputs and assign a probability of occurrence to each input. A benefit of the simulation approach is that we can study many input models using the same simulation program. All we need is a new procedure for generating the new inputs and more simulation runs. In simulation studies, we can usually apply more complicated input models than in mathematical analysis. However, in simulation studies, the generation of inputs must be fast whereas in analytical studies we have no such requirement. We consider the selection of the input model and the generation of inputs for our simulation studies in the next subsection.

The measurement data of a simulation study must be properly treated to get reliable results. Since the inputs of the simulation are usually controlled by random variables with some distribution, also the outputs are usually random variables with some other distribution. In many studies presented in the literature, much effort is put in the implementation of a sophisticated simulation program, but the program is applied to only a few arbitrary selected inputs and the mean of the measured values is presented as the average-case performance metric. The problem is that this approach gives no information on the accuracy of the results. As Pawlikowski [99] said, these kinds of studies are merely programming exercises.

The theory of estimation in statistics gives us tools for analyzing the measurement data. We present here only those parts of estimation theory that we have used in our simulations, see any textbook of statistics, e.g., [35], for a more thorough presentation. Our presentation is based on references [35, 83, 84, 99].

In estimation, we expect that a random sample $(x_1, x_2, \ldots, x_n)$, for instance a series of measurement data, is the realization of a sequence of random variables $X_1, X_2, \ldots, X_n$, which have the same distribution $F(x)$ and are independent. The goal is to find values of properties that describe distribution F. In our study, we want to know μ_x, the expected value of the distribution. The sample mean

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{5.1}$$

can be used as the estimator of μ_x.

A standard statistical technique to assess the accuracy of an estimate is to construct a *confidence interval* for the estimated parameter. A $100(1-\alpha)\%$ confidence interval for a parameter θ is an interval $[L_1, L_2]$ that contains the actual value of θ with probability $(1-\alpha)$. The confidence interval is constructed using the sample $(x_1, x_2, \ldots, x_n)$. If the random variables X_i are normally distributed, then the confidence interval of the expected value is $[\bar{x} - \Delta_x, \bar{x} + \Delta_x]$, where Δ_x can be computed using Student's t-distribution and the estimator $\hat{\sigma}^2[\bar{x}]$ of the variance of X as follows:

$$\Delta_x = t_{n-1,1-\alpha/2}\hat{\sigma}[\bar{x}] \tag{5.2}$$

$$\hat{\sigma}^2[\bar{x}] = \frac{1}{n(n-1)}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{5.3}$$

Given a sample, we can compute different confidence intervals by changing the confidence level $1-\alpha$. Obviously, the higher the confidence level is, the wider the confidence interval.

These equations can also be used when random variables X_i are not normally distributed or when we do not know that they are. The necessary requirement is that the random variables are independent and have an identical distribution. When n, the size of the sample, increases, the random variable $X = X_1+X_2+\ldots+X_n$ tends to the normal distribution and the estimator of the confidence interval is again valid. In practice, the estimator gives good approximation when $n > 30$.

Equations 5.2 and 5.3 can be used after the simulation to compute a $100(1-\alpha)\%$ confidence interval of the expected value. The problem is that before the simulation, we do not know what kind of a confidence interval the simulation will yield. If we fix the number of simulation inputs, i.e., the size of the sample, beforehand, we may either produce confidence intervals that are too wide or too uncertain to be useful or waste simulation time for processing unnecessary inputs. This problem can be solved by using *sequential procedures* [83, 84, 99]. In a sequential procedure, the length of the simulation run is increased until the desired confidence interval is reached. Before the simulation starts, we only have to describe the confidence interval that we would like to have. The simulation procedure produces new inputs and applies the simulation program to them until the desired confidence interval is reached.

We can describe the desired confidence interval, and thus the stopping condition of the simulation, either in absolute or in relative terms. In the former case, we usually state that $|\Delta_x| \leq \delta_{\max}$, i.e., the error of our estimate must be at most $\delta_{\max}$. In the latter case, we have several possibilities. In the methods presented in [83, 84, 99], the $100(1-\alpha)\%$ confidence interval produced by the simulation should be such that the ratio of its half-length to its mid-point does not exceed a fixed constant, i.e., $|\Delta_x/\bar{x}| = \epsilon \leq \epsilon_{\max}$. A shortcoming in these methods is that they do not directly limit the relative error $|(\bar{x}-\mu_x)/\mu_x|$ of the expected value μ_x. Pawlikowski [99] showed that if $[\bar{x}-\Delta_x, \bar{x}+\Delta_x]$ is a $100(1-\alpha)\%$ confidence interval produced in this way, then

$$1-\alpha \leq P[\frac{|\bar{x}-\mu_x|}{|\mu_x|} \leq \frac{\epsilon}{1-\epsilon}] \tag{5.4}$$

We can use this inequality to get a direct limit for the relative error of μ_x. If we replace $\epsilon_{\max}$ by $\epsilon_{\max}/(1+\epsilon_{\max})$ in the stopping condition, then the relative error $|(\bar{x}-\mu_x)/\mu_x|$ is at most $\epsilon_{\max}$. The length of the simulation increases slightly, but according to our experience this causes no problems.

Our sequential simulation procedure is presented in Figure 5.1. As inputs, the simulation procedure needs values for parameters α and $\epsilon_{\max}$ that control the confidence interval. In addition, it needs a procedure A that implements the algorithm which is studied, a procedure G that generates a random input for procedure A, and $p_1, p_2, \ldots, p_k$, the values of parameters that control the generation of the inputs, e.g., the number of vertices in the generated random graph. The random inputs that procedure G yields must be independent. Procedure A applies the algorithm which is studied to the given input and computes the associated performance measure.

The output of the sequential simulation procedure is the estimate $\bar{x} = s/n$ of the expected value μ_x of the performance measure. The relative error of $\bar{x}$ to μ_x is at most $100\epsilon_{\max}\%$ with

```
(1)   s := 0;
(2)   n := 0;
(3)   repeat
(4)       I := G(p_1, p_2, ..., p_k);
(5)       x := A(I);
(6)       s := s + x;
(7)       n := n + 1;
(8)   until |Δ_x/x̄| ≤ ε_max/(1 + ε_max)
```

FIGURE 5.1: A sequential simulation procedure.

probability $1 - \alpha$. Ganguly et al. [44] used a similar procedure in their study of transitive closure computation.

Since we did not know that the performance measures are normally distributed, we generated at least 30 inputs in each simulation. When the expected value of the measure was near zero in some parts of the input space and large in other parts of the input space, the fulfillment of the relative stopping condition required long simulation runs. Knowing that the expected value is near zero in some part of the input space is usually sufficient when the expected value is large in other parts of the input space. Thus, in these experiments we used a stopping condition that is a combination of a relative and an absolute condition. The simulation was stopped when either the relative error or the absolute error became sufficiently small.

5.1.3 Inputs

Since we had no particular application in mind when choosing the input model, we first decided to use the model of directed random graphs $G(n, p)$ that is mostly used in the literature [19, 79]. The model is defined as follows:

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

Although all possible directed graphs can be generated in this model, the graphs drawn from $G(n, p)$ are usually similar in some respects. Karp [79] showed that when n is large and np (the expected outdegree) is equal to a constant c greater than one, it is likely that the graph contains one large component, called the *giant component*, and the other components are small. When n is constant and np grows, the percentage of vertices in the giant component increases rapidly. Our experiments showed that also when n is small, e.g., $n = 100$, the giant component grows rapidly when np grows. In Figure 5.2, we present the results of a simulation that measured the expected value of the percentage of vertices in the largest component of the graph. Parameter n varied between 100 and 50000, and np varied between 0.0 and 8.0. In these simulations, $\alpha = 0.05$ and $\epsilon_{\max} = 0.05$. As we see, the proportion of vertices that were in the giant component was almost independent of n. When np was less than one, no giant component existed. When np was two, about 60% of the vertices were in the giant component, and when np was greater than four, almost all vertices were in the giant component. This

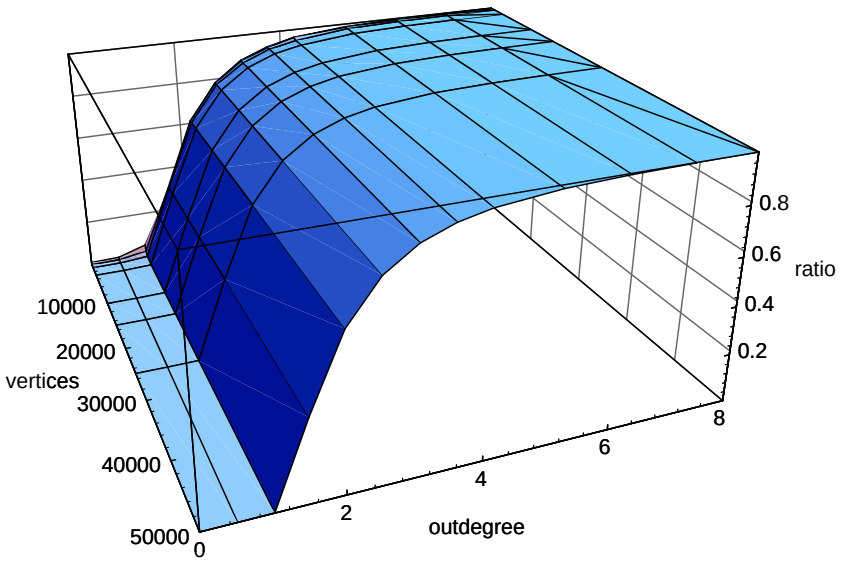


FIGURE 5.2: The percentage of vertices in the largest component in model $G(n,p)$.

phenomenon has a well-known counterpart in the theory of undirected random graphs [19]: also undirected random graphs $G(n,p)$ usually contain one large connected component and the other components contain only a few vertices.

Karp [79] showed also that a component that has more than one vertex, but that is not the giant component, is unlikely. In Figure 5.3, we present the percentage of vertices that are outside the giant component and the trivial components. As we see, almost all vertices are either in the giant component or in trivial components.

If we would have used $G(n,p)$ as our only simulation model, we would have gained no information on graphs with many nontrivial components. To overcome this problem, we decided to also use another model of directed random graphs $G(n,p,l)$, defined as follows:

Definition. The model $G(n,p,l)$ of random graphs is a probability space that contains all directed graphs having the same vertex set $V = \{1,2,\ldots,n\}$ and an edge set $E \subseteq \{(i,j) \mid i,j \in V \wedge j \in [(i-l) \bmod n, \ldots, (i+l) \bmod n]\}$. Each possible edge of a graph in $G(n,p,l)$

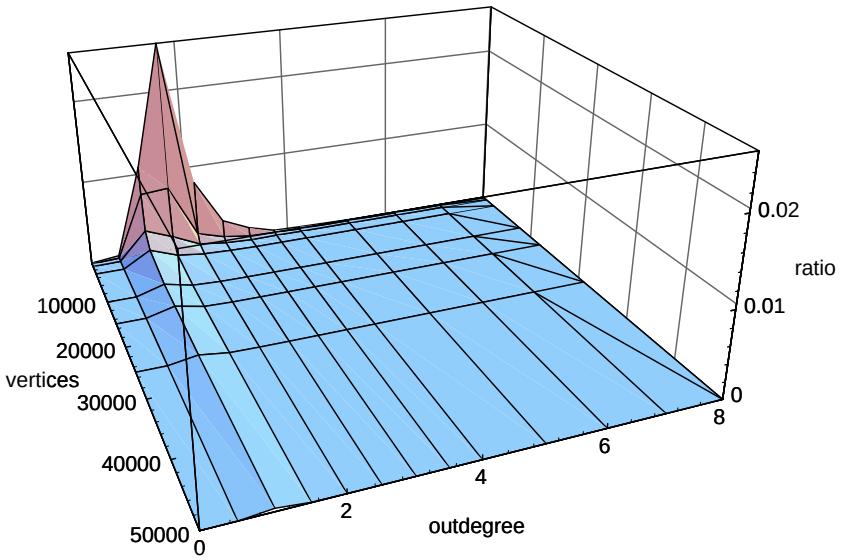


FIGURE 5.3: The percentage of vertices outside the largest component and the trivial components in model $G(n,p)$.

exists with probability p independently of the existence of any other edge in the graph; thus, the probability of a given graph $G = (V, E)$ with e edges is $p^e(1-p)^{n(2l+1)-e}$.

