DIGRAPH: directed graph
- edge s-t from s to t
- edge t-s from t to s

Can you get there from here?
Basic definitions

CONNECTIVITY
- path from s to t in undirected graph

REACHABILITY
- directed path from s to t in digraph

STRONG CONNECTIVITY
- directed paths from s to t AND from t to s

Connectivity ADT implementation (last lecture)
- query: $O(1)$
- preprocessing: $O(E)$
- space: $O(V)$

Can we do as well for reachability and strong connectivity?
void dfsR(Graph G, Edge e, int pre[], int post[]) {
    link t; int i, v, w = e.w; Edge x;
    pre[w] = cnt0++;
    for (t = G->adj[w]; t != NULL; t = t->next)
        if (pre[t->v] == -1)
            dfsR(G, EDGE(w, t->v), pre, post);
    post[w] = cnt1++;
}

void GRAPHsearch(Graph G, int pre[], int post[]) {
    int v;
    cnt0 = 0; cnt1 = 0; depth = 0;
    for (v = 0; v < G->V; v++)
        { pre[v] = -1; post[v] = -1; }
    for (v = 0; v < G->V; v++)
        if (pre[v] == -1)
            search(G, EDGE(v, v), pre, post);
}

Need both PREORDER and POSTORDER numbering
DFS forests

Structure determined by digraph AND search dynamics

- use pre- and post- numbering to distinguish edge types

Edge types
- TREE
- BACK
- DOWN
- CROSS

ONLY the FIRST tree has the set of nodes reachable from its root
**Transitive closure**

Digraph $G$

**Transitive closure** $G^*$ has edge from $s$ to $t$ iff there is a directed path from $s$ to $t$ in $G$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOT symmetric**

supports $O(1)$ reachability queries with $O(V^2)$ space
Boolean matrices and paths in graphs

Adjacency matrix $A$
- $A[s][t]$ is 1 iff path from $s$ to $t$

Square $A \times A = A^2$
- $A[s][t]$ is 1 iff path from $s$ to $t$ of length 2 in $A$ [s-k-t]

Reflexive square $A + A^2$
- $A[s][t]$ is 1 iff path from $s$ to $t$ of length < 2 in $A$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Transitive closure: $A + A^2 + A^3 + A^4 + A^5 + ...$
- same as $\lg V$ reflexive squares
- $[ A + (A + A^2)^2 = A + A^2 + A^3 + A^4 ]$
leads to easy $V^3 \lg V$ transitive closure algorithm
Warshall's algorithm

Method of choice for transitive closure of a dense graph
- running time proportional to $V^3$

```c
for (k = 0; k < G->V; k++)
    for (s = 0; s < G->V; s++)
        if (G->tc[s][k] == 1)
            for (t = 0; t < G->V; t++)
                if (G->tc[k][t] == 1) G->tc[s][t] = 1;
```

Proof of correctness (induction on $k$)
- there is a path from $s$ to $t$ (with no nodes $> k$) if
  EITHER
    - there is path from $s$ to $k$ (with no nodes $> k-1$)
      AND a path from $k$ to $t$ (with no nodes $> k-1$)
    OR there is a path from $s$ to $t$ (with no nodes $> k-1$)
Warshall's algorithm (example)
Transitive closure lower bound

Consider Boolean (0-1) matrices

Premise: Matrix multiplication is not easy
  - grade-school algorithm: \( V^3 \)
  - best known: \( V^c, c>2 \) [practical?]

THM: Transitive closure is no easier than matrix multiplication

Proof:
  - Given a matrix multiplication problem
  - can solve it with a TC algorithm

\[
\begin{array}{ccc}
I & A & 0 \\
0 & I & B \\
0 & 0 & I \\
\end{array} = \begin{array}{ccc}
I & A & AB \\
0 & I & B \\
0 & 0 & I \\
\end{array}
\]

O\((V^2)\) TC would yield O\((V^2)\) matrix multiply (not likely)
Package DFS to implement reachability ADT
  • run new DFS for each vertex

```c
void TCdfsR(Graph G, int v, int w)
{
    link t;
    G->tc[v][w] = 1;
    for (t = G->adj[w]; t != NULL; t = t->next)
        if (G->tc[v][t->v] == 0)
            TCdfsR(G, v, t->v);
}

void GRAPHtc(Graph G, Edge e)
{
    int v, w;
    G->tc = malloc2d(G->V, G->V);
    for (v = 0; v < G->V; v++)
        for (w = 0; w < G->V; w++)
            G->tc[v][w] = 0;
    for (v = 0; v < G->V; v++) TCdfsR(G, v, v);
}

int GRAPHreach(Graph G, int s, int t)
{
    return G->tc[s][t];
}
```

Running time? less than VE (V^2 for sparse graphs)
Violates lower bound? NO (worst case still V^3)
Abstract transitive closure

