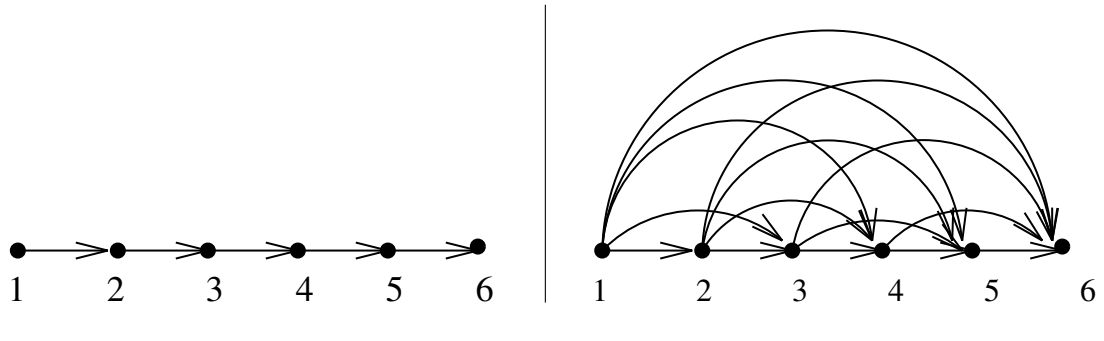

15.5 Transitive Closure and Reduction

Input description: A directed graph $G = (V, E)$.

Problem description: For *transitive closure*, construct a graph $G' = (V, E')$ with edge $(i, j) \in E'$ iff there is a directed path from i to j in G. For *transitive reduction*, construct a small graph $G' = (V, E')$ with a directed path from i to j in G' iff there is a directed path from i to j in G.

Discussion: Transitive closure can be thought of as establishing a data structure that makes it possible to solve reachability questions (can I get to x from y?) efficiently. After constructing the transitive closure, all reachability queries can be answered in constant time by simply reporting the appropriate matrix entry.

Transitive closure is fundamental in propagating the consequences of modified attributes of a graph G. For example, consider the graph underlying any spreadsheet model whose vertices are cells and have an edge from cell i to cell j if the result of cell j depends on cell i. When the value of a given cell is modified, the values of all reachable cells must also be updated. The identity of these cells is revealed by the transitive closure of G. Many database problems reduce to computing transitive closures, for analogous reasons.

There are three basic algorithms for computing transitive closure:

- The simplest algorithm just performs a breadth-first or depth-first search from each vertex and keeps track of all vertices encountered. Doing n such traversals gives an $O(n(n+m))$ algorithm, which degenerates to cubic time if the graph is dense. This algorithm is easily implemented, runs well on sparse graphs, and is likely the right answer for your application.

- Warshall's algorithm constructs transitive closures in $O(n^3)$ using a simple, slick algorithm that is identical to Floyd's all-pairs shortest-path algorithm

of Section 15.4 (page 489). If we are interested only in the transitive closure, and not the length of the resulting paths, we can reduce storage by retaining only one bit for each matrix element. Thus, $D_{ij}^k = 1$ iff j is reachable from i using only vertices $1, \ldots, k$ as intermediates.

- Matrix multiplication can also be used to solve transitive closure. Let M^1 be the adjacency matrix of graph G. The non-zero matrix entries of $M^2 = M \times M$ identify all length-2 paths in G. Observe that $M^2[i,j] = \sum_x M[i,x] \cdot M[x,j]$, so path (i,x,j) contributes to $M^2[i,j]$. Thus, the union $\cup_i^n M^i$ yields the transitive closure T. Furthermore, this union can be computed using only $O(\lg n)$ matrix operations using the fast exponentiation algorithm in Section 13.9 (page 423).

  You might conceivably win for large n by using Strassen's fast matrix multiplication algorithm, although I for one wouldn't bother trying. Since transitive closure is provably as hard as matrix multiplication, there is little hope for a significantly faster algorithm.

The running time of all three of these procedures can be substantially improved on many graphs. Recall that a strongly connected component is a set of vertices for which all pairs are mutually reachable. For example, any cycle defines a strongly connected subgraph. All the vertices in any strongly connected component must reach exactly the same subset of G. Thus, we can reduce our problem finding the transitive closure on a graph of strongly connected components that should have considerably fewer edges and vertices than G. The strongly connected components of G can be computed in linear time (see Section 15.1 (page 477)).

Transitive reduction (also known as *minimum equivalent digraph*) is the inverse operation of transitive closure, namely reducing the number of edges while maintaining identical reachability properties. The transitive closure of G is identical to the transitive closure of the transitive reduction of G. The primary application of transitive reduction is space minimization, by eliminating redundant edges from G that do not effect reachability. Transitive reduction also arises in graph drawing, where it is important to eliminate as many unnecessary edges as possible to reduce the visual clutter.

Although the transitive closure of G is uniquely defined, a graph may have many different transitive reductions, including G itself. We want the smallest such reduction, but there are multiple formulations of the problem:

- A linear-time, quick-and-dirty transitive reduction algorithm identifies the strongly connected components of G, replaces each by a simple directed cycle, and adds these edges to those bridging the different components. Although this reduction is not provably minimal, it is likely to be pretty close on typical graphs.

  One catch with this heuristic is that it might add edges to the transitive reduction of G that are not in G. This may or may not be a problem depending on your application.

- If all edges of our transitive reduction must exist in G, we have to abandon hope of finding the minimum size reduction. To see why, consider a directed graph consisting of one strongly connected component so that every vertex can reach every other vertex. The smallest possible transitive reduction will be a simple directed cycle, consisting of exactly n edges. This is possible if and only if G is Hamiltonian, thus proving that finding the smallest subset of edges is NP-complete.

  A heuristic for finding such a transitive reduction is to consider each edge successively and delete it if its removal does not change the transitive reduction. Implementing this efficiently means minimizing the time spent on reachability tests. Observe that a directed edge (i, j) can be eliminated whenever there is another path from i to j avoiding this edge.

- The minimum size reduction where we are allowed arbitrary pairs of vertices as edges can be found in $O(n^3)$ time. See the references below for details. However, the quick-and-dirty heuristic above will likely suffice for most applications, being easier to program as well as more efficient.

Implementations: The Boost implementation of transitive closure appears particularly well engineered, and relies on algorithms from [Nuu95]. LEDA (see Section 19.1.1 (page 658)) provides implementations of both transitive closure and reduction in C++ [MN99].

None of our usual Java libraries appear to contain implementations of either transitive closure or reduction. However, *Graphlib* contains a Java `Transitivity` library with both of them. See *http://www-verimag.imag.fr/∼cotton/* for details.

Combinatorica [PS03] provides Mathematica implementations of transitive closure and reduction, as well as the display of partial orders requiring transitive reduction. See Section 19.1.9 (page 661).

Notes: Van Leeuwen [vL90a] provides an excellent survey on transitive closure and reduction. The equivalence between matrix multiplication and transitive closure was proven by Fischer and Meyer [FM71], with expositions including [AHU74].

There is a surprising amount of recent activity on transitive closure, much of it captured by Nuutila [Nuu95]. Penner and Prasanna [PP06] improved the performance of Warshall's algorithm [War62] by roughly a factor of two through a cache-friendly implementation.

The equivalence between transitive closure and reduction, as well as the $O(n^3)$ reduction algorithm, was established in [AGU72]. Empirical studies of transitive closure algorithms include [Nuu95, PP06, SD75].

Estimating the size of the transitive closure is important in database query optimization. A linear-time algorithm for estimating the size of the closure is given by Cohen [Coh94].

Related Problems: Connected components (see page 477), shortest path (see page 489).