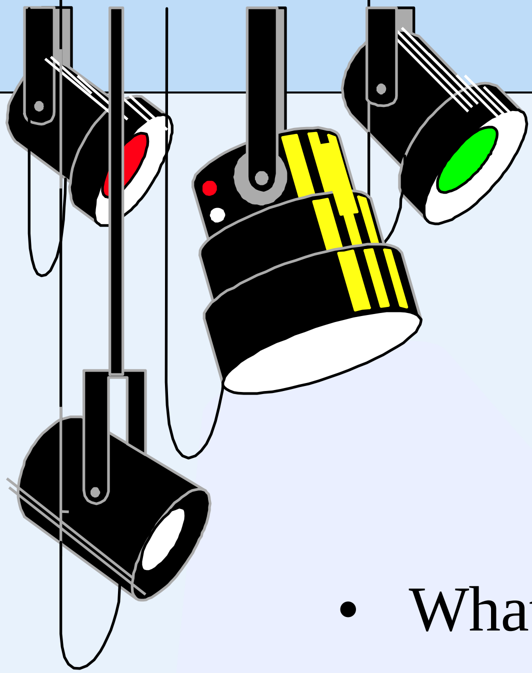
Reachability Queries



Outline: Reachability Query Evaluation

- What is reachability query?
- Reachability query evaluation based on matrix multiplication
- Warren’s algorithm (for generating transitive closures)
- Strassen’s algorithm (for matrix multiplication)
- Reachability based on tree encoding

Motivation

- **Efficient method to evaluate graph reachability queries**

  Given a directed graph *G*, check whether a node *v* is reachable from another node *u* through a path in *G*.

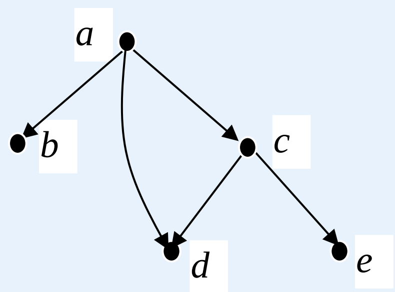
- **Application**
  - **XML data processing**
  - **Type checking in object-oriented languages and databases**
  - **Geographical navigation**
  - **Internet routing**
  - **CAD/CAM, CASE, office systems, software management**

Motivation

- **A simple method**

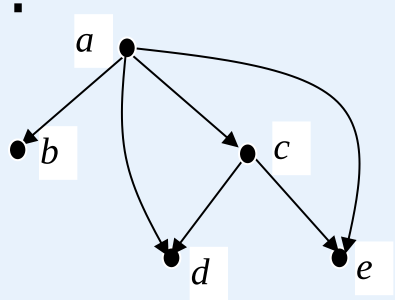
  - *store a transitive closure as a matrix*

G:



$$M = \begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 1 & 0 \\ b & 0 & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 1 & 1 \\ d & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 0 \end{array}$$

*G**:



The transitive closure *G** of a graph *G* is a graph such that there is an edge (*u*, *v*) in *G** iff there is path from *u* to *v* in *G*.

$$M^* = \begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 1 & 1 \\ b & 0 & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 1 & 1 \\ d & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 0 \end{array}$$

Matrix Multiplication

- **Definition**
  - Two matrices A and B are compatible if the number of columns of A equals the number of B.
  - If $A = (a_{ij})$ is an $m \times n$ matrix and $B = (b_{ij})$ is an $n \times p$ matrix, then their matrix product $C = A \times B$ is an $m \times p$ matrix $C = (c_{ik})$ such that

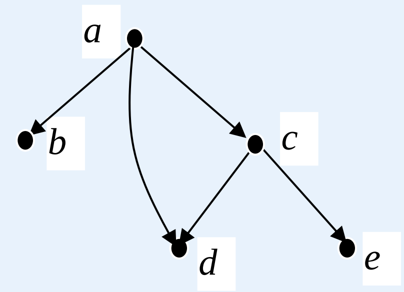
$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

for $i = 1, 2, \ldots, m$ and $k = 1, 2, \ldots, p$.

$$M \times M = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Each entry (i, j) in $M \times M$ represents a path of length 2 from i to j.

G:



Each entry (i, j) in $M \times M$ represents a path of length 2 from i to j.

Each entry (i, j) in $M \times M \times M$ represents a path of length 3 from i to j.

$\vdots$

Each entry (i, j) in $\overbrace{M \times M \times M \ldots \times M}^{k}$ represents a path of length k from i to j.

Define:

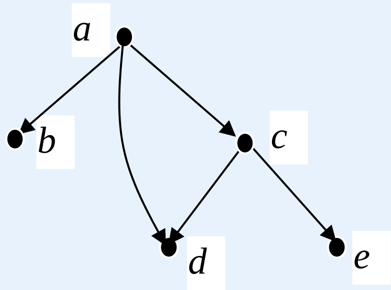
$$\boldsymbol{M^*} = M^{(1)} \vee M^{(2)} \vee M^{(3)} \ldots \vee M^{(n)}$$

Each entry (i, j) in $\boldsymbol{M^*}$ represents a path from i to j.

Time overhead: $O(n^4)$.
Space overhead: $O(n^2)$.
Query time: $O(1)$.

