Chapter 4

Representing successor sets

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

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

4.1 Properties of common set representations

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

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

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

4.2 Interval representation

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

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

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

Example 4.1. A set $S_1 = \{0, 1, 2, 4, 5, 6, 8\}$ is stored as three intervals $[0, 2]$, $[4, 6]$, and $[8, 8]$ and a set $S_2 = \{0, 2, 3, 4, 5, 9, 10\}$ as three intervals $[0, 0]$, $[2, 5]$, and $[9, 10]$. If we insert number 7 to S_1, we get the set $S_1' = \{0, 1, 2, 4, 5, 6, 7, 8\}$, which is stored as two intervals $[0, 2]$ and $[4, 8]$. If we take the union of S_1' and S_2, we get the set $\{0, 1, 2, \ldots, 10\}$, which is stored as a single interval $[0, 10]$. $\square$

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

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

Proof. To produce set S, we draw k different elements from $\{1, 2, \ldots, n\}$ with each element having the same probability of being drawn. Let $I_{n,k}$ denote the expected number of intervals thus created. To get a recursion formula for $I_{n,k}$, we examine the case that element n is in S and the case that element n is not in S. The probability of the former case is k/n and the probability of the latter case is $(n-k)/n$. Assume first that n is in S. The expected number of intervals needed is $I_{n-1,k-1} + x$, where $I_{n-1,k-1}$ is the expected number of intervals needed to store a $k-1$ element set $S' \subseteq \{1, 2, \ldots, n-1\}$ and x is the expected number of additional intervals needed to store element n. If element $n-1$ is in S', no additional intervals are needed, but if $n-1$ is not in S', one additional interval is needed. Thus, x is the probability that element $n-1$ is not in S', which is $(n-1-(k-1))/(n-1) = (n-k)/(n-1)$. Assume now that n is not in S. The expected number of intervals needed is $I_{n-1,k}$, the expected number of intervals needed to store a k element set $S'' \subseteq \{1, 2, \ldots, n-1\}$. Finally, if $n = 0$, $k = 0$, or $k > n$, no intervals are needed, and if $n > 0$ and $k = 1$, one interval is needed. We get the following recursion formula:

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

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

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

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

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

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

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

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

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

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

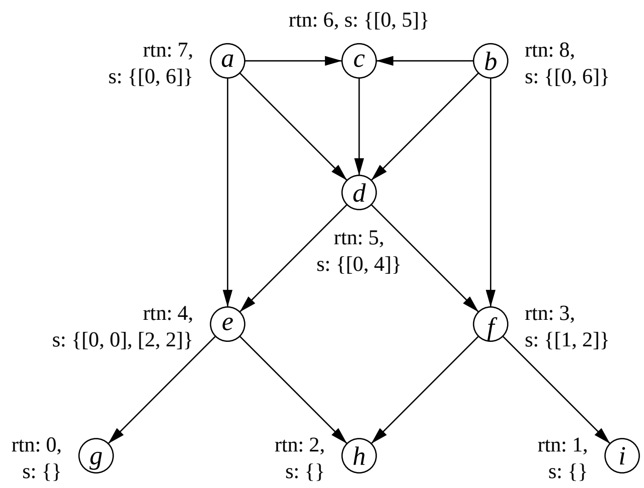

FIGURE 4.1: A DAG and the successor sets of its vertices represented as intervals.

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

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

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

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

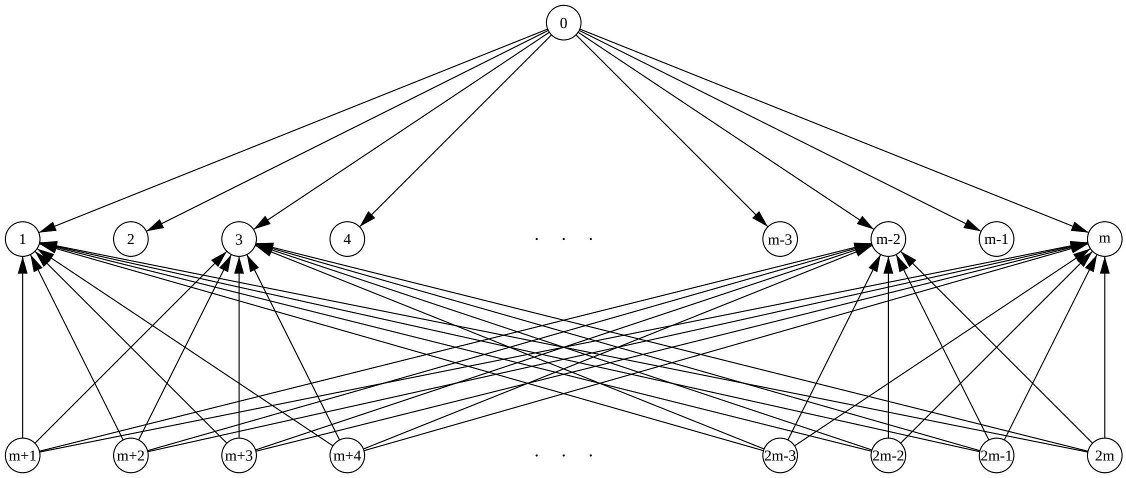

FIGURE 4.2: A graph the successor sets of which may require $\Omega(n^2)$ space.

