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.

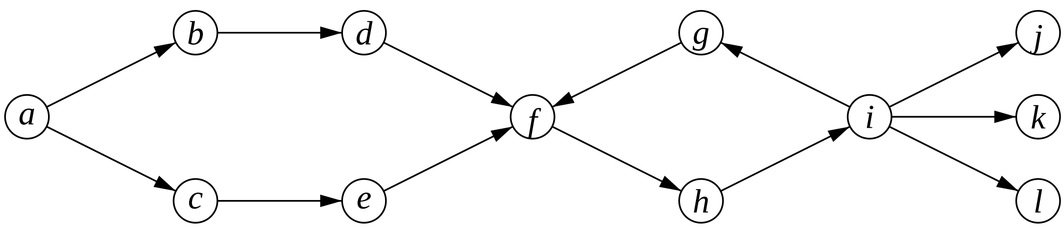

FIGURE 1.1: A directed graph with vertices having the same successor sets and with multiple paths between vertices.

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.