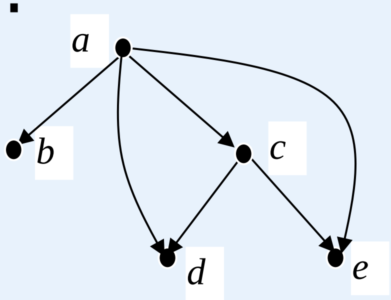
Example

G:



$$M = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

*G**:



$$M^* = M \vee (M \times M) = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Each entry (i, j) in P represents a path from i to j.

Warren's Algorithm

Warren's algorithm is a quite simple way to generate a boolean matrix to represent the transitive closure of a graph G. Assume that G is represented by a boolean matrix M in which $M(i, j) = 1$ if edge (i, j) is in G, and $M(i, j) = 0$ if (i, j) is not in G. Then, the matrix M' for the transitive closure of G can be computed from M, in which $M'(i, j) = 1$ if there exits a path from i to j in G, and $M'(i, j) = 0$ if there is no path from i to j in G. Warren's algorithm is given below:

Algorithm *Warren*

for $i = 2$ to n **do**

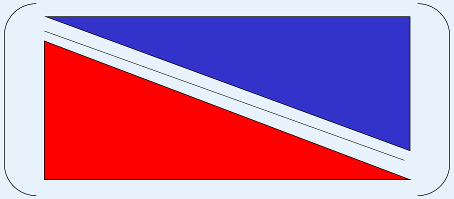
  for $j = 1$ to $i - 1$ **do**

  {**if** $M(i, j) = 1$ **then** set $M(i, *) = M(i, *) \vee M(j, *)$;}

for $i = 1$ to $n - 1$ **do**

  for $j = i + 1$ to n **do**

  {**if** $M(i, j) = 1$ **then** set $M(i, *) = M(i, *) \vee M(j, *)$;}

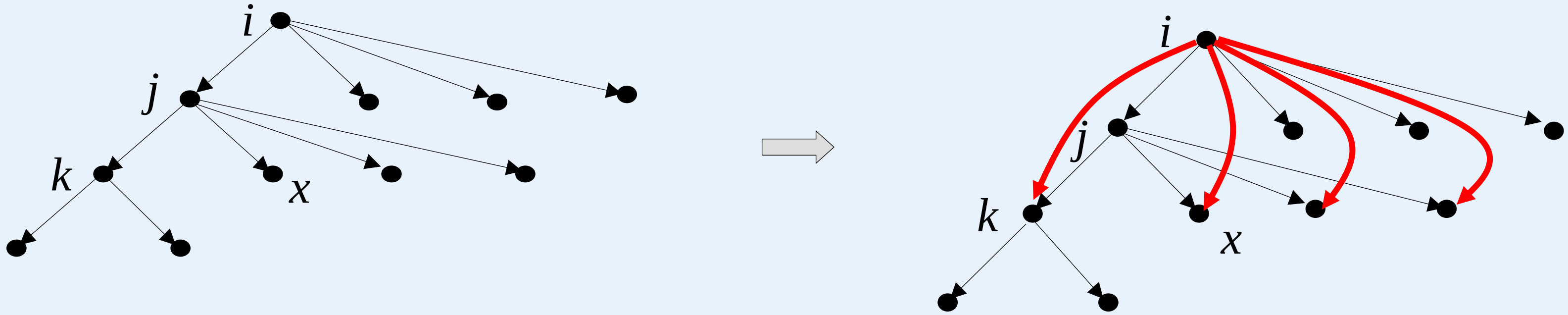


In the algorithm, $M(i, *)$ denotes row i of M.

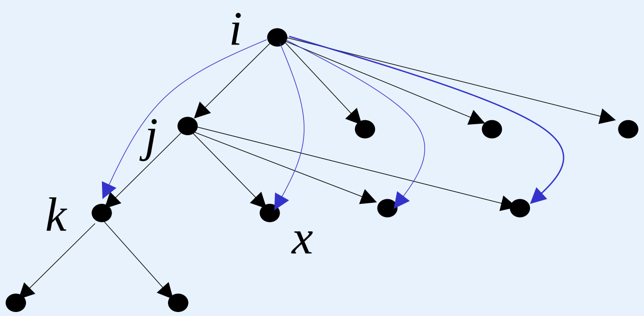
The theoretic time complexity of Warren's algorithm is $O(n^3)$.

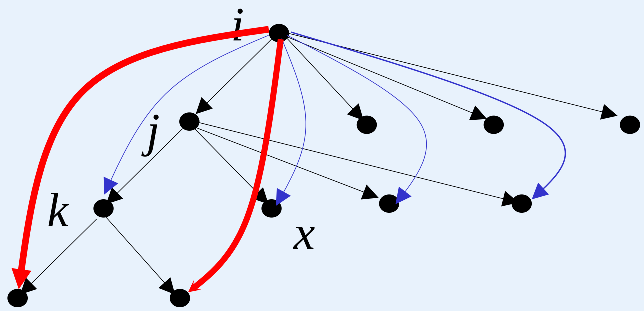
Reachability Queries

if $M(i, j) = 1$ **then** set $M(i, *) = M(i, *) \vee M(j, *)$



if $M(i, k) = 1$ **then** set $M(i, *) = M(i, *) \vee M(k, *)$





S. Warshall, "A Theorem on Boolean Matrices," *JACM*, 9. 1(Jan. 1962), 11 - 12.
H.S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *Commun. ACM* 18, 4 (April 1975), 218 - 220.