This means that edges leaving a vertex i can only go to a vertex j that is between $(i - l) \bmod n$ and $(i + l) \bmod n$ in the modular arithmetic. Parameter l is called the *locality*.

In Figure 5.4, we present the expected percentage of vertices outside the largest component and the trivial components when the graphs were drawn from $G(n, p, l = 5)$. When the expected outdegree $(2l + 1)p$ was between two and four, most vertices were in nontrivial components other than the giant component. The behavior depended only on the outdegree, not on the number of vertices. When the expected outdegree approached five, the graphs became strongly connected. We got similar results with other small values of l, e.g., $l = 10$. When l increased, $G(n, p, l)$ started to resemble $G(n, p)$. Thus, we can consider $G(n, p)$ to have the greatest possible locality. 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. [64] used a similar model in their simulations.

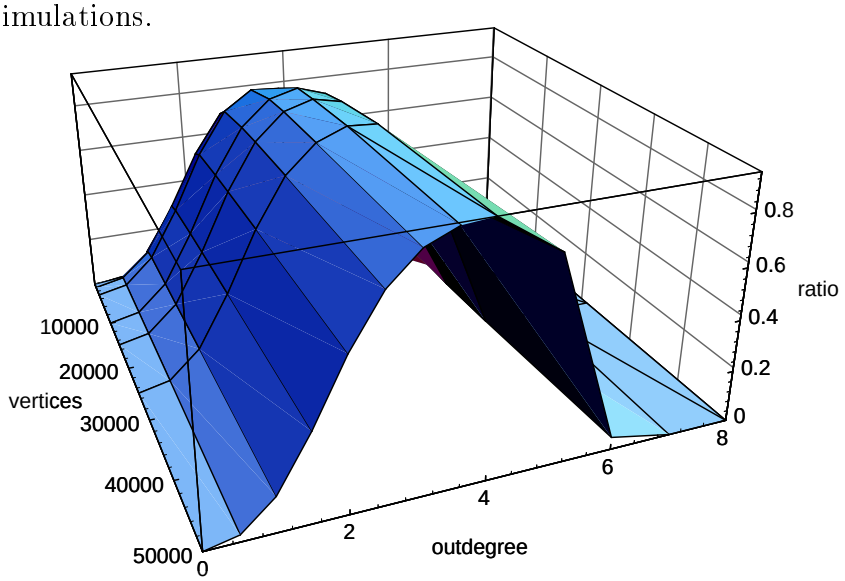


FIGURE 5.4: The percentage of vertices outside the largest component and the trivial components in model $G(n, p, l = 5)$.

Generating inputs

Since the algorithms that we study are efficient, generating the inputs may dominate the execution time in the simulations. We describe below how random graphs of model $G(n, p)$ can be efficiently generated. Random graphs of model $G(n, p, l)$ can be generated with only slight modifications.

By definition, we could generate a random graph $G = (V, E)$ of $G(n, p)$ as follows: for each possible edge (i, j), draw a uniformly distributed random number u, $0 \leq u \leq 1$. If $u \leq p$, then insert (i, j) into E. The weakness in this method is that we always need $\Omega(n^2)$ time, even when the generated graph has $O(n)$ edges.

A simple probability theoretic observation makes it possible to use only $O(n + e)$ time to generate G [78]. In generating the edges, we do n^2 independent 0–1 trials, where the probability of 1 is p in each trial. By definition, the distance between two consecutive trials that yield 1 is geometrically distributed with parameter p. Thus, we generated the edges of G by using the

algorithm presented in Figure 5.5.

```
(1)   x := 0;
(2)   while true do begin
(3)       x := x + geom(p);
(4)       if x > n^2 then
(5)           return
(6)       else
(7)           insert (⌈x/n⌉, x mod n + 1) into E
(8)   end
```

FIGURE 5.5: An algorithm for generating the edges of a random graph of model $G(n, p)$.

The procedure draws $e+1$ geometrically distributed random numbers and generates one new edge per each of these random numbers except the last. Since the edge generation procedure above yields the edges in ascending order, we randomized the order of edges in the adjacency lists after the generation.

We generated geometrically distributed random numbers efficiently by drawing a uniformly distributed random number u, $0 \leq u \leq 1$, and converting it into a geometrically distributed random number x by an inverse transformation [100]

$$x = \lceil \ln(u) / \ln(1 - p) \rceil \tag{5.5}$$

In generating the uniformly distributed random numbers, we used a linear-congruential generator

$$u_i + 1 = 7^5 u_i \bmod (2^{31} - 1) \tag{5.6}$$

The same generator is used, for instance, in IBM's SIMPL/1 [59] simulation package and the mathematical library IMSL [60]. This generator was highly recommended by Park and Miller [98] in their study of random number generators.

5.1.4 Performance evaluation design

Above we have discussed three important issues in a performance evaluation study: selecting the evaluation technique, coping with the variance of performance metrics, and selecting the inputs. We discuss now some other important issues that must be considered in designing a performance evaluation study.

Jain presented in his book [69] an overall approach to performance evaluation studies that consists of ten steps. We list these steps below and explain how the steps apply to our experiments.

1. *State the goals of the study and define the system boundaries.* If this step is missed, it is unlikely that any useful results are obtained in the study. Our goals were simple, e.g., to compare the number of operations in different algorithms. The system boundaries were mostly obvious.

2. *List services and possible outcomes.* A computer system that is evaluated offers a set of services. For instance, a computer network allows its users to send packets to specified destinations. Each service has several possible outcomes, some desirable and some undesirable. For instance, the packets may be delivered to the correct destination in the computer network or may get lost. In our experiments, the evaluated system offered a single service, computing the transitive closure of the given graph, and the service had only desirable outcomes.

3. *Select performance metrics.* The performance metrics are the criteria to compare the performance. We used several different metrics: the number of specific operations, the size of a data structure in number of elements, and the execution time in seconds.

4. *List system and workload parameters.* All parameters that affect the performance of the evaluated system should be identified. Workload parameters characterize the inputs that are given to the system. System parameters characterize the software and the hardware. We have above described the input models that we used in our experiments and listed the parameters used in generating the inputs. Most system parameters were related to the software, e.g., the algorithms and data structures that were used.

5. *Select factors and their levels.* Those parameters that are varied in the experiment are called factors. The values of the factors are called levels.

6. *Select the evaluation technique.* The three categories of evaluation techniques are analytical modeling, simulation, and direct measuring of a real system. We explained above why we selected simulation as our main evaluation technique. In the execution time experiments, we used direct measuring.

7. *Select workload.* The workload consists of a list of service requests to the evaluated systems. Our workloads were random graphs drawn from $G(n,p)$ and $G(n,p,l)$.

8. *Design experiments.* We should select a sequence of experiments that offer maximal information with minimal effort. In our experiments, we first ran a few initial simulation runs with inaccurate confidence intervals to get an overall picture of the behavior of the system. Then we selected appropriate levels for the input factors and ran the simulations again with more accurate confidence intervals. Thus, the selection of the levels of the input factors and the experimental design were connected to each other.

9. *Analyze and interpret data.* Since we used the sequential simulation procedure, no additional statistical analysis of the result was needed. In some studies, we used a regression model to fit curves to the data.

10. *Present results.* We present the results of the experiments in graphic form using two or three dimensional plots.

5.2 The size of transitive closure representations

5.2.1 Experimental setting

Our goal was to compare the average size of different transitive closure representations. We wanted to find out which representations need the smallest and the largest memory space on the average in different input models. We also wanted to find out how the average memory requirements vary with different input parameter levels. The representations were

1. Successor lists that contain vertices when components are not detected, i.e., each vertex has a separate successor list. We call this representation the $v \times v$ *-representation*, since its size corresponds to the size of the transitive closure relation represented as pairs of vertices.

2. Successor lists that contain vertices when components are detected, i.e., each component has a single successor list and the list contains vertices. We call this representation the $c \times v$ *-representation*, since its size corresponds to the size of the transitive closure relation represented as component–vertex pairs.

3. Successor lists that contain components when components are detected, i.e., each component has a single successor list and the list contains components. We call this representation the $c \times c$ *-representation*, since its size corresponds to the size of the transitive closure relation of the condensation graph represented as pairs of components.

4. Interval representation, presented in section 4.2.

5. Chain representation, presented in section 4.3.

We used the number of required memory words as the performance metric. We assumed that in a list representation one vertex or component takes one word of memory space. A pair in the chain representation or an interval in the interval representation takes two words. No other system parameters were used.

We computed the sizes of the different representations by first computing the transitive closure using algorithm STACK_TC and the interval representation and then computing the sizes of the $v \times v$ -representation, $c \times v$ -representation, and $c \times c$ -representation by using the interval representation and the strong components. Since the interval representation is time and space efficient, this approach made it possible to use large input graphs. Unfortunately, the size of the chain representation cannot be computed directly from the interval representation. Therefore, we also had to compute the transitive closure using the chain representation.

The workload consisted of randomly generated sparse graphs drawn from $G(n, p)$ and $G(n, p, l)$. We did not study dense inputs, since the behavior of each representation is obvious when the inputs are dense: if the strong components are not detected, we usually need quadratic memory space, and if components are detected, we usually need linear or smaller than linear memory space.

The workload factors were the number of vertices n, the expected outdegree d, and the locality l. The edge probability p of the input model was computed from d using n and l. After some initial simulation runs, we selected the following levels for the factors: factor n

had seven different levels 100, 300, 1000, 3000, 10000, 25000, and 50000, factor d had levels ranging from 0.0 to 6.0 by 0.1, and factor l had levels 5, 10, and 20. We could not use vertex counts $n = 25000$ and $n = 50000$ in model $G(n, p)$ with the chain representation, since the representation required too much memory.

The simulation control parameters were $\alpha = 0.1$ and $\epsilon_{\max} = 0.1$ or $\delta_{\max} = n/100$, i.e., we computed a 90% confidence interval I such that either the relative error of I is at most 10% or the absolute error of I is at most $n/100$. This combined stopping condition was needed, since the sizes of some representations are small in some parts of the input space and large in other parts of the input space.

5.2.2 Results

We study how the average size varied in different representations when the input model and the parameter levels were varied. We present some results using three dimensional plots with the number of vertices n and the expected outdegree d on the x- and y-axis. The average size is presented on the z-axis as a *representation size ratio*, i.e., the average number of memory words required divided by the number of vertices. Examine carefully the scale of the z-axis; since the maximum average size varied much in different representations and also in different input models with the same representation, we cannot use the same scale on the z-axis in all graphs. We also describe the dependency of the maximum average sizes on the number of vertices n. We estimated these dependencies from the simulation results using a regression model.

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

We study first 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 levels 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 5.6), 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

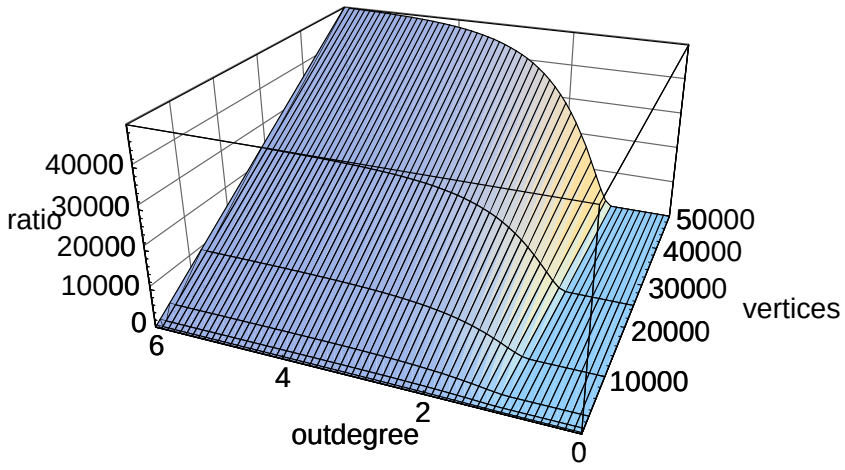


FIGURE 5.6: The representation size ratio of the $v \times v$ -representation in model $G(n, p)$.