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

A collection of intervals representing a successor set can be stored in several ways. Since the intervals are distinct, they can be ordered and stored in an ordered binary tree. The binary tree of intervals can be balanced, but usually this is not necessary, since the trees are small. Another possibility is to store the intervals in an ordered array. Let S, S_1, and S_2 be successor sets that are represented as collection of intervals I, I_1, and I_2, respectively. The union of S_1 and S_2 takes $O(|I_1|+|I_2|)$ time. Testing the presence of an element x in set S takes $O(\log(|I|))$ time in the worst case if the intervals are stored as a balanced binary tree or as an ordered array and $O(|I|)$ time if the intervals are stored in an unbalanced binary tree. Inserting an element x into set S takes $O(\log(|I|))$ in the worst case if the intervals are stored as a balanced binary tree and $O(|I|)$ time if the intervals are stored as an unbalanced binary tree or an ordered array. The binary tree representation requires twice as much memory space as the ordered array representation. The memory allocation is simpler for the array representation. In some transitive closure algorithms, e.g., in our algorithm STACK_TC, at most one successor set is under construction at any moment. In these algorithms, it is a good idea to use the array representation at least for all completely constructed successor sets. The successor set that is under construction can be stored as a binary tree to speed up insertion operations. If the number of intervals needed to represent a successor set is usually low, it is probably wise to store all successor sets as ordered arrays of intervals.

The idea of the interval representation came from [4], where Agrawal et al. studied how the transitive closure of an acyclic graph can be stored in a compressed form. The stored transitive closure can be used to directly answer queries about successors of a vertex without the need for computing the successors on-the-fly. Another goal they had was to incrementally propagate changes to the stored closure when the base relation changes. Since their goals were different from ours, their strategy also was different. Our goal here is the efficient computation of the full transitive closure, and a compact representation is a means to reach this goal. Agrawal et al. [4], on the other hand, tried to represent the transitive closure as compactly as possible, and the efficiency of the computation of the transitive closure was not important. Another difference is that their inputs were restricted to acyclic graphs whereas our inputs may have cycles. Our method overcomes this problem by detecting the strong components during the traversal.

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

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

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

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

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

4.3 Chain representation

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

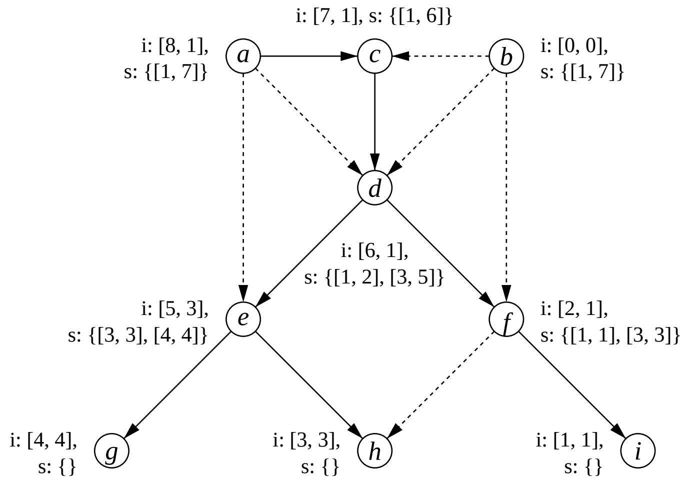

FIGURE 4.3: The interval representation of the graph in Figure 4.1 when the method of [4] is used.

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

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

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

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

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

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

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

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

The components adjacent from C are processed in the reverse topological order. The chains are stored as stacks. If a component X adjacent from C is the topmost element of its chain, we extend that chain by storing C on top of the chain. If no such component is found, i.e., no component adjacent from C is the topmost element of the chain containing it, we create a new chain and store C on top of that chain.