Strassen's Algorithm

Strassen's algorithm runs in $O(n^{\lg 7}) = O(n^{2.81})$ time. For sufficiently large values of n, it outperforms Warren's algorithm.

- **An overview of the algorithm**

Strassen's algorithm can be viewed as an application of a familiar design technique: divide and conquer. Consider the computation $C = A \times B$, where A, B, and C are $n \times n$ matrices. Assuming that n is an exact power of 2, we divide each of A, B, and C into four $n/2 \times n/2$ matrices, rewriting the equation $C = A \times B$ as follows:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

$$r = ae + bg$$
$$s = af + bh$$
$$t = ce + dg$$
$$u = af + dh$$

Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products. So the time complexity of the algorithm satisfies the following recursive equation:

$$T(n) = 8T(n/2) + O(n^2)$$

The solution of this equation is $T(n) = O(n^3)$.

Strassen discovered a different approach that requires only 7 recursive multiplications of $n/2 \times n/2$ matrices and $O(n^2)$ scalar additions and subtractions, yielding the recurrence:

$$\begin{aligned} T(n) &= 7T(n/2) + O(n^2) \\ &= O(n^{\lg 7}) \\ &= O(n^{2.81}). \end{aligned}$$

Strassen's algorithm works in four steps:

1. Divide the input matrices A and B into $n/2 \times n/2$ matrices.
2. Using $O(n^2)$ scalar additions and subtractions, computer 14 matrices $A_1, B_1, A_2, B_2, \ldots, A_7, B_7$, each of which is $n/2 \times n/2$.
3. Recursively compute the seven matrix products $P_i = A_i \times B_i$ for $i = 1, 2, \ldots, 7$.
4. Computer the desire submatrices r, s, t, u of the result matrix C by adding and/or subtracting various combinations of the P_i matrices, using only $O(n^2)$ scalar additions and subtraction.

$A_1 = a, A_2 = (a + b), A_3 = (c + d), A_4 = d, A_5 = (a + d), A_6 = (b - d), A_7 = (c - a)$

$B_1 = (f - h), B_2 = h, B_3 = e, B_4 = (g - d), B_5 = (e + h), B_6 = (g + h), B_7 = (e + f)$

$r = ae + bg = P_5 + P_4 - P_2 + P_6$, $s = af + bh = P_1 + P_2$,

$t = ce + dg = P_3 + P_4$, $u = af + dh = P_5 + P_1 - P_3 + P_7$.

7 matrix multiplication, 18 matrix additions and subtractions.

Assume that $n = 2^m$. We have

$$T(2^m) = 7T(2^{m-1}) + 18(2^{m-1})^2.$$

$$A_m = 7A_{m-1} + 18(2m-1)^2, \qquad A_1 = 18.$$

$$G(x) = A_1 + A_2 x + A_3 x^2 + \ldots$$

$$= A_1 + (7A_1 + 18\cdot 2^2)x$$

$$+ (7A_2 + 18\cdot 2^3)x^2$$

$$\ldots \ldots$$

$$= 18 + 7x\, G(x) + 18\cdot 4x/(1 - 4x)$$

$$(1 - 7x)G(x) = 18(4x/(1 - 4x) + 1) = 18/(1 - 4x)$$

Reachability Queries

$$(1 - 7x)G(x) = 18(4x/(1 - 4x) + 1) = 18/(1 - 4x)$$

$$G(x) = 18/(1 - 4x)(1 - 7x) = 18\left(\frac{-4/3}{1 - 4x} + \frac{7/3}{1 - 7x}\right)$$

$$G(x) = 6\sum_{k=0}^{\infty}(7^{k+1} - 4^{k+1})x^k$$

⇩

$$A_m = 6(7^m - 4^m), \qquad m = \log_2 n$$

$$= O(6 \cdot 7^{\log_2 n})$$

$$= O(6 \cdot n^{\log_2 7})$$

$$= O(n^{2.81})$$

- **Determining the submatrix products**

It is not clear exactly how Strassen discovered the submatrix products that are the key to making his algorithm work. Here, we reconstruct one plausible discovery method.

Write $P_i = A_i \times B_i$

$$= (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d)\ (\beta_{i2}e + \beta_{i1}f + \beta_{i3}g + \beta_{i4}h),$$

where the coefficients α_{ij}, β_{ij} are all drawn from the set {-1, 0, 1}. We guess that each product is computed by adding or subtracting some of the submatrices of *A*, adding or subtracting some of submatrices of *B*, and then multiplying the two results together.

Reachability Queries

$$P_i = A_i \times B_i = (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d)\ (\beta_{i1}e + \beta_{i2}f + \beta_{i3}g + \beta_{i4}h)$$

$$= (a\ b\ c\ d) \begin{pmatrix} \alpha_{i1} \\ \alpha_{i2} \\ \alpha_{i3} \\ \alpha_{i4} \end{pmatrix} (\beta_{i1}\ \beta_{i2}\ \beta_{i3}\ \beta_{i4}) \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix}$$

$$= (a\ b\ c\ d) \begin{pmatrix} \alpha_{i1}\beta_{i1} & \alpha_{i1}\beta_{i2} & \alpha_{i1}\beta_{i3} & \alpha_{i1}\beta_{i4} \\ \alpha_{i2}\beta_{i1} & \alpha_{i2}\beta_{i2} & \alpha_{i2}\beta_{i3} & \alpha_{i2}\beta_{i4} \\ \alpha_{i3}\beta_{i1} & \alpha_{i3}\beta_{i2} & \alpha_{i3}\beta_{i3} & \alpha_{i3}\beta_{i4} \\ \alpha_{i4}\beta_{i1} & \alpha_{i4}\beta_{i2} & \alpha_{i4}\beta_{i3} & \alpha_{i4}\beta_{i4} \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix}$$

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

$r = ae + bg$
$s = af + bh$
$t = ce + dg$
$u = af + dh$

$r = ae + bg$

$$= (a\ b\ c\ d) \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix}$$

So r is represented by a matrix:

$$\begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

'·' – represents 0.
'+' – represents +1.
'-' – represents -1.