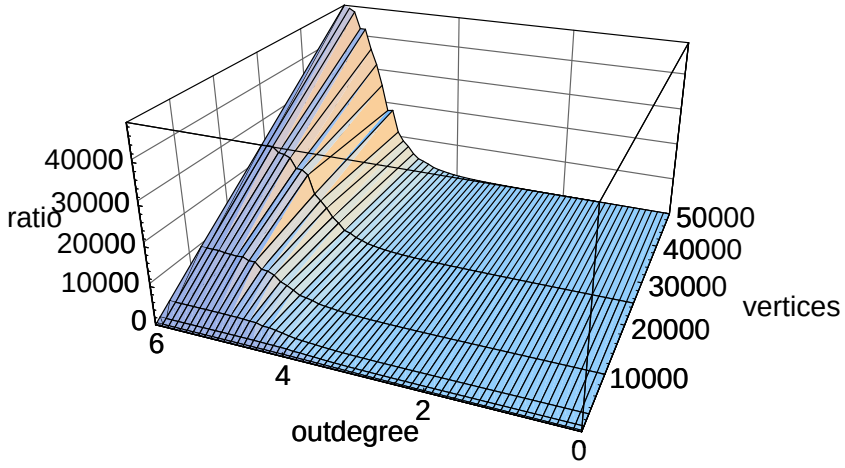


FIGURE 5.7: The representation size ratio of the $v \times v$ -representation in model $G(n, p, l = 5)$.

$G(n, p, l = 5)$ between outdegrees five and six (Figure 5.7). Thus, the rapid growth started earlier with greater values of locality l.

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

The size of $c \times v$ -representation first increased monotonically as d grew and then decreased monotonically. The difference between the behavior in different models was that the greater the locality was, the greater the maximum average size and the earlier the maximum was reached. In model $G(n, p, l = 5)$, the maximum average size was about $1.2n^{1.3}$ words (reached at $d \approx 5$), in model $G(n, p, l = 10)$ about $0.40n^{1.7}$ words (reached at $d \approx 4$), and in model $G(n, p, l = 20)$ about $0.15n^{1.9}$ words. In model $G(n, p)$, the maximum average size was about $0.15n^2$ words (reached at $d \approx 1.7$ (Figure 5.8)). Thus, the dependency of the maximum average size on the number of vertices seemed to become quadratic when the locality grew.

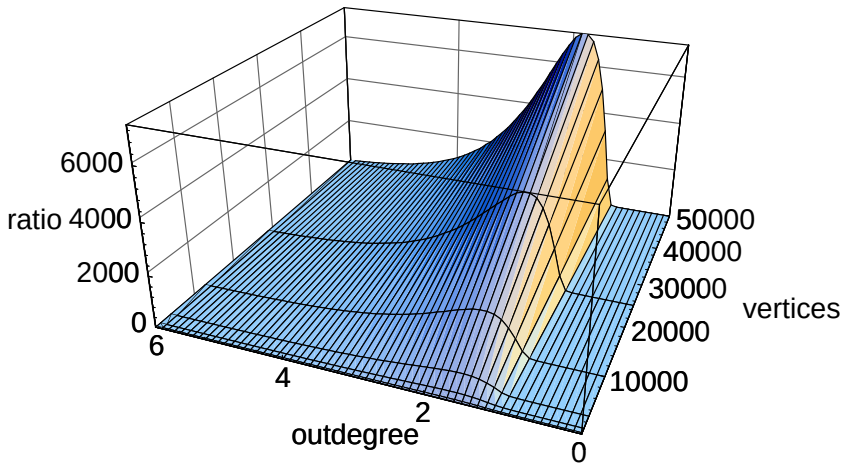


FIGURE 5.8: The representation size ratio of the $c \times v$ -representation in model $G(n, p)$.

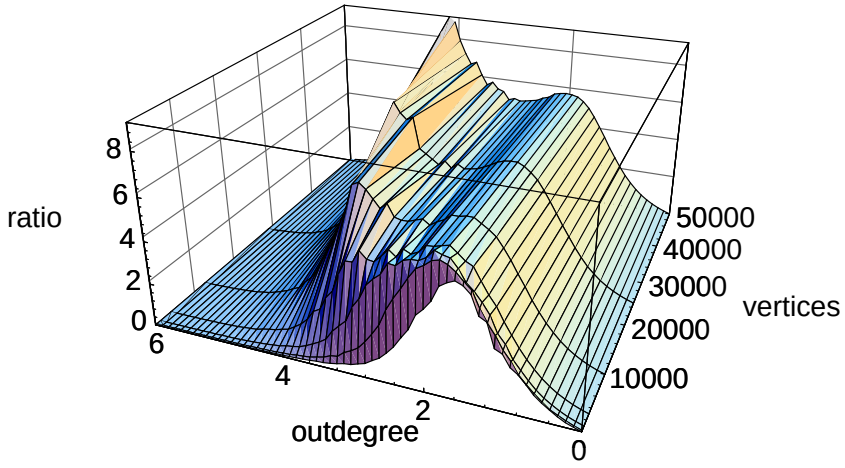


FIGURE 5.9: The representation size ratio of the $c \times c$ -representation in model $G(n, p, l = 10)$.

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

The $c \times c$ -representation behaved like the $c \times v$ -representation in models $G(n, p)$ and $G(n, p, l = 20)$, except that the maximum average sizes were smaller (about $0.063n^2$ words in $G(n, p)$ and about $0.11n^{1.7}$ words in $G(n, p, l = 20)$) and were reached at lower levels of d. In model $G(n, p, l = 10)$, the behavior was more complicated (Figure 5.9). 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 $0.61n^{1.3}$ 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 5.10). The maximum average size was about $2.8n$ words.

The average size of the chain representation

The chain representation behaved like the $c \times c$ -representation in all models, except that the average sizes were mostly about twice as large as in the $c \times c$ -representation.

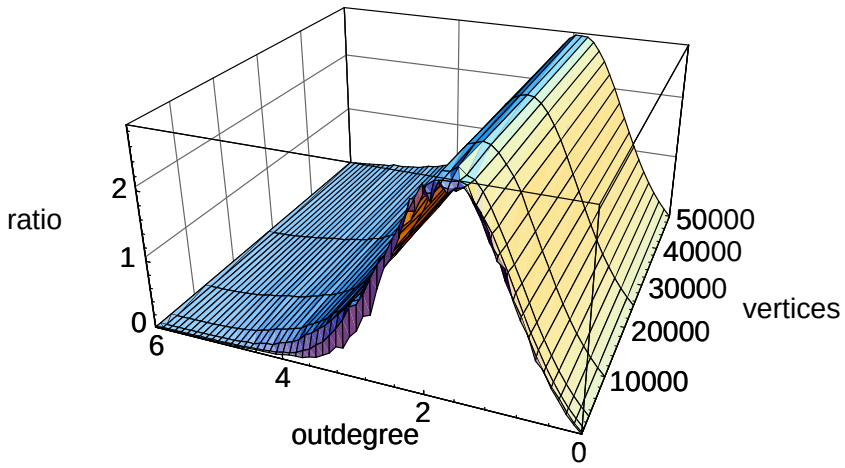


FIGURE 5.10: The representation size ratio of the $c \times c$ -representation in model $G(n, p, l = 5)$.

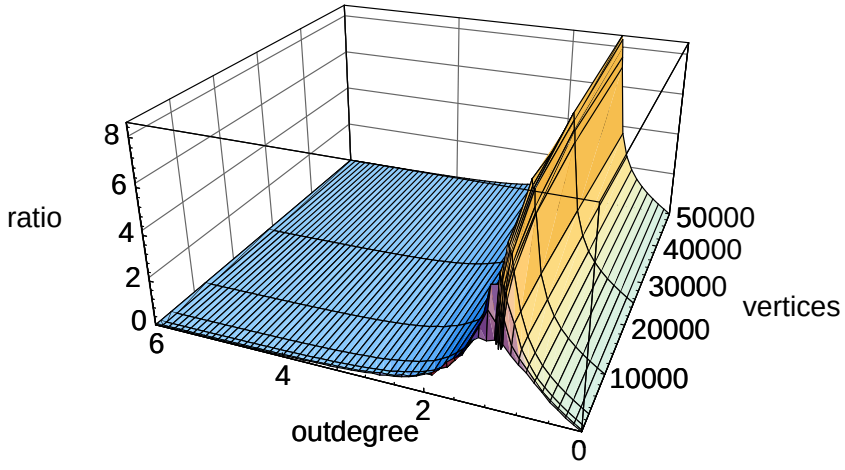


FIGURE 5.11: The representation size ratio of the interval representation in model $G(n,p)$.

The average size of the interval representation

The interval representation behaved in models $G(n,p,l)$ with all three values of l like the $c \times c$ -representation in model $G(n,p,l=5)$. The average size first increased when the outdegree grew and then decreased, and the maximum average size grew linearly to the number of vertices. The maximum average size was about $2.2n$ words in $G(n,p,l=5)$, about $2.6n$ words in $G(n,p,l=10)$, and about $3.3n$ 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 $0.55n \log n$ words (Figure 5.11). The maximum average size was also reached with smaller values of d.

Table 5.1 shows the maximum average sizes of the representations at levels $n = 10000$, $n = 25000$, and $n = 50000$. In addition, it shows the estimated growth of the maximum average size as a function of n.

5.2.3 Discussion

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 memory 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. As expected, the $c \times v$ -representation required less memory than the $v \times v$ -representation, but more than the $c \times c$ -representation. Contrary to our initial expectations, the chain representation saved no space compared to the $c \times c$ -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 $0.55n \log n$ 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.

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
$c \times v$	none	1.5×10^7	9.2×10^7	3.7×10^8	$0.15n^2$
	$l = 20$	6.5×10^6	3.2×10^7	1.2×10^8	$0.15n^{1.9}$
	$l = 10$	2.1×10^6	8.8×10^6	2.9×10^7	$0.4n^{1.7}$
	$l = 5$	250,000	820,000	2.1×10^6	$1.2n^{1.3}$
$c \times c$	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	$0.11n^{1.7}$
	$l = 10$	65,000	200,000	460,000	$0.61n^{1.3}$
	$l = 5$	29,000	71,000	140,000	$2.8n$
Chains	none	6.9×10^6	–	–	$0.069n^2$
	$l = 20$	930,000	4.0×10^6	1.3×10^7	$0.13n^{1.7}$
	$l = 10$	110,000	350,000	820,000	$1.1n^{1.3}$
	$l = 5$	28,000	70,000	140,000	$2.8n$
Intervals	none	73,000	200,000	430,000	$0.55n \log n$
	$l = 20$	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	$2.2n$

TABLE 5.1: The maximum average sizes of the representations.

5.3 Successor set construction

5.3.1 Experimental setting

Our goal was to detect the average complexity of constructing the successor sets in different algorithms and representations. The algorithms and representations were:

1. EBERT [36] and unordered list representation.
2. EBERT and interval representation.
3. SCHMITZ [105] and unordered list representation.
4. SCHMITZ and interval representation.
5. CR_TC, presented in section 3.3, and unordered list representation.
6. CR_TC and interval representation.
7. STACK_TC, presented in section 3.4, and unordered list representation.
8. STACK_TC and interval representation.

We chose EBERT of those algorithms that compute the successor sets during the strong component detection, since it does at most as many unions as EKS [40], and it is easier to implement than GDFTC [64], which does the same number of unions as EBERT. EBERT constructs the successor sets from vertices instead of components. Hence, we had to modify the interval representation to use vertex exit numbers instead of component numbers in EBERT. Since at most one successor set is under construction at any moment during the execution of STACK_TC and SCHMITZ, we used a bit vector in these algorithms for duplicate elimination. Since many successor sets are simultaneously under construction in CR_TC and EBERT, we did not use bit vectors in CR_TC and in EBERT. For the same reason, the interval representation stored the intervals as binary trees in EBERT and in CR_TC, but as arrays in SCHMITZ and in STACK_TC.

To find out the savings gained by topological sorting in STACK_TC, we also used a version of STACK_TC, called STACK_TC_N, which does not sort the components in the component stack.

The performance metric was the total number of basic elements accessed in successor set operations. A basic element was a node of a list in the list representation and an interval in the interval representation. We call the average of the total number of accessed basic elements the *access count*. In addition to the access count, we counted separately the numbers of basic elements accessed in unions, insertions, membership tests, and set initializations.

