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.
```

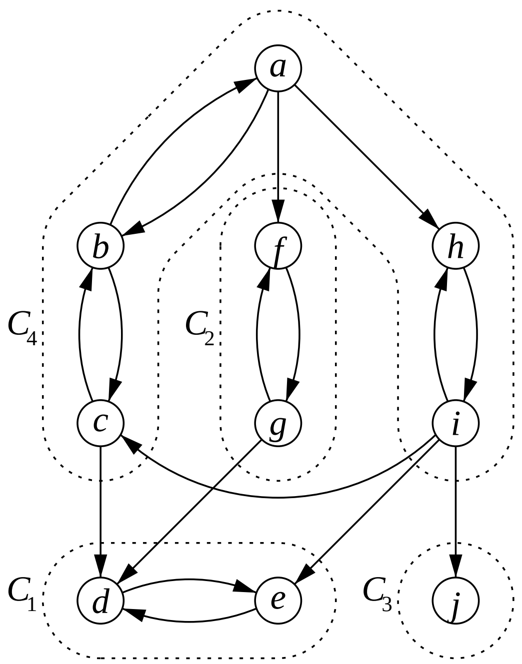
FIGURE 3.1: Tarjan's algorithm detects the strongly connected components of graph $G = (V, E)$.

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
$enter(a)$		$\textsc{pop}(stack)$	(f, c, b, a)
$Root(a) := a$		$Comp(g) := C_2$	(f, c, b, a)
$\textsc{push}(a, stack)$	(a)	$\textsc{pop}(stack)$	(c, b, a)
$enter(b)$	(a)	$Comp(f) := C_2$	(c, b, a)
$Root(b) := b$	(a)	$exit(f)$	(c, b, a)
$\textsc{push}(b, stack)$	(b, a)	$enter(h)$	(c, b, a)
$enter(c)$	(b, a)	$Root(h) := h$	(c, b, a)
$Root(c) := c$	(b, a)	$\textsc{push}(h, stack)$	(h, c, b, a)
$\textsc{push}(c, stack)$	(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)	$\textsc{push}(i, stack)$	(i, h, c, b, a)
$\textsc{push}(d, stack)$	(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)
$\textsc{push}(e, stack)$	(e, d, c, b, a)	$\textsc{push}(j, stack)$	(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)	$\textsc{pop}(stack)$	(i, h, c, b, a)
create C_1	(e, d, c, b, a)	$Comp(j) := C_3$	(i, h, c, b, a)
$\textsc{pop}(stack)$	(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)
$\textsc{pop}(stack)$	(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)	$\textsc{pop}(stack)$	(h, c, b, a)
$Root(b) := a$	(c, b, a)	$Comp(i) := C_4$	(h, c, b, a)
$exit(b)$	(c, b, a)	$\textsc{pop}(stack)$	(c, b, a)
$enter(f)$	(c, b, a)	$Comp(h) := C_4$	(c, b, a)
$Root(f) := f$	(c, b, a)	$\textsc{pop}(stack)$	(b, a)
$\textsc{push}(f, stack)$	(f, c, b, a)	$Comp(c) := C_4$	(b, a)
$enter(g)$	(f, c, b, a)	$\textsc{pop}(stack)$	(a)
$Root(g) := g$	(f, c, b, a)	$Comp(b) := C_4$	(a)
$\textsc{push}(g, stack)$	(g, f, c, b, a)	$\textsc{pop}(stack)$	$()$
$Root(g) := f$	(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 $x \xrightarrow{+} r$ 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

```
procedure VISIT1(v);
begin
    Root(v) := v; Comp(v) := Nil;
    for each vertex w such that (v,w) ∈ E do begin
        if w is not already visited then VISIT1(w);
        if Comp(w) = Nil then Root(v) := MIN(Root(v), Root(w))
    end;
    if Root(v) = v then begin
        create a new component C;
        Comp(v) := C;
        insert v into component C;
        while TOP(stack) > v do begin
            w := POP(stack);
            Comp(w) := C;
            insert w into component C
        end
    end else PUSH(v, stack);
end;
begin /* Main program */
    Initialize stack to contain a value < any vertex in V;