ADT function for reachability in digraphs

THM: DFS-based transitive closure provides
- $VE$ preprocessing time
- $V^2$ space
- constant query time

GOAL:
- $V^2$ (or $VE$) preprocessing time
- $V$ space
- constant query time

$V^2$ preprocessing guarantee not likely by TC lower bound

Next attempt:
- is the problem easier if there are no cycles (DAG)??
Topological sort (DAG)

DAG: directed acyclic graph

Topological sort: all edges point left to right

Reverse TS: all edges point right to left
DFS topological sort

Easy alg for reverse TS: DFS!
(postorder visit is reverse TS)

void TSdfsR(Graph G, int v, int ts[])
{
    int w;
    pre[v] = 0;
    for (w = 0; w < G->V; w++)
        if (G->adj[w][v] != 0)
            if (pre[w] == -1) TSdfsR(G, w, ts);
    ts[cnt0++] = v;
}

Quick hack for arrays:

- switch rows and cols to process reverse
DAG Transitive closure

Compute TC row vectors (in postorder) during reverse TS

Good news: can skip down edges
Bad news: there may not be any down edges
void TCdfsR(Dag D, int w, int v)
{
    int u, i;
    pre[v] = cnt0++;
    for (u = 0; u < D->V; u++)
        if (D->adj[v][u])
            {
            D->tc[v][u] = 1;
            if (pre[u] > pre[v]) continue;
            if (pre[u] == -1) TCdfsR(D, v, u);
            for (i = 0; i < D->V; i++)
                if (D->tc[u][i] == 1)
                    D->tc[v][i] = 1;
            
            
    }
}

worst-case cost bound: VE (no help!)
actual cost is V(V+ no. of down edges)
V^2 algorithm? lower bound?
Progress report on reachability ADT

Classical TC algs (Warshall) give
- query: $O(1)$
- preprocessing: $O(V^3)$
- space: $O(V^2)$

Reducing preprocessing to $O(VE)$ is easy DFS application

NO PROGRESS on reducing space to $O(V)$

NO PROGRESS on better guarantees EVEN FOR DAGs (!!!)

Next attempt:
- Is the STRONG reachability problem easier??
**Strong components**

**STRONG COMPONENTS**: mutually reachable vertices

![Graph of strong components](image)

<table>
<thead>
<tr>
<th>sc</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**KERNEL DAG**

- reachability among strong components
- collapse each strong component to a single vertex
Kosaraju’s SC algorithm

- Run DFS on reverse digraph
- Run DFS on digraph, using reverse postorder from first DFS to seek unvisited vertices at top level

**THM:**
- Trees in (second) DFS forest are strong components

\[18.18\]
Kosaraju’s algorithm implementation

Add vertex-indexed array sc to graph representation

Use standard recursive DFS, with postorder numbering

```c
void SCdfsR(Graph G, int w)
{
    link t;
    G->sc[w] = cnt1;
    for (t = G->adj[w]; t != NULL; t = t->next)
        if (G->sc[t->v] == -1) SCdfsR(G, t->v);
    post[cnt0++] = w;
}
```

ADT function for constant-time strong reach queries

```c
int GRAPHstrongreach(Graph G, int s, int t)
{ return G->sc[s] == G->sc[t]; }
```
int GRAPHsc(Graph G)
{
    int i, v; Graph R;
    R = GRAPHreverse(G);
    cnt0 = 0; cnt1 = 0;
    for (v = 0; v < G->V; v++) R->sc[v] = -1;
    for (v = 0; v < G->V; v++)
        if (R->sc[v] == -1) SCdfsR(R, v);
    cnt0 = 0; cnt1 = 0;
    for (v = 0; v < G->V; v++)
        G->sc[v] = -1;
    for (v = 0; v < G->V; v++)
        postR[v] = post[v];
    for (i = G->V-1; i >=0; i--)
        if (G->sc[postR[i]] == -1)
            { SCdfsR(G, postR[i]); cnt1++; }
    return cnt1;
}
Fast abstract transitive closure

1. Find strong components and build kernel DAG
2. Compute TC of kernel DAG
3. Reachability query:
   IF in same strong component, YES
   ELSE check reachability in kernel DAG

Running time depends on graph structure
- density (fast if sparse)
- size of kernel DAG (fast if small)
- cross edges in kernel DAG (fast if few)

Meets performance goals for many graphs

Huge sparse DAG? STILL OPEN
Fast transitive closure implementation

Testimony to benefits of careful ADT design

Dag K;

void GRAPHtc(Graph G)
{
  int v, w; link t; int *sc = G->sc;
  K = DAGinit(GRAPHsc(G));
  for (v = 0; v < G->V; v++)
    for (t = G->adj[v]; t != NULL; t = t->next)
      DAGinsertE(K, dagEDGE(sc[v], sc[t->v]));
  DAGtc(K);
}

int GRAPHreach(Graph G, int s, int t)
{
  return DAGreach(K, G->sc[s], G->sc[t]);
}