The workload consisted of randomly generated sparse graphs drawn from $G(n,p)$ and $G(n,p,l=10)$. The workload factors were the number of vertices n and the expected outdegree d. After some initial simulation runs, we selected the following levels for the factors: factor n had five different levels 1000, 3000, 10000, 25000, and 50000, and factor d had levels ranging from 0.0 to 10.0 by 0.1. Since the list representation required too much memory, we could not use levels of n above $n = 10000$. With EBERT, we could not even use level $n = 10000$.

The simulation control parameters were $\alpha = 0.05$ and $\epsilon_{\max} = 0.05$ or $\delta_{\max} = n/100$, i.e., we computed a 95% confidence interval I such that either the relative error of I is at most 5% or the absolute error of I is at most $n/100$.

5.3.2 Results

We present the results using three dimensional plots with the number of vertices n and the expected outdegree d on the x- and y-axis. The access count divided by the number of vertices is presented on the z-axis. Note that the scale of the x-axis and the scale of the z-axis are different in different plots. We also describe the dependency of the maximum access count on the number of vertices n. Note that the maximum access count is the *greatest average value* at a fixed level of n, not the greatest value detected in all simulation runs.

Ebert's algorithm and the list representation

In both models, EBERT behaved almost as in the worst case. The access count was quadratic to the number of vertices and linear to the expected outdegree. In $G(n,p)$, the access count was about $0.42n^2d$ (Figure 5.12). In $G(n,p,l=10)$ the access count was about $0.17n^2d$. Most of these accesses occurred in insertions. Since the memory requirements were quadratic to the number of vertices, we could not use vertex counts greater than $n = 3000$.

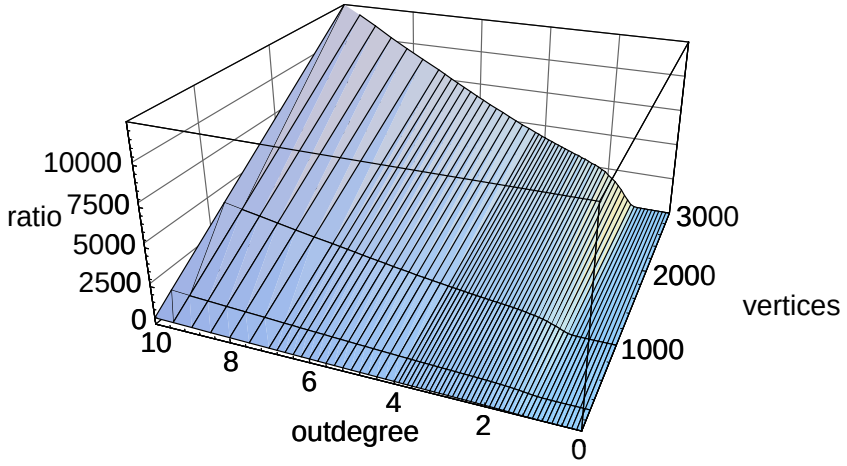


FIGURE 5.12: The access count divided by the number of vertices in EBERT with the list representation: model $G(n,p)$.

Ebert's algorithm and the interval representation

We encountered memory space problems here also and could not use vertex counts above $n = 3000$. The memory requirements and the access counts were here too quadratic to the number of vertices. In model $G(n,p)$, the access count increased between outdegrees $d = 0.0$ and $d \approx 2.0$ and then decreased (Figure 5.13). The maximum access count, reached at $d \approx 2.0$, was about $0.25n^2$. If we had used greater values of d, the access count would have increased again, since the access count in insertions seemed to grow linearly to the expected outdegree. In model $G(n,p,l = 10)$, the access count increased rapidly from $d = 0.0$ to $d \approx 5.0$ and then increased slowly between $d \approx 5.0$ and $d = 10.0$ (Figure 5.14). At $d = 10.0$, the access count was about $0.23n^2$. In both models, most accesses occurred in unions.

Algorithm CR_TC and the list representation

In both models, the access count first increased as the expected outdegree grew and then decreased. In $G(n,p)$, the maximum access count was about $0.14n^2$ and it was reached at $d \approx 1.5$ (Figure 5.15). In $G(n,p,l = 10)$, the maximum access count was reached at $d \approx 3.6$

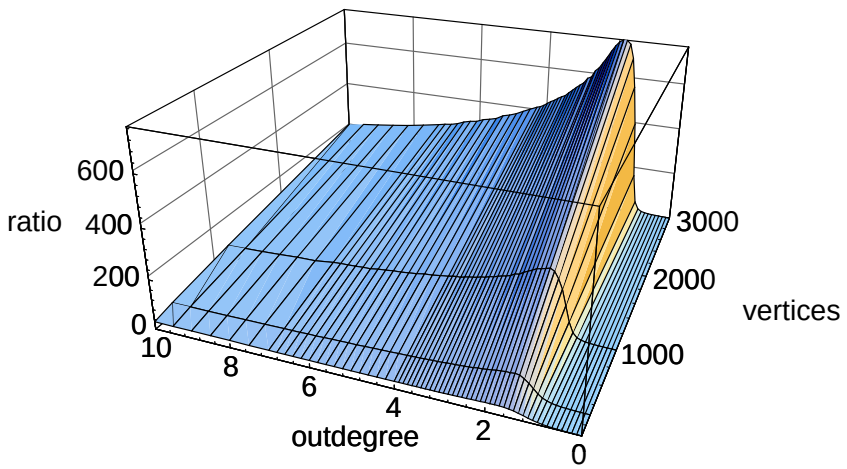


FIGURE 5.13: The access count divided by the number of vertices in EBERT with the interval representation: model $G(n,p)$.

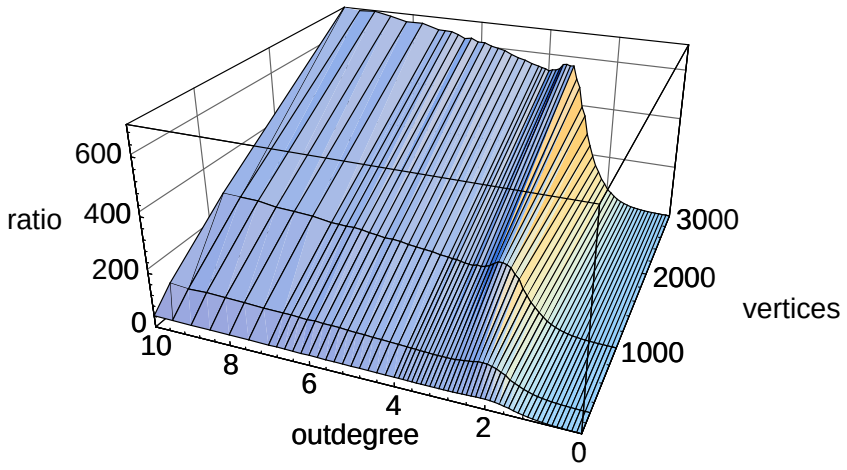


FIGURE 5.14: The access count divided by the number of vertices in EBERT with the interval representation: model $G(n, p, l = 10)$.

(Figure 5.16). The maximum access count seemed to be neither linear nor quadratic to the number of vertices. In both models, the number of accesses in unions and insertions were roughly equal.

Algorithm CR_TC and the interval representation

In model $G(n, p)$, CR_TC behaved with the interval representation as with the list representation except that the maximum access count was much smaller, about $1.4 \times 10^{-3} n^2$. Most accesses occurred in unions. In model $G(n, p, l = 10)$, the maximum access count was $9.6 \times 10^{-5} n$, i.e., linear to the number of vertices (Figure 5.17). The number of accesses in unions and insertions were roughly equal.

Schmitz's algorithm and the list representation

In model $G(n, p)$, the access count increased as the expected outdegree grew from $d = 0.0$ to $d \approx 1.5$ and then decreased (Figure 5.18). Although it cannot be seen in Figure 5.18, after $d \approx 4.0$ the access count started to grow again. This growth was linear to the expected

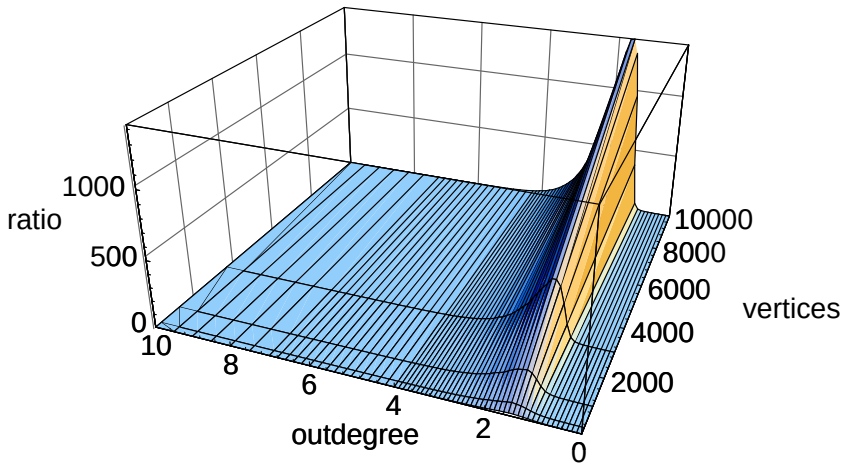


FIGURE 5.15: The access count divided by the number of vertices in CR_TC with the list representation: model $G(n, p)$.

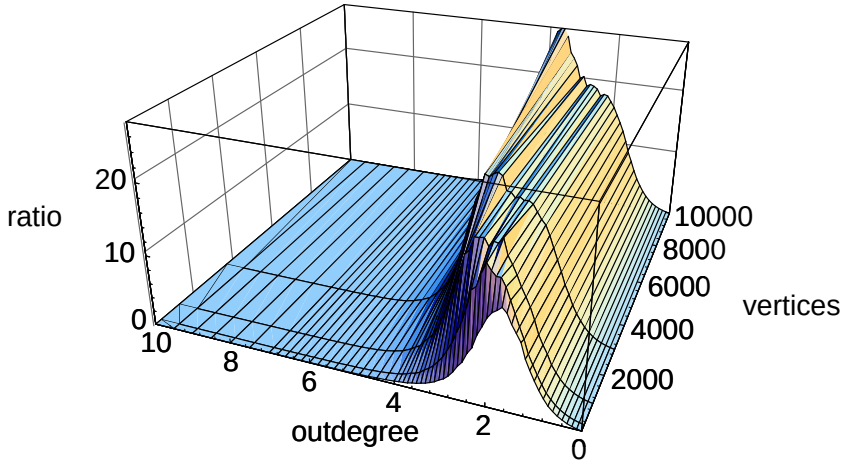


FIGURE 5.16: The access count divided by the number of vertices in CR_TC with the list representation: model $G(n, p, l = 10)$.

outdegree and was caused by insertions. The local maximum access count at $d \approx 1.5$ was about $0.082n^2$ and it was caused by unions. In model $G(n, p, l = 10)$, a similar local maximum appeared between $d = 2.0$ and $d = 3.8$; it was neither linear nor quadratic to the number of vertices (Figure 5.19).

Schmitz's algorithm and the interval representation

In model $G(n, p)$, SCHMITZ behaved with the interval representation roughly as with the list representation except that the local maximum access count at $d \approx 2.0$ was much smaller, about $1.6 \times 10^{-3}n^2$. In model $G(n, p, l = 10)$, the 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 5.20). The (local) maximum at $d \approx 5.0$ was neither linear nor quadratic to the number of vertices.