Reachability Queries

$s = af + bh$

$$= \begin{pmatrix} \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$t = ce + dg$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \end{pmatrix}$$

$s = cf + dh$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}$$

We will create 7 matrices in such a way that the above 4 matrices can be generated by addition and subtraction operations over these 7 matrices. Furthermore, the 7 matrices themselves can be produced by 7 multiplications and some additions and subtractions.

Reachability Queries

$P_1 = A_1 \cdot B_1 = a \cdot (f - h) = af - ah$ $P_2 = A_2 \cdot B_2 = (a + b) \cdot h = ah + bh$

$$= \begin{pmatrix} \cdot & + & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \qquad = \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$s = af + bh$

$$= \begin{pmatrix} \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = P_1 + P_2$$

$P_3 = A_3 \cdot B_3 = (c + d) \cdot e = ce + de$ $\quad$ $P_4 = A_4 \cdot B_4 = d \cdot (g - e) = dg - de$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix} \qquad = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{pmatrix}$$

$t = ce + dg$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \end{pmatrix} = P_3 + P_4$$

$P_5 = A_5 \cdot B_5 = (a + d) \cdot (e + h)$
$= ae + ah + de + dh$

$$= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}$$

$P_6 = A_6 \cdot B_6 = (b - d) \cdot (g + h)$
$= bg + bh - dg - dh$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{pmatrix}$$

$r = ae + bg$

$$= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = P_5 + P_4 - P_2 + P_6$$

$$P_7 = A_7 \cdot B_7 = (a - c) \cdot (e + f)$$
$$= ae + af - ce - cf$$

$$= \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & - & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

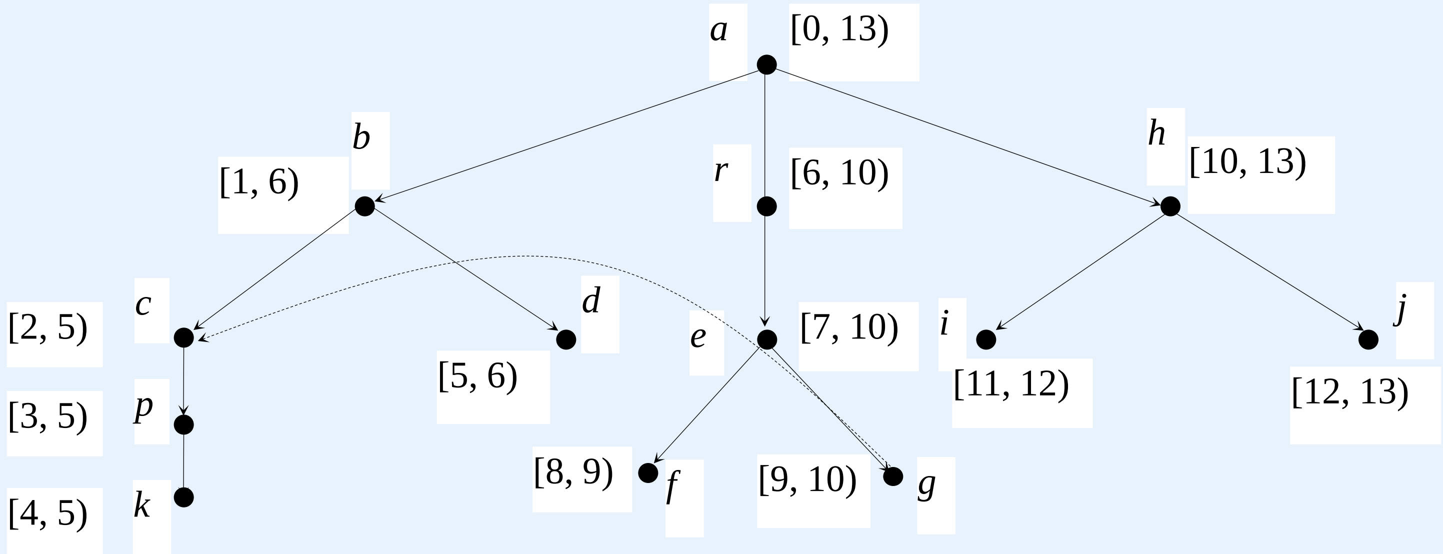
$$u = cf + dh$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} = P_5 + P_1 - P_3 - P_7$$

First kind of tree encoding

- **Definition**
  - We can assign each node *v* in a tree *T* an interval $[\alpha_v, \beta_v)$, where α_v is *v*'s preorder number (denoted *pre*(*v*)) and β_v - 1 is equal to the largest preorder number among all the nodes in *T*[*v*] (subtree rooted at *v*).
  - So another node *u* labeled $[\alpha_u, \beta_u)$ is a descendant of *v* (with respect to *T*) iff $\alpha_u \in [\alpha_v, \beta_v)$.
  - If $\alpha_u \in [\alpha_v, \beta_v)$, we say, $[\alpha_u, \beta_u)$ is subsumed by $[\alpha_v, \beta_v)$. This method is called the *tree labeling*.

