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*.

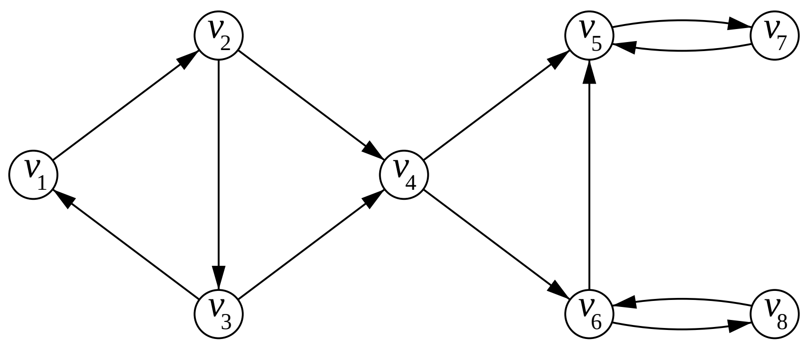
Definition. An *adjacency matrix* of a graph $G = (V, E)$ is an $n \times n$ Boolean matrix A such that $A[i, j]$ is **true** iff (if and only if) G has an edge (v_i, v_j).

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. □



$$\begin{array}{cc} & \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} & \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

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

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^*.

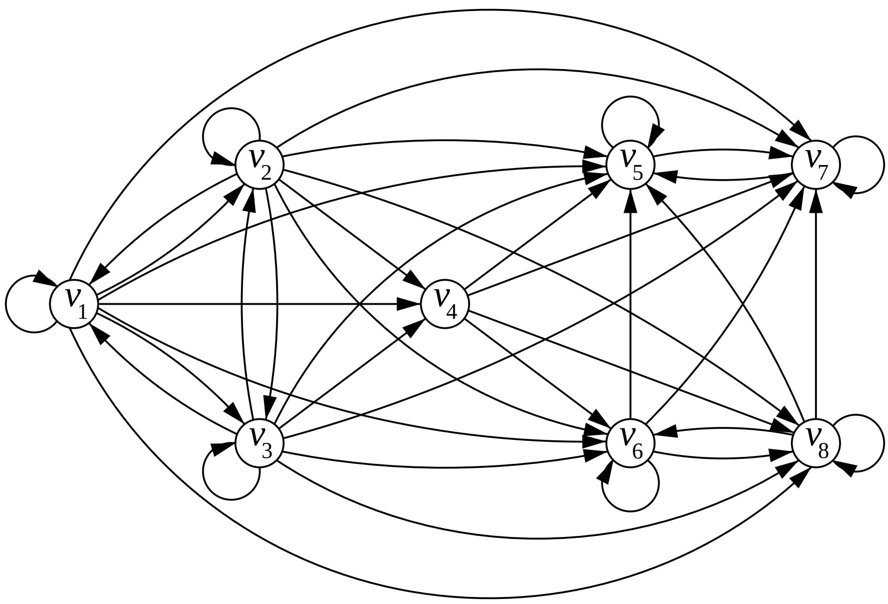


FIGURE 2.2: The transitive closure of graph G of Figure 2.1.

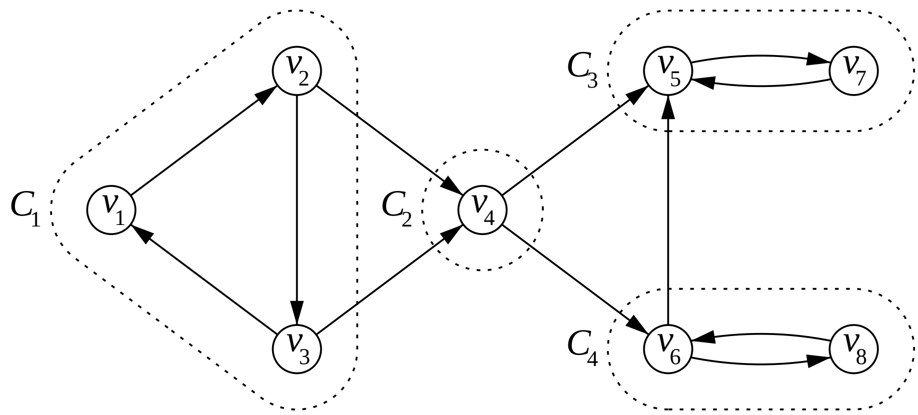
Example 2.4. The transitive closure of graph G, presented in Figure 2.2, can be changed to the reflexive transitive closure of graph G by inserting the edge (v_4, v_4), since $E^* = E^+ \cup I$ where $I = \{(v, v) \mid v \in V\}$, the identity relation. □

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 $Comp(v) = Y$.