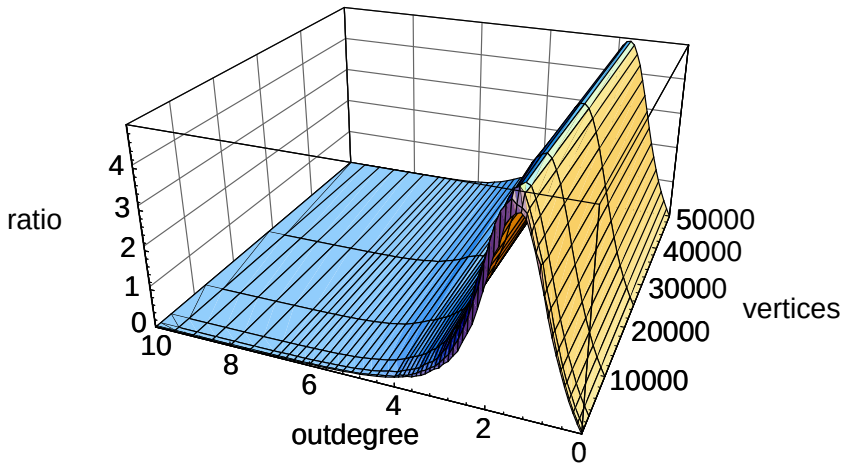


FIGURE 5.17: The access count divided by the number of vertices in CR_TC with the interval representation: model $G(n, p, l = 10)$.

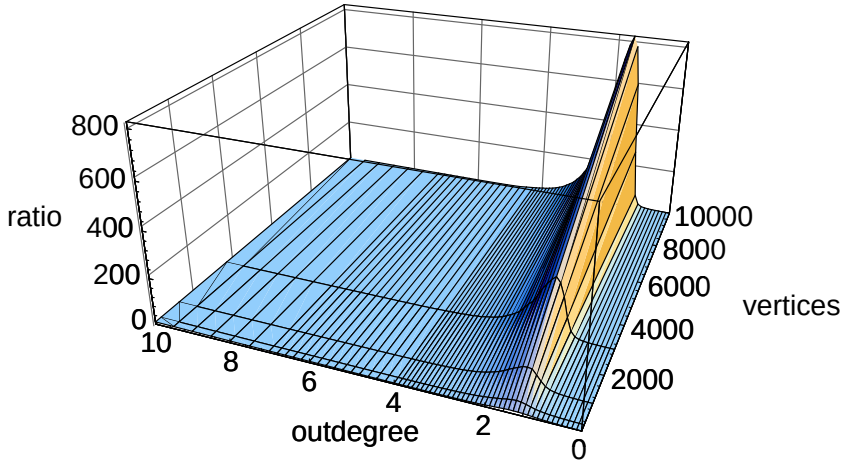


FIGURE 5.18: The access count divided by the number of vertices in SCHMITZ with the list representation: model $G(n,p)$.

Algorithm STACK_TC and the list representation

In model $G(n,p)$, STACK_TC behaved like CR_TC with the list representation. However, the maximum access count, reached at $d \approx 1.5$, was smaller, about $0.076n^2$. In model $G(n,p,l = 10)$, STACK_TC behaved roughly like CR_TC except that the maximum access count, reached between $d \approx 2.0$ and $d \approx 3.8$, was smaller and did not grow as fast as in CR_TC (Figure 5.21). In both models, most accesses occurred in unions. STACK_TC_N, the version of STACK_TC that does not sort the component stack, required only slightly more accesses than STACK_TC in both models.

Algorithm STACK_TC and the interval representation

In model $G(n,p)$, STACK_TC behaved with the interval representation as with the list representation except that the maximum access count was much smaller, about $6.8 \times 10^{-4}n^2$. The maximum access count of STACK_TC_N was about $1.4 \times 10^{-3}n^2$. In model $G(n,p,l = 10)$, the maximum access count of both algorithms was about $9.0 \times 10^{-5}n$, i.e., linear to the number of vertices (Figure 5.22). In both models, the number of accesses in unions and insertions were roughly equal.

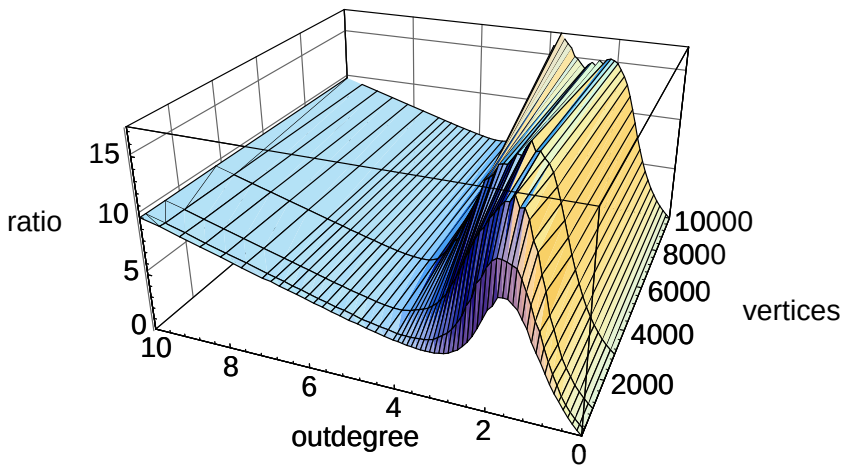


FIGURE 5.19: The access count divided by the number of vertices in SCHMITZ with the list representation: model $G(n,p,l = 10)$.

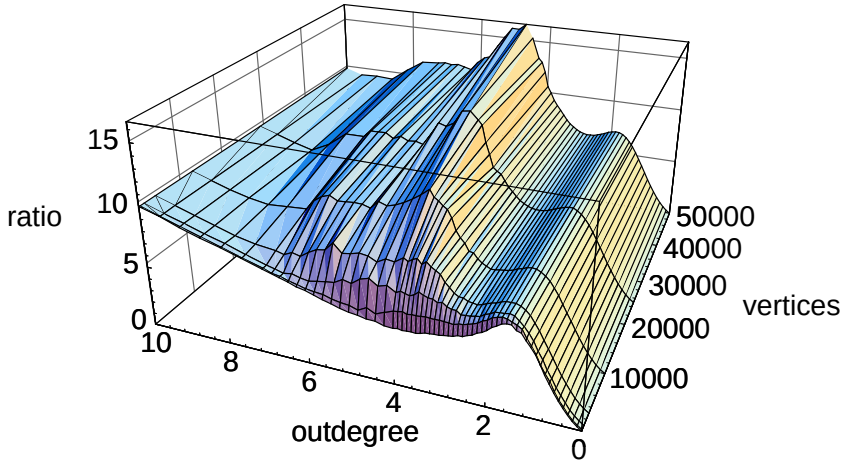


FIGURE 5.20: The access count divided by the number of vertices in SCHMITZ with the interval representation: model $G(n, p, l = 10)$.

Table 5.2 lists the maximum access counts at levels $n = 3000$, $n = 10000$, and $n = 25000$ when the expected outdegree was between 0.0 and 10.0. In addition, it shows the estimated growth of the maximum access count as a function of n.

5.3.3 Discussion

STACK_TC was the clear winner. In both models and with both representations, STACK_TC had the smallest maximum access count. STACK_TC_N usually did more accesses than STACK_TC, but the difference was significant only in model $G(n, p)$ with the interval representation. SCHMITZ was not much worse than STACK_TC and STACK_TC_N. Note, however, that when the expected outdegree grew, the access counts of STACK_TC and STACK_TC_N tended to zero, whereas the access count of SCHMITZ grew linearly. Note also that the maximum access counts grew linearly to the number of vertices in CR_TC, STACK_TC, and STACK_TC_N, but faster than linearly in SCHMITZ. Thus, the difference between SCHMITZ and our algorithms should become greater with greater inputs. EBERT was very inefficient compared to the other algorithms in both models and with both representations.

Other experiments that we have done also indicate that the maximum access counts of our

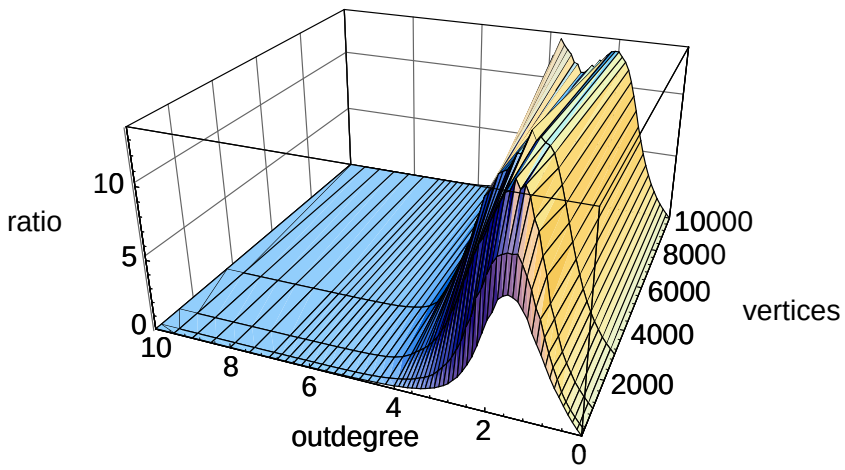


FIGURE 5.21: The access count divided by the number of vertices in STACK_TC with the list representation: model $G(n, p, l = 10)$.

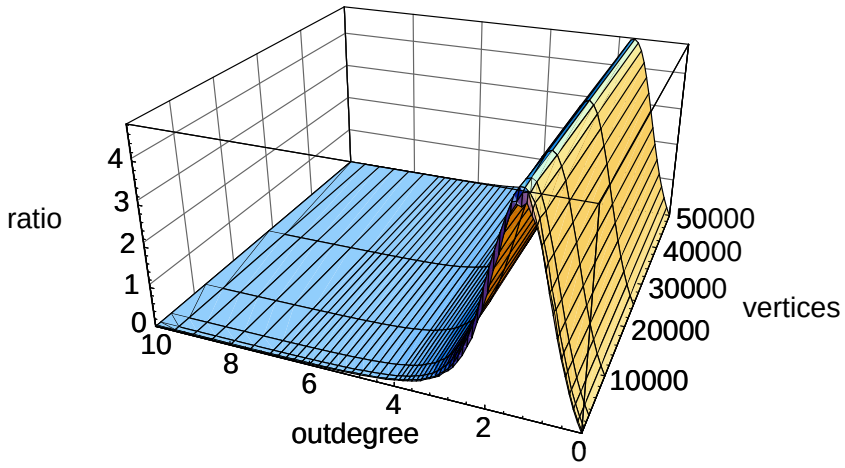


FIGURE 5.22: The access count divided by the number of vertices in STACK_TC with the interval representation: model $G(n, p, l = 10)$.

algorithms are linear to the number of vertices in all models $G(n, p, l)$ when l is small. Since the other operations in the algorithms take $O(n + e)$ time, these results indicate that in model $G(n, p, l)$ with a small value of l, our new algorithms should require a time linear to the size of the input graph on the average. In model $G(n, p)$, the algorithms should also require a linear time except when the expected outdegree is slightly greater than one. Then, the average execution time should be quadratic to the number of vertices.

Algorithm	Repr.	Loc.	$n = 3000$	$n = 10000$	$n = 25000$	Growth
EBERT	list	none	3.8×10^7	–	–	$0.40n^2$
	list	$l = 10$	1.5×10^7	–	–	$1.7n^2$
	interval	none	2.3×10^6	–	–	$0.25n^2$
	interval	$l = 10$	2.1×10^6	–	–	$0.23n^2$
CR_TC	list	none	1.3×10^6	1.4×10^7	–	$0.14n^2$
	list	$l = 10$	64,000	280,000	–	$3.0n \log n$
	interval	none	27,000	150,000	800,000	$1.2 \times 10^{-3}n^2$
	interval	$l = 10$	15,000	49,000	120,000	$4.9n$
SCHMITZ	list	none	770,000	8.3×10^6	–	$0.083n^2$
	list	$l = 10$	50,000	170,000	–	$1.9n \log n$
	interval	none	42,000	250,000	1.2×10^6	$1.7 \times 10^{-3}n^2$
	interval	$l = 10$	30,000	120,000	350,000	$1.5n \log n$
STACK_TC	list	none	710,000	7.7×10^6	–	$0.077n^2$
	list	$l = 10$	41,000	130,000	–	$1.4n \log n$
	interval	none	25,000	110,000	490,000	$7.0 \times 10^{-4}n^2$
	interval	$l = 10$	14,000	48,000	120,000	$4.8n$
STACK_TC_N	list	none	760,000	8.2×10^6	–	$0.082n^2$
	list	$l = 10$	44,000	140,000	–	$1.5n \log n$
	interval	none	27,000	190,000	1.0×10^6	$1.5 \times 10^{-3}n^2$
	interval	$l = 10$	15,000	51,000	130,000	$5.1n$

TABLE 5.2: The maximum access counts; $0.0 \leq d \leq 10.0$.