    for each vertex v ∈ V do
        if v is not already visited then VISIT1(v)
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
$\mathit{enter}(a)$	$()$	$\mathit{Comp}(f) := C_2$	(g, b, c)
$\mathit{Root}(a) := a$	$()$	$\text{POP}(\mathit{stack})$	(b, c)
$\mathit{enter}(b)$	$()$	$\mathit{Comp}(g) := C_2$	(b, c)
$\mathit{Root}(b) := b$	$()$	$\mathit{exit}(f)$	(b, c)
$\mathit{enter}(c)$	$()$	$\mathit{enter}(h)$	(b, c)
$\mathit{Root}(c) := c$	$()$	$\mathit{Root}(h) := h$	(b, c)
$\mathit{enter}(d)$	$()$	$\mathit{enter}(i)$	(b, c)
$\mathit{Root}(d) := d$	$()$	$\mathit{Root}(i) := i$	(b, c)
$\mathit{enter}(e)$	$()$	$\mathit{Root}(i) := h$	(b, c)
$\mathit{Root}(e) := e$	$()$	$\mathit{enter}(j)$	(b, c)
$\mathit{Root}(e) := d$	$()$	$\mathit{Root}(j) := j$	(b, c)
$\text{PUSH}(e, \mathit{stack})$	(e)	create C_3	(b, c)
$\mathit{exit}(e)$	(e)	$\mathit{Comp}(j) := C_3$	(b, c)
create C_1	(e)	$\mathit{exit}(j)$	(b, c)
$\mathit{Comp}(d) := C_1$	(e)	$\mathit{Root}(i) := b$	(b, c)
$\text{POP}(\mathit{stack})$	$()$	$\text{PUSH}(i, \mathit{stack})$	(b, c)
$\mathit{Comp}(e) := C_1$	$()$	$\mathit{exit}(i)$	(i, b, c)
$\mathit{exit}(d)$	$()$	$\mathit{Root}(h) := b$	(i, b, c)
$\mathit{Root}(c) := b$	$()$	$\text{PUSH}(h, \mathit{stack})$	(h, i, b, c)
$\text{PUSH}(c, \mathit{stack})$	(c)	$\mathit{exit}(h)$	(h, i, b, c)
$\mathit{exit}(c)$	(c)	create C_4	(h, i, b, c)
$\mathit{Root}(b) := a$	(c)	$\mathit{Comp}(a) := C_4$	(h, i, b, c)
$\text{PUSH}(b, \mathit{stack})$	(b, c)	$\text{POP}(\mathit{stack})$	(i, b, c)
$\mathit{exit}(b)$	(b, c)	$\mathit{Comp}(h) := C_4$	(i, b, c)
$\mathit{enter}(f)$	(b, c)	$\text{POP}(\mathit{stack})$	(b, c)
$\mathit{Root}(f) := f$	(b, c)	$\mathit{Comp}(i) := C_4$	(b, c)
$\mathit{enter}(g)$	(b, c)	$\text{POP}(\mathit{stack})$	(c)
$\mathit{Root}(g) := g$	(b, c)	$\mathit{Comp}(b) := C_4$	(c)
$\mathit{Root}(g) := f$	(b, c)	$\text{POP}(\mathit{stack})$	$()$
$\text{PUSH}(g, \mathit{stack})$	(g, b, c)	$\mathit{Comp}(c) := C_4$	$()$
$\mathit{exit}(g)$	(g, b, c)	$\mathit{exit}(a)$	$()$
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 $\mathit{Root}(w)$ belongs to the same component as v. Tests $\mathit{Comp}(w) = \mathit{Nil}$ and $\mathit{Comp}(\mathit{Root}(w)) = \mathit{Nil}$ always yield the same result. If we replace the test $\mathit{Comp}(w) = \mathit{Nil}$ by the test $\mathit{Comp}(\mathit{Root}(w)) = \mathit{Nil}$, 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 $\mathit{Comp}(w) = C$ 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.