Example:



For a directed graph, the intervals cannot be used to check reachability. The containment is just a sufficient condition, not a necessary condition.

Reachability checking based on tree encoding

Directed acyclic graphs (DAGs)

- Find a spanning tree *T* of *G*, and assign each node *v* an interval.
- Examine all the nodes in *G* in reverse topological order and do the following:

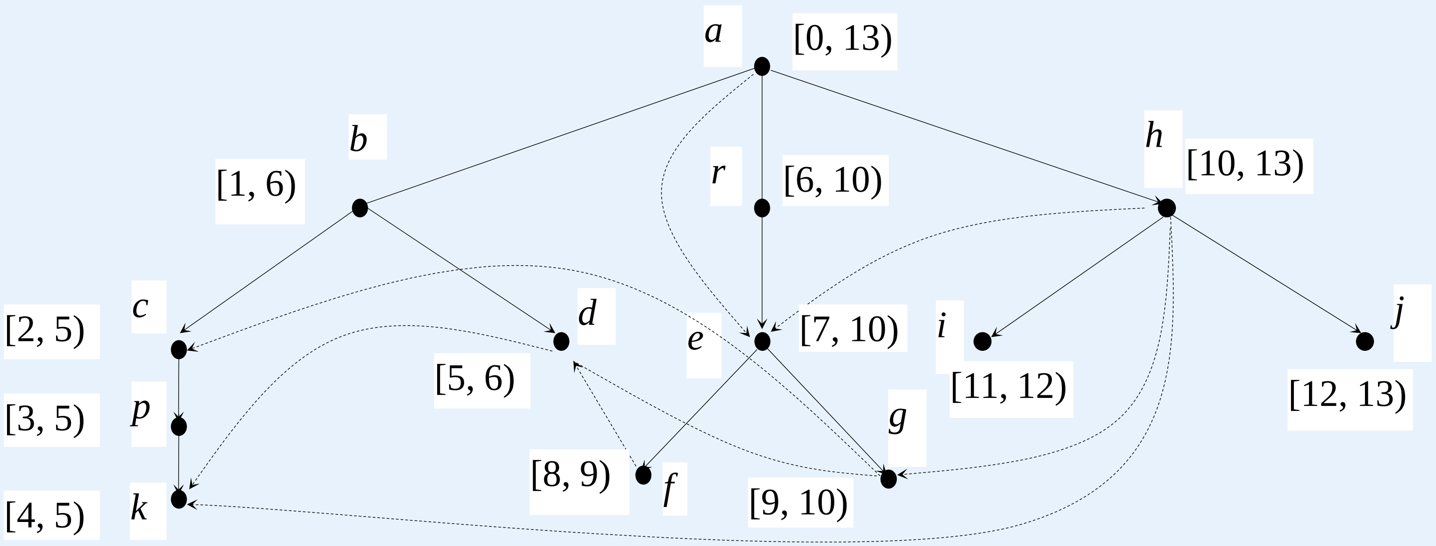
For every edge (*p*, *q*), add all the intervals associated with the node *q* to the intervals associated with the node *p*.

When adding an interval [*i*, *j*) to the interval sequence associated with a node, if an interval [*i'*, *j'*) is subsumed by [*i*, *j*), it will be discarded from the sequence. In other words: if $i' \in [i, j)$, then discard [*i'*, *j'*]. On the other hand, if an interval [*i'*, *j'*) is equal to [*i*, *j*) or subsumes [*i*, *j*). [*i*, *j*) will not be added to the sequence. Otherwise, [*i*, *j*) will be inserted.

Topological order of a directed acyclic graph:

Linear ordering of the vertices of G such that if $(u, v) \in E$, then u appears somewhere before v.

Example:



Topological order: *a*, *b*, *r*, *h*, *e*, *f*, *g*, *d*, *c*, *p*, *k*, *i*, *j*

Reverse topological order:

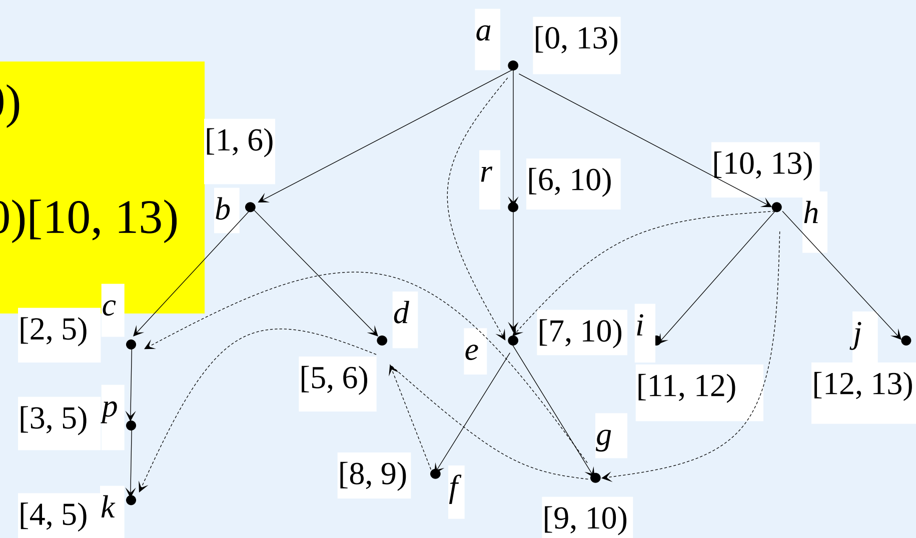
A sequence of the nodes of *G* such that for any edge (*u*, *v*) *v* appears before *u* in the sequence.