5.4 Execution time

5.4.1 Experimental setting

Our goal was to compare the execution times of different transitive closure algorithms. We were mainly interested in the relative efficiency of the algorithms. The algorithms were SCHMITZ, CR_TC, STACK_TC, and STACK_TC_N. Since EBERT was very inefficient compared to the other algorithms in the measurements presented in section 5.3, we decided not to use it.

We used only the interval representation, since our goal was to compare algorithms and not representation. Note that in [92], we presented a similar comparison between SCHMITZ and STACK_TC_N using unordered lists augmented with a bit vector as the representation.

The performance metric was the execution time measured using the timing facilities provided by the computer. The simulations were executed in single-user mode to eliminate measurement error caused by other processes. Note that here the method of evaluation was direct measurement and not simulation.

Although the computer that is used in these simulations is clearly a system parameter that affects the results, we assumed that the relative efficiency of the algorithms does not depend much on the computer. Therefore, we used only a single computer, a Sun Sparcstation-10/20 having 96 megabytes of main memory. The operating system was SunOS 4.1.3. We implemented the algorithms and the data structures in C++. An implementation parameter that affects the efficiency of the algorithms is whether or not recursion is removed. Deep recursion seemed to be inefficient in the Sparcstation architecture and in some situations dominated the execution times in our initial simulation runs. Therefore, we removed recursion from the algorithm implementations.

The workload consisted of randomly generated sparse graphs drawn from $G(n, p)$ and $G(n, p, l)$. The workload factors were the number of vertices n, the expected outdegree d, and the locality l. After some initial simulation runs, we selected the following levels for the factors: factor n had three levels 3000, 10000, 25000, factor d had levels ranging from 0.0 to 10.0 by 0.5, and factor l had levels 5 and 10.

The simulation control parameters were $\alpha = 0.05$ and $\epsilon_{\max} = 0.05$ or $\delta_{\max} = n/100$, i.e., we computed a 95% confidence interval I such that either the relative error of I is at most 5% or the absolute error of I is at most $n/100$.

5.4.2 Results

Since all algorithms behaved rather similarly in models $G(n, p, l = 5)$ and $G(n, p, l = 10)$, we present only the results in $G(n, p, l = 10)$. For each vertex count n, we present the results using two dimensional plots having the expected outdegree in the x-axis and the execution time in the y-axis (Figures 5.23–5.28).

5.4.3 Discussion

STACK_TC_N was the overall winner. It was always faster than SCHMITZ and CR_TC. In model $G(n, p, l = 10)$ with $d < 6$, STACK_TC_N was slightly faster than STACK_TC. In model $G(n, p)$, STACK_TC was slower than STACK_TC when n was small, but slightly faster than STACK_TC_N

when n grew. This is consistent with the results of section 5.3; the maximum access count grew significantly faster in STACK_TC_N than in STACK_TC in model $G(n,p)$. Thus, the topological sorting in STACK_TC seems to become useful in model $G(n,p)$ when the graphs are large.

SCHMITZ behaved like STACK_TC between $d = 0.0$ and $d = 1.5$, but was slower with greater values of d. With greater values of d, SCHMITZ required twice or three times as much time as STACK_TC and STACK_TC_N. The extra time in SCHMITZ is caused by two factors. First, SCHMITZ always scans the input graph twice. Second, as the simulations in section 5.3 showed, SCHMITZ needs more successor set accesses than STACK_TC and STACK_TC_N.

CR_TC was considerably slower than the other algorithms between $d = 0.0$ and $d = 1.5$, but became faster than SCHMITZ when d was greater than three. It seems that the strategy of creating the successor sets during the strong component detection is inferior to the strategy of creating the successor set immediately after the detection of the component.

Interestingly, the execution times grew slightly faster than linearly to the number of vertices when they should have grown linearly. The explanation is that with smaller values of n, a greater portion of data fits into the processor cache. We confirmed this hypothesis by applying Tarjan's algorithm to the same inputs; the execution time of Tarjan's algorithm also grew faster than linearly to n.

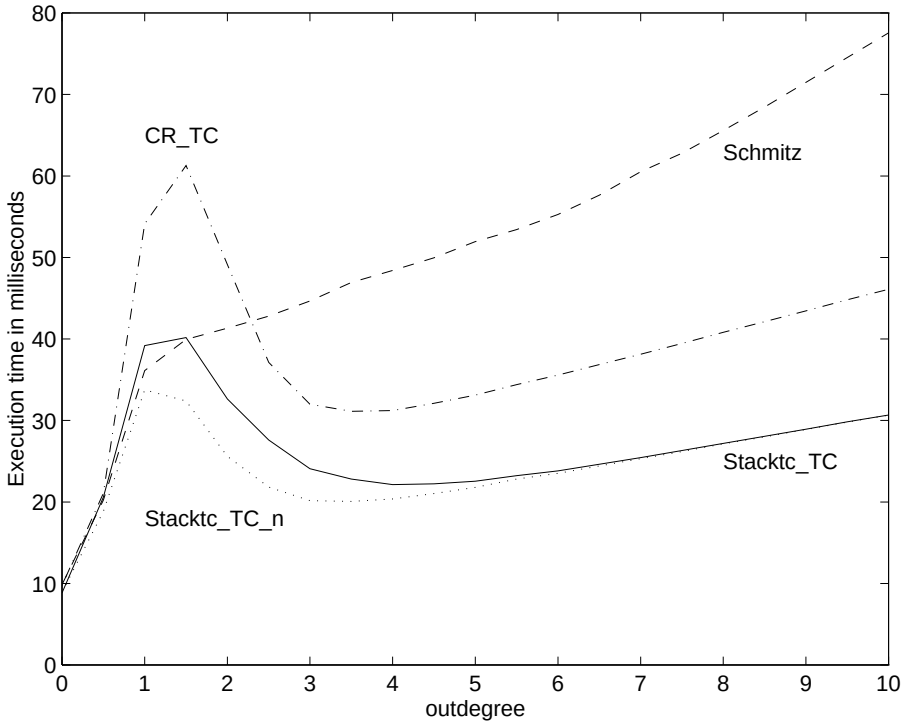


FIGURE 5.23: Execution times in model $G(3000, p)$.

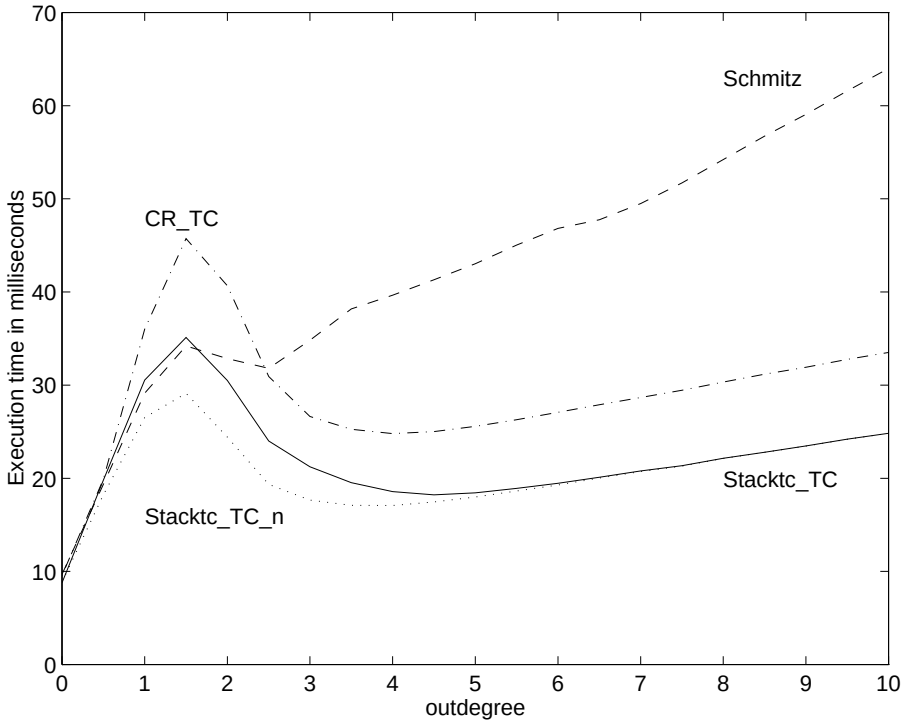


FIGURE 5.24: Execution times in model $G(3000, p, l = 10)$.

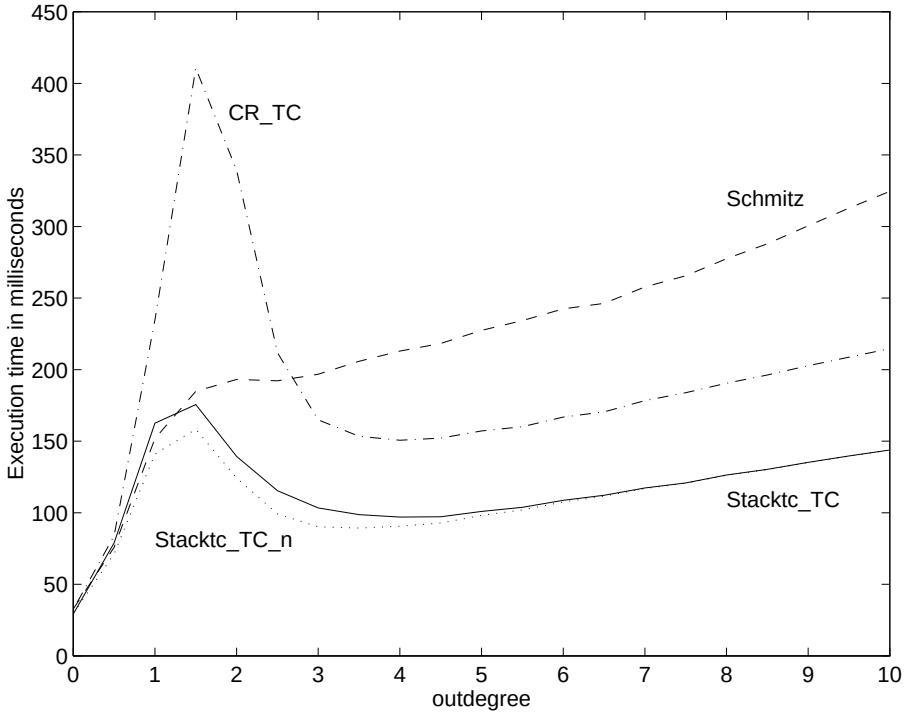


FIGURE 5.25: Execution times in model $G(10000, p)$.

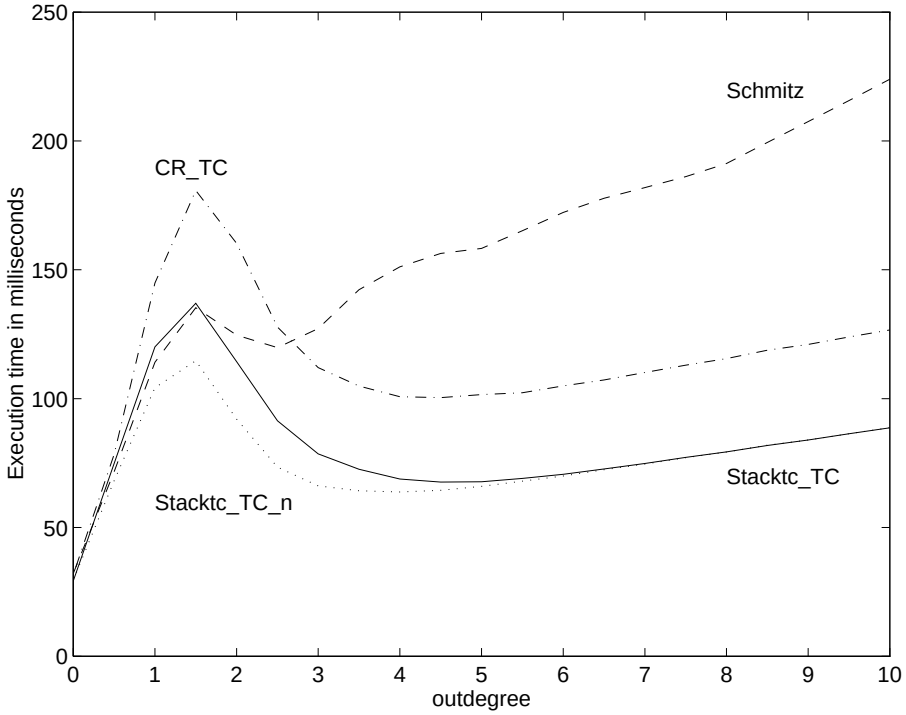


FIGURE 5.26: Execution times in model $G(10000, p, l = 10)$.