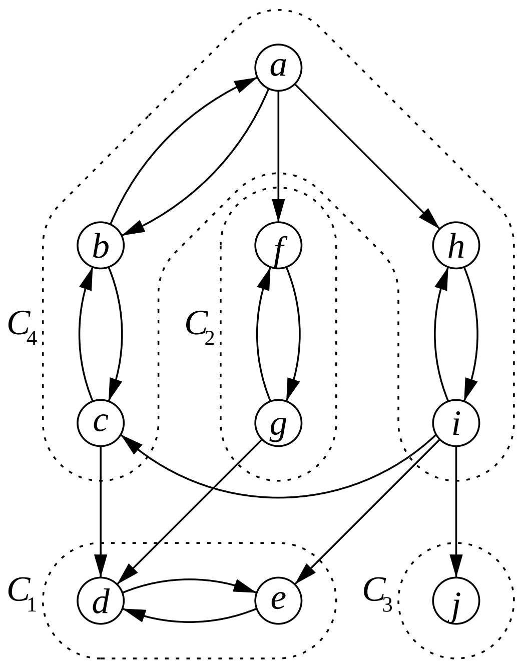
```
(1)   procedure SIMPLE_TC(v);
(2)   begin
(3)       Root(v) := v; Comp(v) := Nil;
(4)       PUSH(v, stack);
(5)       Succ(v) := {w | (v, w) ∈ E};
(6)       for each vertex w such that (v, w) ∈ E do begin
(7)           if w is not already visited then SIMPLE_TC(w);
(8)           if Comp(w) = Nil 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)

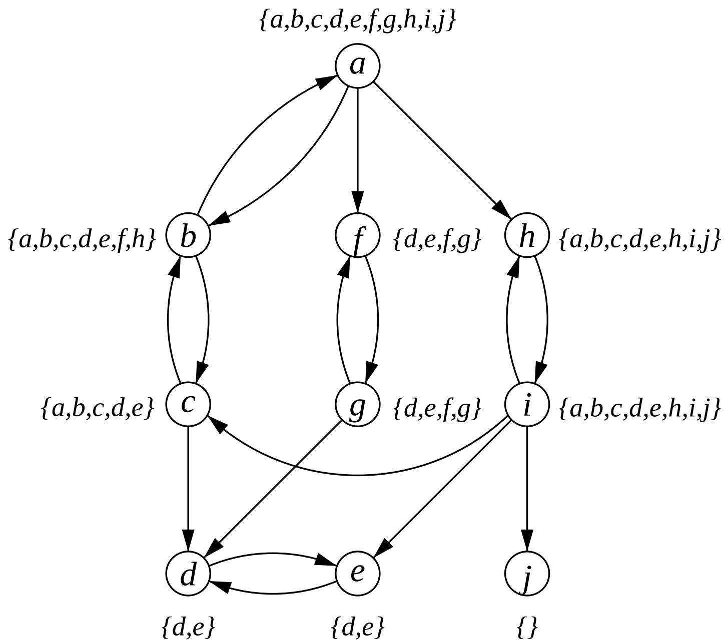


FIGURE 3.10: **(a)** The adjacency lists of graph G of Figure 3.9(a). **(b)** The partial successor sets of the vertices when SIMPLE_TC is applied to G.

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.