k, p, c, d, f, g, i, j, e, r, b, h, a

Reverse topological order

$L(k) = [4, 5)$
$L(p) = [3, 5)$
$L(c) = [2, 5)$
$L(d) = [4, 5)[5, 6)$
$L(f) = [4, 5)[5, 6)[8, 9)$
$L(g) = [2, 5)[5, 6)[9, 10)$
$L(i) = [11, 12)$
$L(j) = [12, 13)$
$L(e) = [2, 5)[5, 6)[7, 10)$

$L(r) = [2, 5)[5, 6)[6, 10)$
$L(b) = [1, 6)$
$L(h) = [2, 5)[5, 6)[7, 10)[10, 13)$
$L(a) = [10, 13)$



Generation of interval sequences

- Create interval sequences for all the nodes along the reverse topological order
- First of all, we notice that each leaf node is exactly associated with one interval, which is trivially sorted.
- Let v_1, ..., v_l be the child nodes of v, associated with the interval sequences L_1, ..., L_l, respectively.
- Assume that the intervals in each L_i are sorted according to the first element in each interval. We will merge all L_i's into the interval sequence associated L with v as follows.

- Let $[a_1, b_1)$ (from L) and $[a_2, b_2)$ (from L_i) be the interval encountered. We will perform the following checkings:

$L = \ldots [a_1, b_1) \ldots$
$L_i = \ldots [a_2, b_2) \ldots$

-If $a_2 >= a_1$ **then**

{**if** $a_2 \in [a_1, b_1)$ **then** go to the interval next to $[a_2, b_2)$ and compare it with $[a_1, b_1)$ in a next step

else go to the interval next to $[a_1, b_1)$ and compare it with $[a_2, b_2)$ in a next step.}

-If $a_1 > a_2$ **then**

{**if** $a_1 \in [a_2, b_2)$ **then** remove $[a_1, b_1)$ from L and compare the interval next to $[a_1, b_1)$ with $[a_2, b_2)$ in a next step

else insert $[a_2, b_2)$ into L before $[a_1, b_1)$.}

Obviously, $|L| \leq b$ (the number of the leaf nodes in the spanning tree T) and the intervals in L are sorted. The time spent on this process is $O(d_v b)$, where d_v represents the outdegree of v. So the whole cost is bounded by

$$O(\sum_v d_v b) = O(be).$$

Reachability checking for DAGs

- Let u and v be two nodes of G.
- u is a descendant of v iff there exists an interval $[\alpha, \beta)$ in $L(v)$ such that $\alpha_u \in [\alpha, \beta)$.

Example:

$[\alpha_k, \beta_k) = [4, 5)$
$L(r) = [2, 5)[5, 6)[6, 10)$

⇨

Node k is a descendant of node r.

Reachability checking for cyclic graphs

- Using the Tarjan's algorithm to recognize all the *strongly connected components* (*SCCs*). In each SCC, any two nodes are reachable from each other.
- Collapse each SCC to a single node. In this way, any cyclic graph *G* is transformed to a DAG *G'*.
- Let *u* and *v* be to two nodes in *G*. Check their reachability according to two cases:
  - *u* and *v* are in two different SCC.
  - *u* and *v* are in the same SCC.

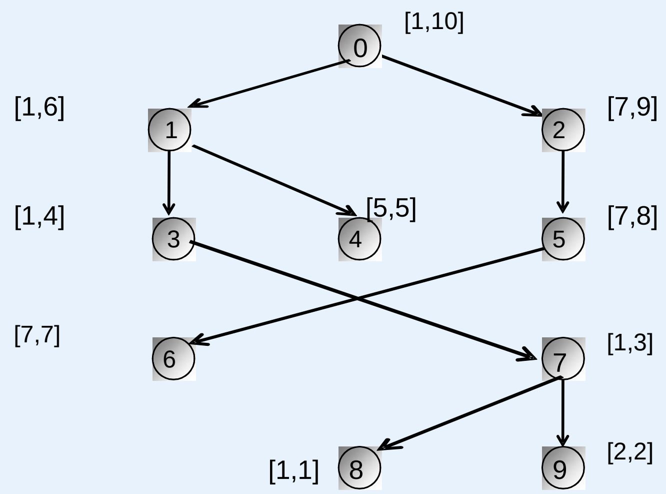
Second kind of tree encoding: Using tree encoding as a filter

- Each node v in a tree T is labeled with a with a range

$$I_v = [r_x, r_v],$$

where r_v is the postorder number of v (the postorder numbers are assumed to begin at 1) and r_x is the lowest postorder number of any node x in the subtree rooted at v (i.e., including v).
- This approach guarantees that the containment between intervals is equivalent to the reachability relationship between the nodes, since the postorder traversal enters a node before all of its descendants have been visited.
- In other words, $u \rightsquigarrow v \Leftrightarrow I_v \subseteq I_u$.