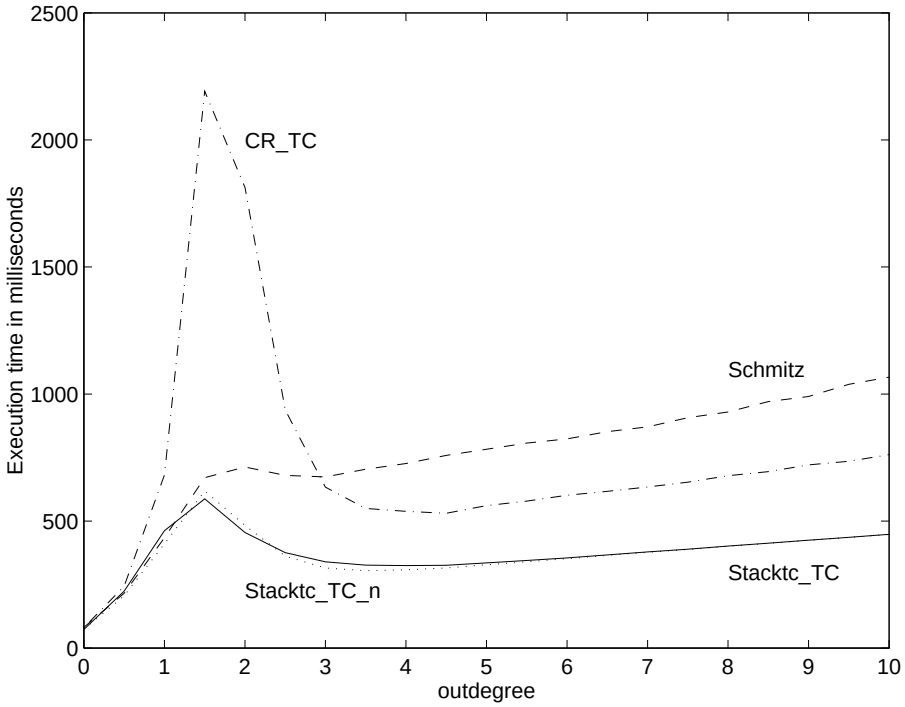


FIGURE 5.27: Execution times in model $G(25000, p)$.

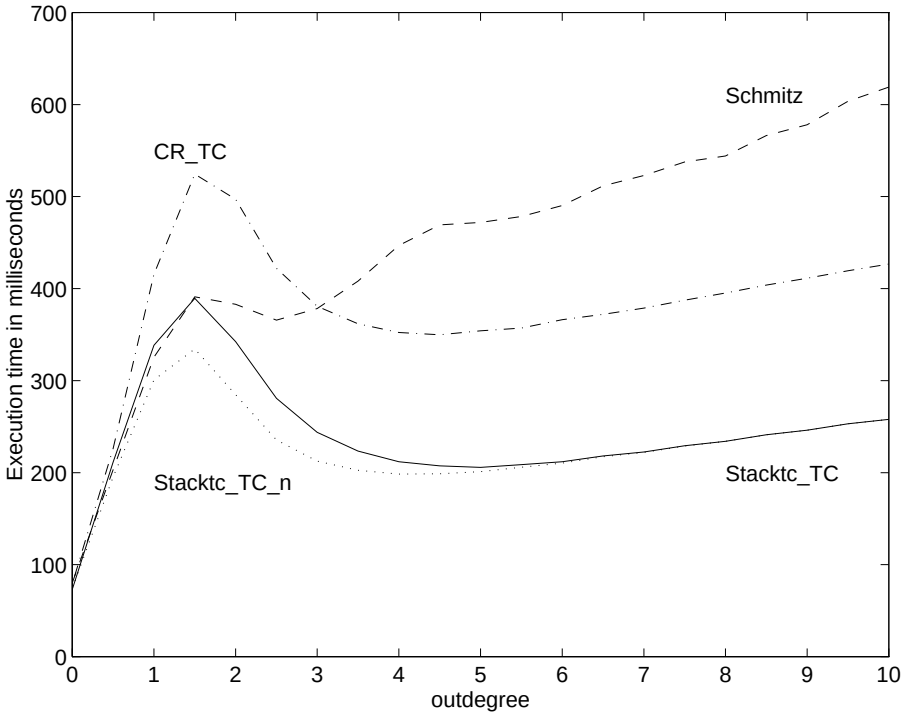


FIGURE 5.28: Execution times in model $G(25000, p, l = 10)$.

Chapter 6

Conclusions

6.1 Summary of the main results

We presented two new transitive closure algorithms that are based on Tarjan's strong component algorithm. The first new algorithm, called CR_TC, constructs the successor sets during the strong component detection. The second new algorithm, called STACK_TC, constructs a successor set of a strong component immediately after detecting the component. Both algorithms construct the successor sets from strong components, not from vertices. An improvement to previous transitive closure algorithms that are based on strong component detection is that the new algorithms scan the input graph only once without generating a partial successor set for each vertex. CR_TC has a smaller worst-case execution time than the previous algorithms that construct the successor sets during strong component detection [36, 40, 64]. STACK_TC has a smaller worst-case execution time than any previous transitive closure algorithm that is based on strong component detection except Munro's algorithm [91], which has a smaller worst-case execution time when the inputs are dense. However, Munro's algorithm is impractical, since it has high constant costs, and its worst-case execution time with sparse inputs is large.

We showed how unnecessary stack operations can be avoided in Tarjan's strong component algorithm.

We presented two new transitive closure representations. The first representation is based on intervals of integers. Each component is represented as its reverse topological number, and a successor set is represented as a collection of intervals of consecutive component numbers. A successor set is represented very compactly compared to traditional set representations such as lists or trees. The representation generalizes the method for compressing the transitive closure of an acyclic graph presented by Agrawal et al. [4]. 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.

We presented another new representation that is based on the chain decomposition of an acyclic input graph by Simon [112]. This new representation also 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 showed how the new representations help avoiding redundancy caused by multiple paths between pairs of vertices in a graph.

We described simulation experiments in which we studied the average-case behavior of

transitive closure algorithms and representations. In the first experiment, we studied the average size of representations. The interval representation required a much smaller memory space than traditional set representations. The average size of the interval representation was linear to the number of vertices in random graph model $G(n, p, l)$. It was also linear in model $G(n, p)$ except when the expected outdegree was slightly greater than one. In this case, the average size was about $0.55n \log n$. The chain representation did not save any space compared to a list representation containing strong components.

In the second experiment, we studied the average complexity of constructing the successor sets with different algorithms and representations. STACK_TC required the least number of accesses of basic elements of the representation both in the list representation and in the interval representation. With the interval representation, the average number of accesses in CR_TC and in STACK_TC was linear to the number of vertices except in random graph model $G(n, p)$ when the expected outdegree was slightly greater than one. In this case, the average number of accesses was quadratic to the number of vertices. Since a basic element access takes a constant time, and since the other operations in CR_TC and in STACK_TC always take a time linear to the size of the graph, this indicates that the new algorithms are usually linear to the size of the input when the interval representation is used.

In the third experiment, we compared the execution times of Schmitz's algorithm [105], algorithm CR_TC, and two versions of algorithm STACK_TC. STACK_TC was the fastest algorithm. It was about two to three times faster than Schmitz's algorithm when the expected outdegree was greater than three. CR_TC was slower than the other algorithms when the expected outdegree was below three, but became faster than Schmitz's algorithm in greater expected outdegrees.

These results satisfy rather well two of our initial goals that we listed in the introduction. Algorithm STACK_TC is more efficient than the previous algorithms. The interval representation is usually linear to the number of vertices, and the execution times of STACK_TC and CR_TC with interval representation are usually linear to the size of the input graph.

6.2 Further research

The third of our goals was that the methods that we develop can be used efficiently both in a main memory environment and in a secondary memory environment. However, in this thesis, we only studied the behavior in a main memory environment. We shall study further the behavior of our new transitive closure algorithms and representations in a secondary memory environment. Hirvisalo [50] adapted algorithm STACK_TC to a disk environment and compared it with algorithm BTC by Ioannidis et al. [64]. The preliminary results indicate that STACK_TC with the interval representation requires less disk I/O than BTC. It seems that since the interval representation is very compact, the main cost in a disk environment is related to the traversal of the input graph and not to the successor set creation. Thus, we must design methods for speeding up graph traversal in a disk environment.

We shall study other problems that are related to the full transitive closure problem, e.g., partial transitive closure problems and generalized transitive closure problems. Some techniques that we used in the transitive closure problem can also be used in these other problems, but new approaches are also required. For instance, the interval representation cannot be used

to store path information compactly.

We also want to develop better methods for algorithm performance study by simulation. Hirvisalo [50] developed a set of tools for analyzing the disk behavior of transitive closure and other related algorithms. These tools will be developed further.

In the simulation experiments, we used two models of random graphs $G(n,p)$ and $G(n,p,l)$ to generate the inputs. We shall try to find other useful models of random graphs, and study the behavior of transitive closure algorithms and representations in these models. Some new models of random graphs could, for instance, mimic the inputs occuring in some specific application areas of transitive closure computation.

Algorithm performance study by simulation has its pros and cons. A weakness is that it requires much time; another weakness is that the results are valid only in those parts of the space of possible inputs that the simulations cover. Thus, we shall also try to do mathematical average case analysis when it seems to be possible.

Bibliography

[1] S.K. Abdali and B.D. Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science*, 40(2,3):257–274, 1985.

[2] L. Adleman, K.S. Booth, F.P. Preparata, and W.L. Ruzzo. Improved time and space bounds for Boolean matrix multiplication. *Acta Informatica*, 11:61–75, 1978.

[3] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. In *Proceedings of the IEEE 3rd International Conference on Data Engineering*, pages 879–885, Los Angeles, CA, February 1987.

[4] 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.

[5] R. Agrawal, S. Dar, and H.V. Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *ACM Transactions on Database Systems*, 15(3):427–458, September 1990.

[6] R. Agrawal and H.V. Jagadish. Direct algorithms for computing the transitive closure of database relations. In *Proceedings of the 13th International VLDB Conference*, 1987.

[7] R. Agrawal and H.V. Jagadish. Materialization and incremental update of path information. In *Proceedings of the 5th International Conference on Data Engineering*, Los Angeles, CA, February 1989.

[8] R. Agrawal and H.V. Jagadish. Hybrid transitive closure algorithms. In *Proceedings of the 16h International VLDB Conference*, pages 326–334, Brisbane, Australia, 1990.

[9] R. Agrawal and J. Kiernan. An access structure for generalized transitive closure queries. In *Proceedings of the IEEE 9th International Conference on Data Engineering*, pages 429–438, Vienna, Austria, April 1993.

[10] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

[11] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Mass., 1983.

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

[13] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[14] V.L. Arlazov, E.A. Dinic, M.A. Kronrod, and I.A. Faradžev. On economical construction of the transitive closure of an oriented graph. *Soviet Math. Dokl.*, 11(5):1209–1210, 1970.

[15] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *ACM-SIGMOD 1986 Conference on Management of Data*, pages 16–52, 1986.

[16] François Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database System. The Story of O_2*. Morgan Kaufmann Publishers, Inc., 1992.

[17] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3):255–299, April 1991.

[18] 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.

[19] B. Bollóbas. *Random Graphs*. Academic Press, New York, 1985.

[20] 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.

[21] A.L. Buchsbaum, P.C. Kannelakis, and J.S. Vitter. A data structure for arc insertion and regular path finding. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 22–31, 1990.

[22] 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.

[23] R.G.G. Cattell. *Object Data Management*. Addison-Wesley, 1991.

[24] 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.