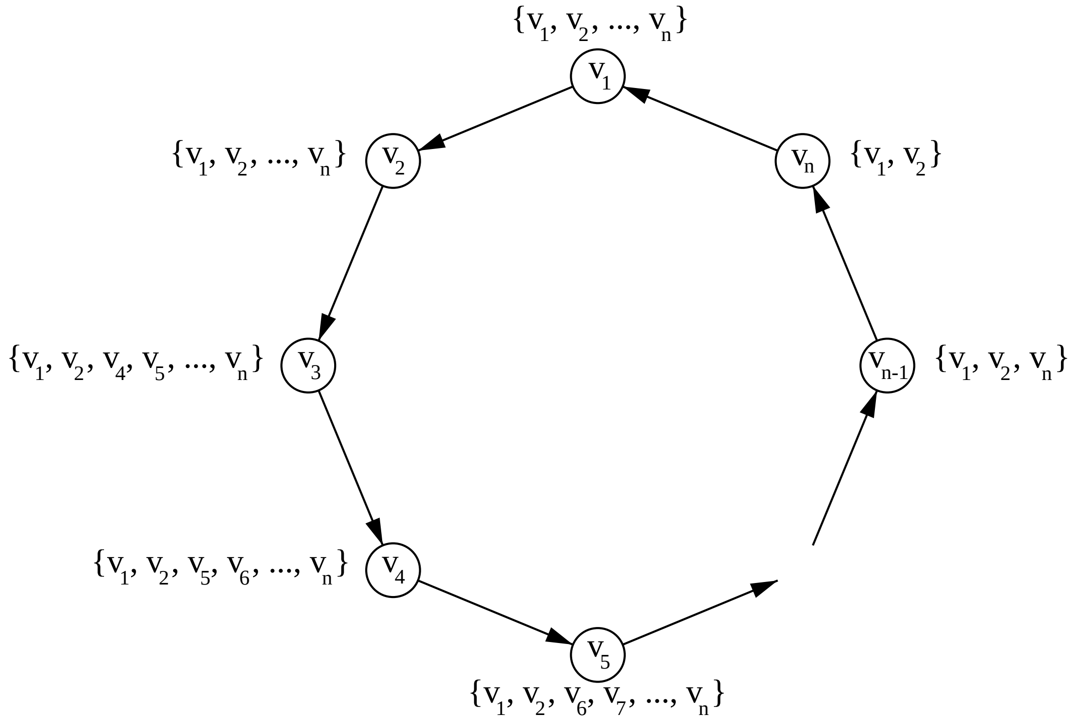


FIGURE 3.11: A cycle with partially computed successor sets.

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 $Succ(Comp(v))$.

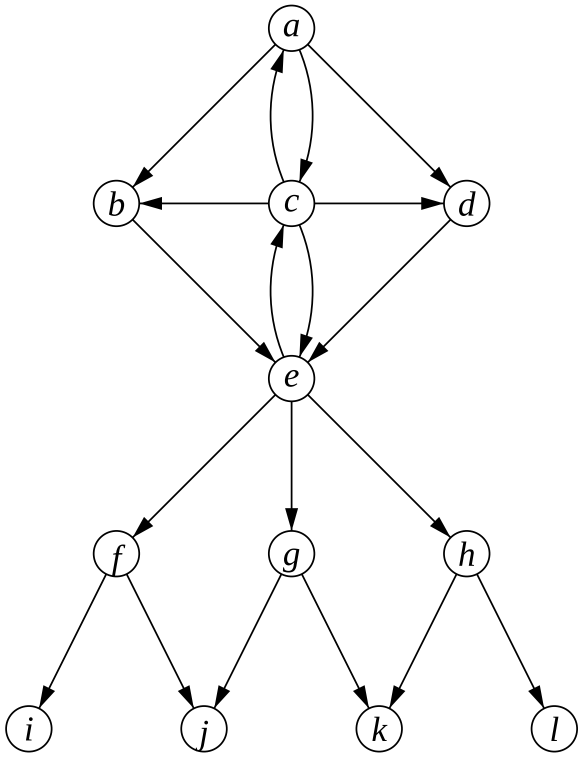


FIGURE 3.12: An example graph that leads to redundant operations in SIMPLE_TC.