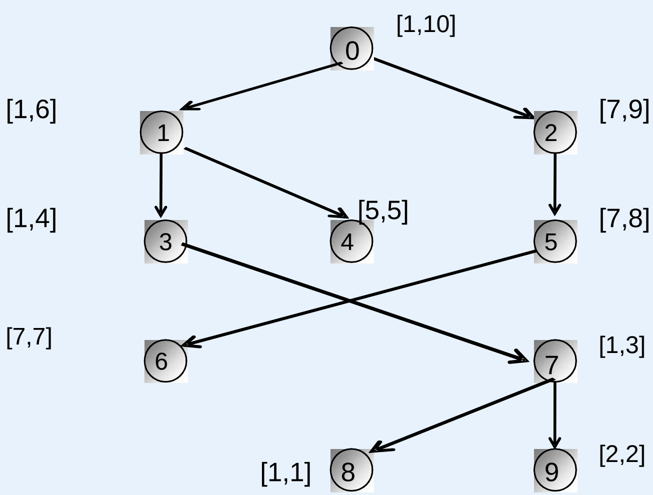
Example:

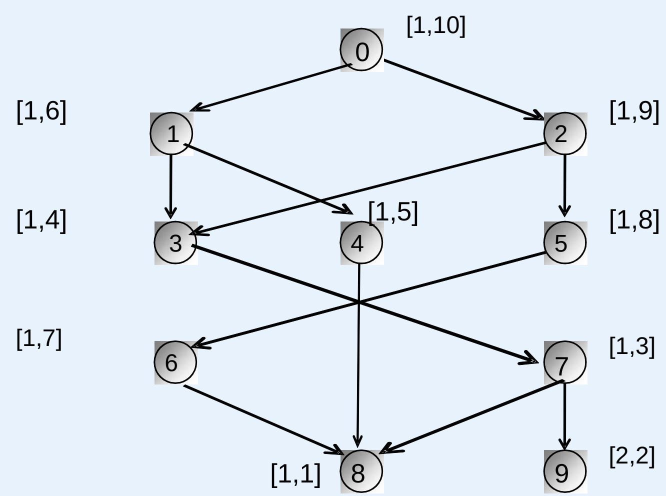


The above figure shows the interval labeling on a tree, assuming that the children are ordered from left to right. It is easy to see that reachability can be answered by interval containment. For example, $1 \rightsquigarrow 9$, since $I_9 = [2, 2] \subset [1, 6] = I_1$, but $2 \not\rightsquigarrow 7$, since $I_7 = [1, 3] \not\subset [7, 9] = I_2$.

Using tree encoding as a filter

To generalize the interval labeling to a DAG *G*, we have to ensure that a node is not visited more than once, and a node will keep the postorder number r_v of its first visit. Its r_x is now the lowest postorder number in the sub-graph rooted at *v*.

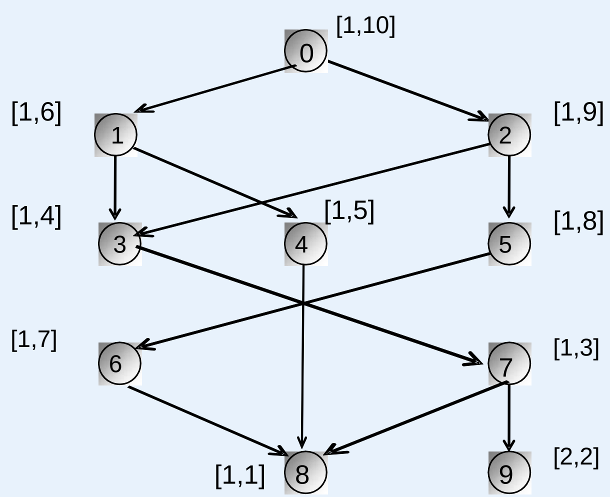




Reachability Queries

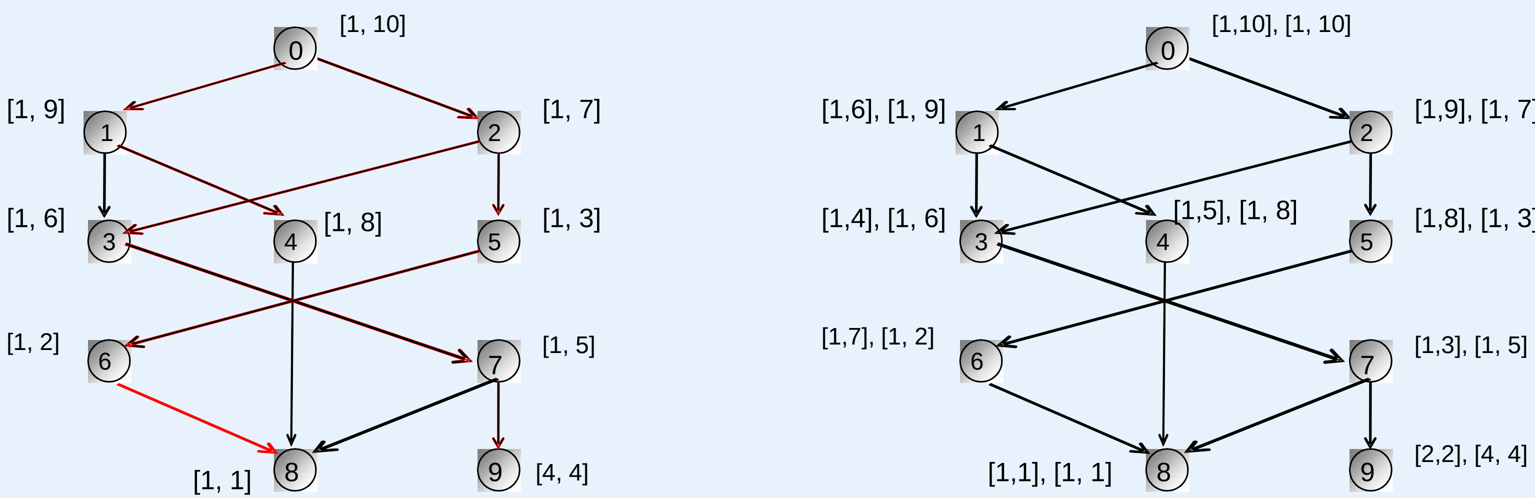
The above shows an interval labeling on a DAG, assuming a left to right ordering of the children. As one can see, interval containment of nodes in a DAG is not exactly equivalent to reachability.