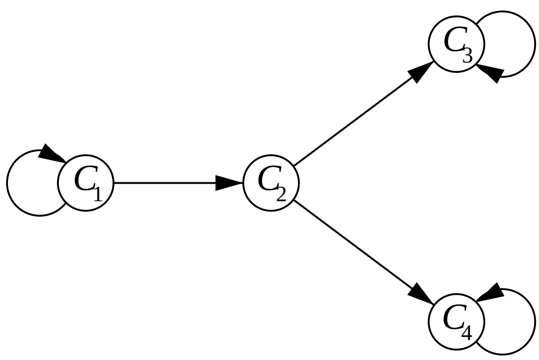
Each vertex of a strong component has the same successor set; this can be used in designing efficient transitive closure algorithms.

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.



(a)



(b)

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 $r \xrightarrow{*} v$.

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, $E^+ = E \cup E^2 \cup E^3 \cup E^4 \cup E^5$. □

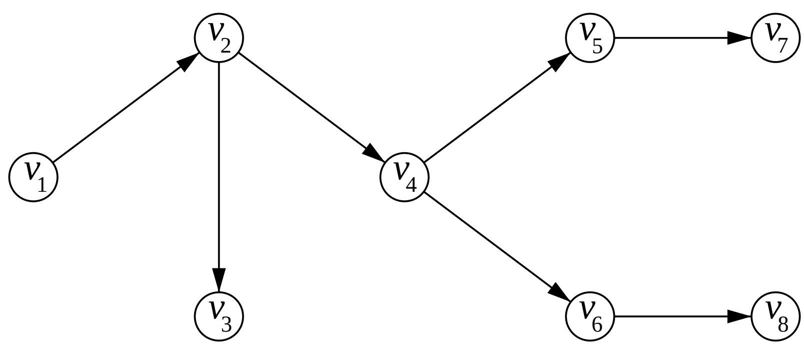


FIGURE 2.4: A spanning tree of graph G of Figure 2.1.

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

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

FIGURE 2.5: Warshall's algorithm

```
for i := 2 to n do
    for k := 1 to i − 1 do
        if A[i,k] then
            for j := 1 to n do A[i,j] := A[i,j] or A[k,j];
for i := 1 to n − 1 do
    for k := i + 1 to n do
        if A[i,k] then
            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 transitive 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].

$R^+ := R;$
$\Delta := R;$
do
    $\Delta := \Delta \circ R - R^+$
    $R^+ := R^+ \cup \Delta$
while $\Delta \neq \emptyset$

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.

$R^+ := R;$
$\Delta := R;$
$\delta := R$
do
    $\delta := \delta \circ \delta;$
    $\Delta := R^+ \circ \delta;$
    $R^+ := R^+ \cup \Delta \cup \delta$
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.

$T := R$ sorted by both its fields;
for $(x, y) \in T$ in sorted order **do**
    if $x > y$ **then**
        $T := T \cup \{(x, y)\} \circ T$;
for $(x, y) \in T$ in sorted order **do**
    if $x < y$ **then**
        $T := T \cup \{(x, y)\} \circ T$

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.

for $k := 1$ **to** $n/\sqrt{s}$ **do begin**
    $A_{k,k} := A^*_{k,k}$;
    for $i := 1$ **to** $n/\sqrt{s}$ **do**
        for $j := 1$ **to** $n/\sqrt{s}$ **do**
            $A_{i,j} := A_{i,j} + A_{i,k} \times A_{k,k} \times A_{k,j}$;

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.