[25] J-P. Cheiney and C. de Maindreville. A parallel transitive closure algorithm using hash-based clustering. In *Database Machines, Proceedings of the Sixth International Workshop, IWDM'89*, volume 368 of *Lecture Notes in Computer Science*, pages 301–316, Deauville, France, June 1989. Springer-Verlag.

[26] J-P. Cheiney and C. de Maindreville. A parallel strategy for transitive closure using double hash-based clustering. In *Proceedings of the 16h International VLDB Conference*, pages 347–358, Brisbane, Australia, 1990.

[27] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[28] E.F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[29] 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.

[30] I.F. Cruz and T.S. Norvell. Aggregative closure: An extension of transitive closure. In *Proceedings of the IEEE 5th International Conference on Data Engineering*, pages 84–391, Los Angeles, California, February 1989.

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

[32] S. Dar and R. Agrawal. Extending SQL with generalized transitive closure. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):799–812, 1993.

[33] S. Dar and H.V. Jagadish. A spanning tree transitive closure algorithm. In *Proceedings of the IEEE 8th International Conference on Data Engineering*, pages 2–11, Tempe, Arizona, February 1992.

[34] S. Dar and R. Ramakrishnan. A performance study of transitive closure algorithms. In *Proceedings of the ACM-SIGMOD 1994 Conference on Management of Data*, 1994.

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

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

[37] J. Eder. Extending SQL with general transitive closure and extreme value selection. *IEEE Transactions on Knowledge and Data Engineering*, 2(4):381–390, December 1990.

[38] J. Eloranta. Equivalence concepts and algorithms for transition systems and ccs-like languages. Lic.Ph. thesis, Helsinki University, Department of Computer Science, 1990.

[39] R.W. Engles. Structured tables. ANSI X3H2-87-331, December 1987.

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

[41] 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.

[42] R.W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[43] M.E. Furman. Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Soviet Math. Dokl.*, 11(5):1252, 1970.

[44] 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.

[45] 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.

[46] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1988.

[47] K-H. Guh, C. Sun, and C. Yu. Real time retrieval and update of materialized transitive closure. In *Proceedings of the IEEE 7th International Conference on Data Engineering*, pages 690–697, Kobe, Japan, April 1991.

[48] K-H. Guh and C. Yu. Efficient management of materialized generalized transitive closure in centralized and parallel environments. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):371–381, August 1992.

[49] K.C. Guh and C.T. Yu. Evaluation of transitive closure in distributed database systems. *IEEE Journal on Selected Areas in Communications*, 7(3):399–407, April 1989.

[50] V. Hirvisalo. Transitiivisen sulkeuman laskeminen tietokantaympäristössä (transitive closure computation in database environment). Master's thesis, Helsinki University of Technology, Laboratory of Information Processing Science, 1994. In Finnish.

[51] M.A.W. Houtsma, P. Apers, and S. Ceri. Complex transitive closure queries on a fragmented graph. In *Proceedings of the 3rd International Conference on Database Theory*, Lecture Notes in Computer Science, pages 470–484, Berlin, 1990. Springer-Verlag.

[52] M.A.W. Houtsma, P. Apers, and S. Ceri. Distributed transitive closure computations: The disconnection set approach. In *Proceedings of the 16th VLDB Conference*, pages 335–346, Brisbane, Australia, 1990.

[53] 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.

[54] M.A.W. Houtsma, F. Cacace, and S. Ceri. Parallel hierarchical evaluation of transitive closure queries. In *Proceedings of the 1st International Conference on Parallel and Distributed Systems*, pages 130–137, Miami Beach, Florida, December 1991.

[55] K. Hua and S. Hannenhalli. Parallel transitive closure computations using topological sort. In *Proceedings of the 1st International Conference on Parallel and Distributed Systems*, pages 122–129, Miami Beach, Florida, December 1991.

[56] K.A. Hua, J.X.W. Su, and C.M. Hua. Efficient evaluation of traversal recursive queries. In *Proceedings of the IEEE 9th International Conference on Data Engineering*, pages 549–558, Vienna, Austria, April 1993.

[57] 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.

[58] T. Ibaraki and N. Katoh. On-line computation of transitive closures of graphs. *Information Processing Letters*, 16:95–97, 1983.

[59] IBM. *SIMPL/1 Program Reference Manual.* IBM, New York, 1972.

[60] IMSL. *IMSL Stat/library User's Manual.* International Mathematical and Statistical Libraries, Houston, Texas, 1987.

[61] Y.E. Ioannidis. On the computation of the transitive closure of relational operators. In *Proceedings of the 12th International VLDB Conference*, pages 403–411, 1986.

[62] Y.E. Ioannidis and R. Ramakrishnan. Efficient transitive closure algorithms. In *Proceedings of the 14th VLDB Conference*, pages 335–346, Los Angeles, California, 1988.

[63] Y.E. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. Technical report, Computer Science Department, University of Wisconsin, Madison, WI, 1991.

[64] 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.

[65] G. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

[66] G. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28(1):5–11, 1988.

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

[68] H.V. Jagadish, R. Agrawal, and L. Ness. A study of transitive closure as a recursion mechanism. In *Proceedings of the ACM-SIGMOD 1987 Conference on Management of Data*, pages 331–344, San Francisco, California, 1987.

[69] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation and modeling.* John Wiley & Sons, Inc., New York, 1991.

[70] H. Jakobson. Mixed-approach algorithms for transitive closure. In *Proceedings of the 10th ACM Symposium on Principles of Database Systems*, pages 199–205, 1991.

[71] H. Jakobson. On tree-based techniques for query evaluation. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, pages 380–392, San Diego, California, 1992.

[72] B. Jaumard and M. Minoux. An efficient algorithm for the transitive closure and a linear worst-case complexity result for a class of sparse graphs. *Information Processing Letters*, 22:163–169, 1986.

[73] B. Jiang. A suitable algorithm for computing partial transitive closures in databases. In *Proceedings of the IEEE 6th International Conference on Data Engineering*, pages 264–271, Los Angeles, California, February 1990.

[74] B. Jiang. Traversing graphs in paging environment, BFS or DFS? *Information Processing Letters*, 37(3):143–147, 1991.

[75] B. Jiang. I/O- and CPU-optimal recognition of strongly connected components. *Information Processing Letters*, 45(3):111–115, March 1993.

[76] M. Jun and T. Takaoka. An $O(n^2)$ parallel algorithm to compute the all pairs shortest paths and transitive closure. *Journal of Information Processing*, 12(2):119–124, 1989.

[77] R. Kabler, Y.E. Ioannidis, and M. Carey. Performance evaluation of algorithms for transitive closure. *Information Systems*, 17(5):415–441, September 1992.

[78] S. Kapidakis. *Average-Case Analysis of Graph-Searching*. PhD thesis, Princeton University, Department of Computer Science, October 1990.

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

[80] S.C. Kleene. Representation of events in nerve nets and finite automate. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, N.J., 1956.

[81] J.A. La Poutré and J. van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In H. Gottler and H.J. Schneider, editors, *Graph-Theoretic Concepts in Computer Science*, volume 314 of *Lecture Notes in Computer Science*, pages 106–120, Berlin, 1988. Springer-Verlag.

[82] P.-A. Larson and V. Deshpande. A file structure supporting traversal recursion. In *Proceedings of the ACM-SIGMOD 1989 Conference on Management of Data*, pages 243–252, 1989.

[83] S.S. Lavenberg and C.H. Sauer. Sequential stopping rules for the regenerative method of simulation. *IBM Journal of Research and Development*, 21:545–558, 1977.

[84] A.M. Law and W.D. Kelton. Confidence intervals for steady-state simulations, II: A survey of sequential procedures. *Management Science*, 28(5):550–562, May 1982.

[85] D.J. Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76, 1977.

[86] H. Lu. New strategies for computing the transitive closure of a database relation. In *Proceedings of the 13th International VLDB Conference*, pages 267–274, 1987.

[87] H. Lu, K. Mikkilineni, and J. Richardson. Design and evaluation of algorithms to compute the transitive closure of a database relation. In *Proceedings of the 3rd International Conference on Data Engineering*, pages 112–119, Los Angeles, California, February 1987.

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

[89] J.A. McHugh. *Algorithmic Graph Theory.* Prentice-Hall, Englewood Cliffs, N.J., 1990.

[90] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1984.

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

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

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

[94] E. Nuutila and E. Soisalon-Soininen. On finding the strong components in a directed graph. Technical Report TKO-B94, Helsinki University of Technology, Laboratory of Information Processing Science, 1993.

[95] E. Nuutila and E. Soisalon-Soininen. A single pass algorithm for transitive closure. Technical Report TKO-B95, Helsinki University of Technology, Laboratory of Information Processing Science, 1993.

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

[97] P. O'Neill and E.J. O'Neill. A fast expected time algorithm for Boolean matrix multiplication and transitive closure. *Information and Control*, 22:132–138, 1973.

[98] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

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

[100] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, Mass., 1988.

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

[102] G.Z. Qadah, L.J. Henschen, and J.J. Kim. Efficient algorithms for the instantiated transitive closure queries. *IEEE Transactions on Software Engineering*, 17(3):296–309, March 1991.

[103] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal recursion: a practical approach to supporting recursive applications. In *Proceedings of the ACM-SIGMOD 1986 International Conference on Management of Data*, pages 166–176, Washington D.C, May 1986.

[104] B. Roy. Transitivité et connexité. *C.R. Acad. Sci. Paris*, 249:216–218, 1959.

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

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

[107] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1988.

[108] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 7:67–72, 1981.

[109] P. Shaw. CLOSURE expressions. ANSI X3H2-87-330, December 1987.

[110] P. Shaw. A generalization of recursive expressions for non-linear recursion of fixed degree. ANSI X3H2-88-93REV, April 1988.

[111] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. In Laurent Kott, editor, *Automata, Languages and Programming, 13th International Colloquium*, volume 26 of *Lecture Notes in Computer Science*, pages 376–386, Rennes, France, July 1986. Springer-Verlag.

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

[113] S. Sippu and E. Soisalon-Soininen. A generalized transitive closure for relational queries. In *Proceedings of the 7th ACM Symposium on Principles of Database Systems*, pages 325–332, 1988.

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

[115] J. Sullivan. Tree structured traversal. ANSI X3H2-87-306, November 1987.

[116] R. Tamassia and J.S. Vitter. Optimal parallel algorithms for transitive closure and point locations in planar structures. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 399–408, Santa Fe, New Mexico, June 1989.

[117] A.S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

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

[119] R.E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, 1981.

[120] R.E. Tarjan. Amortized computational complexity. *SIAM Journal of Algebraic Discrete Methods*, 6:306–318, 1985.

[121] I.H. Toroslu and G.Z. Qadah. The efficient computation of strong partial transitive closures. In *Proceedings of the IEEE 9th International Conference on Data Engineering*, pages 530–537, Vienna, Austria, April 1993.

[122] J. Ullman and M. Yannakakis. The input/output complexity of transitive closure. In *Proceedings of the ACM-SIGMOD 1990 International Conference on Management of Data*, pages 44–53, 1990.

[123] J. D. Ullman. *Principles of Database and Knowledge-base Systems, Vol II: The New Technologies*. Computer Science Press, Rockville, MD., 1989.

[124] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986.

[125] P. Valduriez and S. Khoshafian. Parallel evaluation of the transitive closure of a database relation. *International Journal of Parallel Programming*, 17(1):19–42, 1988.

[126] J. Van Leeuwen. On efficiently computing the product of two binary relations. *International Journal of Computer Mathematics*, 5:193–201, 1976.

[127] 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.

[128] H.S. Warren. A modification of Warshall's algorithm for the transitive closure of binary relations. *Communications of the ACM*, 18(4):218–220, 1975.

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

[130] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 230–242, 1990.

[131] D.M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30:369–384, 1993.

[132] Stanley B. Zdonik and D. Maier. Fundamentals of object-oriented databases. In *Readings in Object-Oriented Databases*, pages 1–32. Morgan Kaufmann, San Mateo, California, 1990.

[133] Stanley B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Databases*. Morgan Kaufmann, San Mateo, California, 1990.

[134] M.M. Zloof. Query-by-example: Operations on the transitive closure. Technical Report RC 5526, IBM, Yorktown Heights, NY, 1975.