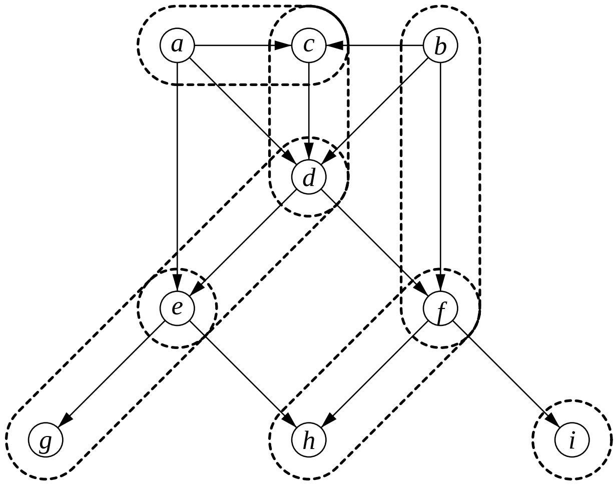

FIGURE 4.5: The chain partition of a graph when Simon's method [111, 112] is used.

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

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

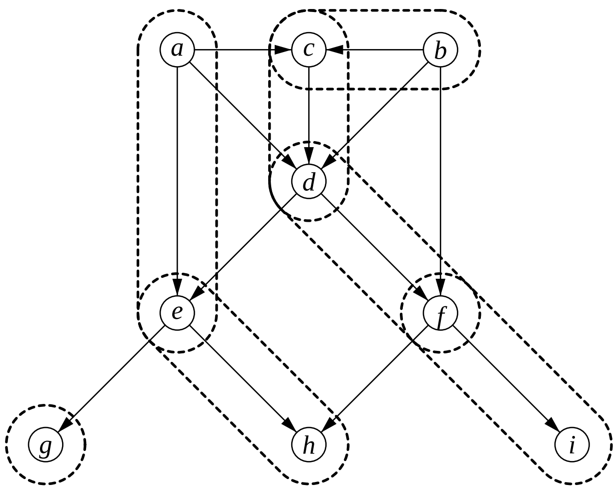

FIGURE 4.6: The chain partition of a graph when our method is used.

4.4 Avoiding multiple paths

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

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

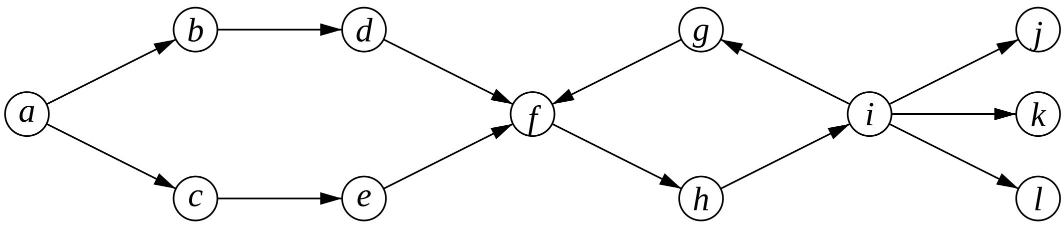

FIGURE 4.7: A directed graph with multiple paths between vertices.

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

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

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

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

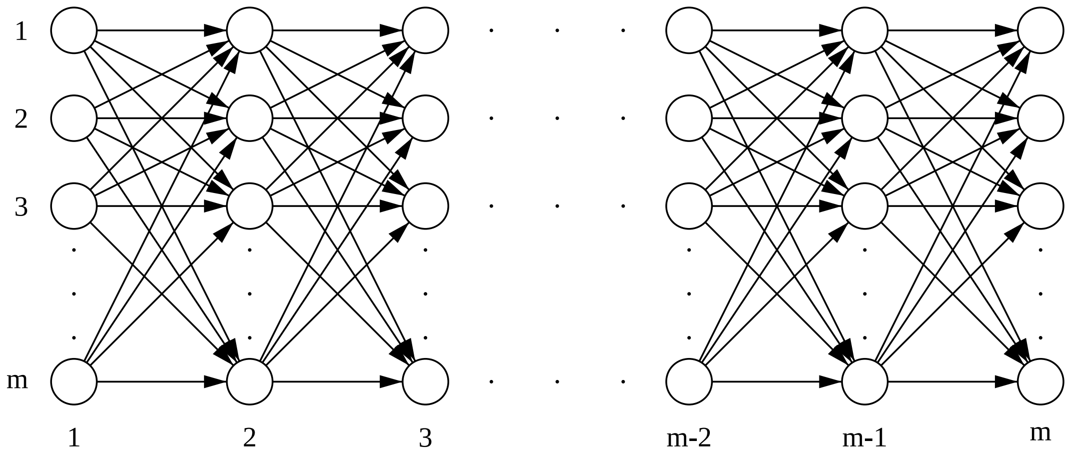

FIGURE 4.8: An example from Jakobson's paper[70].

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

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

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

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

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

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