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

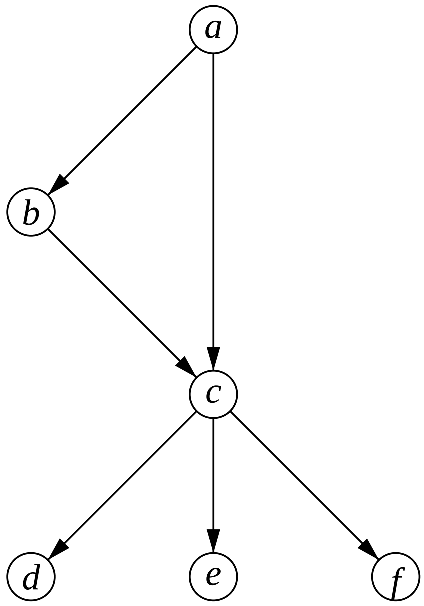


FIGURE 3.13: An example graph that causes SIMPLE_TC to add $Succ(c)$ twice to $Succ(a)$.

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.

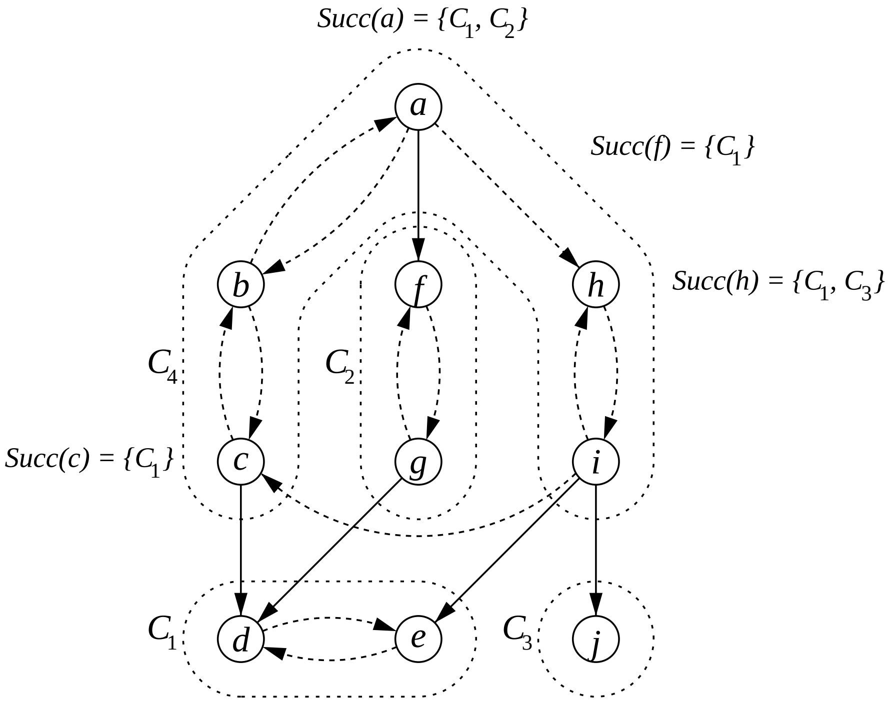


FIGURE 3.15: Graph G and the non-empty partial successor sets of the vertices when CR_TC is applied to G.

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.