For example, $5 \not\rightsquigarrow 4$, but $I_4 = [1, 5] \subseteq [1, 8] = I_5$. In other words, $I_v \subseteq I_u$ does not imply that $u \rightsquigarrow v$. On the other hand, one can show that $I_v \not\subset I_u \Rightarrow u \not\rightsquigarrow v$. (So the containment is a necessary condition, not a sufficient condition.)



Reachability Queries

- Instead of using a single interval, one can employs multiple intervals that are obtained via random graph traversals.
- We use the symbol *d* to denote the number of intervals to keep per node, which also corresponds to the number of graph traversals used to obtain the label.
- The following figure shows a DAG labeling using 2 intervals (the first interval assumes a left-to-right ordering of the children, whereas the second interval assumes a right-to-left ordering).



Index construction

An interval I_u^i is denoted as

$$I_u^i = [I_u^i[1], I_u^i[2]\ ] = [r_x, r_u]$$

Algorithm 1: Randomized Intervals

RandomizedLabeling(G, d):

foreach $i \leftarrow 1$ to d **do** //d – number of intervals for each node

  $r \leftarrow 1$ // global variable: postorder number of node

  $Roots \leftarrow \{n : n \in roots(G)\}$

  foreach $x \in$ *Roots in random order* **do**

    Call **RandomizedVisit**(x, i, G)

RandomizedVisit(x, i, G) :

if *x visited before* **then return**

foreach $y \in Children(x)$ *in random order* **do**

  Call **RandomizedVisit**(y, i, G)

$r_c^* \leftarrow min\{I_c^i[1] : c \in Children(x)\}$

$I_x^i \leftarrow [min(r, r_c^*), r]$

$r \leftarrow r + 1$

Reachability queries

- Assume that each node is associated with an single interval.
- To answer reachability queries between two nodes, *u* and *v*, we will first check whether $I_v \not\subset I_u$. If so, we can immediately conclude that $u \not\rightsquigarrow v$.
- On the other hand, if $I_v \subseteq I_u$, nothing can be concluded immediately since we know that the index can have false positives, i.e., exceptions. In this case, a DFS (depth-first search) is conducted, with recursive containment check based pruning, to answer queries. In the worst case, it needs O(*n*) time. Another way is to check the exception lists associated with the nodes:

$$E_x = \{y : (x, y) \text{ is an exception, i.e., } I_y \subseteq I_x \text{ and } x \not\rightsquigarrow y\}.$$

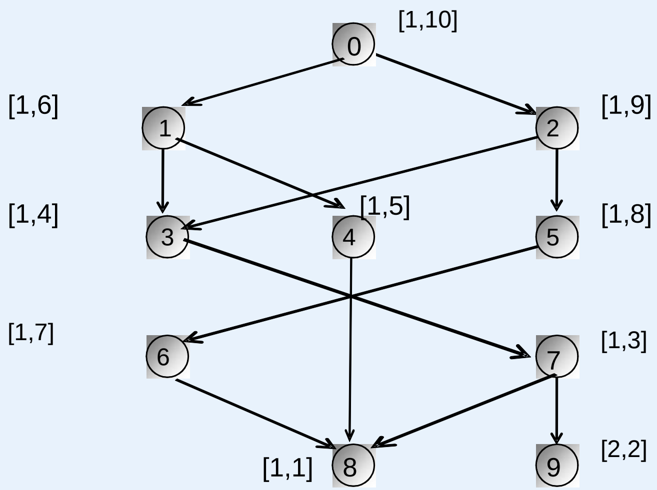
DFS with prunning

Algorithm 2: Reachability Testing (*for the case of only one interval*)

Reachable(u, v, G):

```
if I_v ⊄ I_u then
        return False // u ↝̸ v
else if use exception lists then
        if v ∈ E_u then return False // u ↝̸ v
        else return True // u ↝ v
else // DFS with pruning
        foreach c ∈ Children(u) such that I_v ⊆ I_c do
                if Reachable(c, v, G) then
                        return True // u ↝ v
        return False // u ↝̸ v
```

Reachability Queries



Exception lists:

$E_2 = \{1, 4\}$
$E_4 = \{3, 7, 9\}$
$E_5 = \{1, 3, 4, 7, 9\}$
$E_6 = \{1, 3, 4, 7, 9\}$