```
procedure STACK_TC(v);
begin
    Root(v) := v; Comp(v) := Nil;
    PUSH(v, vstack);
    SavedHeight(v) := HEIGHT(cstack);
    SelfLoop(v) := false;
    for each vertex w such that (v, w) ∈ E do begin
        if w = v then SelfLoop(v) := true
        else begin
            if w is not already visited then STACK_TC(w);
            if Comp(w) = Nil then Root(v) := MIN(Root(v), Root(w))
            else if (v, w) is not a forward edge then
                PUSH(Comp(w), cstack);
        end
    end;
    if Root(v) = v then begin
        create a new component C;
        if TOP(vstack) ≠ v or SelfLoop(v) then Succ(C) := {C}
        else Succ(C) := ∅;
        sort the components in cstack between SavedHeight(v) and HEIGHT(cstack)
        into a topological order and eliminate duplicates;
        while HEIGHT(cstack) ≠ SavedHeight(v) do begin
            X := POP(cstack);
            if X ∉ Succ(C) then Succ(C) := Succ(C) ∪ {X} ∪ Succ(X);
        end;
        repeat
            w := POP(vstack);
            Comp(w) := C;
            insert w into component C;
        until w = v
    end
end;
begin /* Main program */
    vstack := ∅; cstack := ∅;
    for each vertex v ∈ V do
        if v is not already visited then STACK_TC(v)
end.
```

FIGURE 3.16: Algorithm STACK_TC.

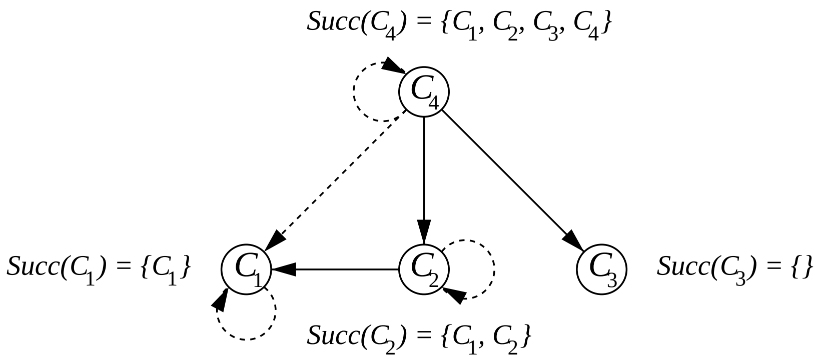


FIGURE 3.17: The condensation graph $\overline{G}$ of graph G presented in Figure 3.15(a) and the corresponding successor sets.

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; $e \le n^2$.
s	The number of strong components; $s \le n$.
s_{cyc}	The number of cyclic strong components; $s_{cyc} \le s$.
$\overline{e}$	The number of edges in the condensation graph without the self-loop edges; $\overline{e} \le e$.
$\overline{e}_r$	The number of edges in the transitive reduction of the condensation graph without the self-loop edges; $\overline{e}_r \le \overline{e}$.
$\overline{e}^+$	The number of edges in the transitive closure of the condensation graph; $\overline{e} \le \overline{e}^+$.
e_i	The number of intracomponent edges; $e_i \le e$.
e_o	The number of intercomponent edges; $e_o = e - e_i \le e$.
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; $e_{i1} \le e_i$.
e_{it}	The number of intracomponent tree edges; $e_{it} = n - s$ and $e_{it} \le e_i$.
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; $\overline{e}_r \le e_{o1} \le e_o$.
e_{o2}	The number of intercomponent edges (v, w) such that *Root*(w) is not in *Succ*(*Root*(v)) when SCHMITZ checks the edge; $\overline{e}_r \le e_{o2} \le e_o$.
e_{oct}	The number of intercomponent tree and cross edges; $\overline{e}_r \le e_{oct} \le e_o$.
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; $\overline{e}_r \le e_{oct1} \le e_{oct}$.
n_1	The number of nonroot vertices with a non-empty partial successor set in CR_TC; $n_1 \le n - s = e_{it}$.
σ	$\sum_{C \in \Pi} \sum_{v \in C} \mathit{Outdeg}(v)\, \mathit{Outdeg}(C)$, the time needed to compute the edge basis in Schmitz's variant algorithm. Here Π is the set of strong components and $\mathit{Outdeg}(C)$ 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; $\alpha \le 2.376$.
$Z(n)$	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. $\square$

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.