Analysis of DFS, BFS, and apps that incorporate them

Note Title 4/5/2012

$G = \langle V, E \rangle$ – directed graph

$E \subseteq V \times V$

$|E| = m \qquad |V| = n$

$m \leq n \cdot n = n^2 \qquad m \leq n^2$

DFS and BFS visit each vertex at most once, and try to traverse each edge at most once.

However, this **not** necessarily means that they run in $\Theta(n+m) = \Theta(\max(n,m))$ worst-case time.

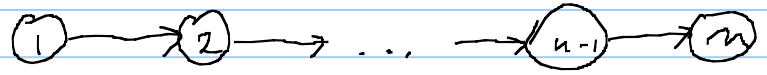
The running times of DFS and BFS depend both on the graphs and their implementations.

1. Adjacency matrix implementation.

For each vertex v of G, entire row of its adjacency matrix must be examined in order to find all vertices adjacent to V. This requires $\Theta(n)$ steps (since the adjacency matrix has $n \times n$ size).

Repeating the above for each vertex V of G (there are n of them) will result in $\Theta(n \times n) = \Theta(n^2)$ worst-case running time.

Example of worst-case graph:



2. Adjacency lists implementation

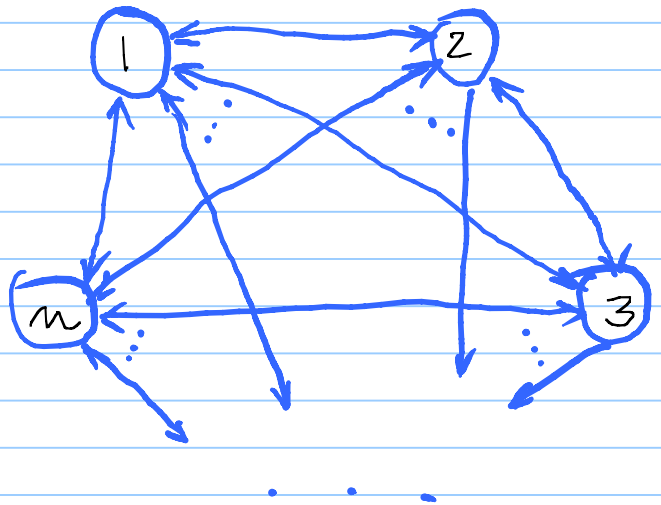
For each vertex v of G, every vertex w adjacent to v must be examined. This will require examination of each edge that has v as its source. It will be done once for each edge, each in $O(1)$ time.

Repeating the above for each vertex v of G will require up to n visits (for each vertex v there is at most one visit, each done in $O(1)$ time) **plus** examination of every edge of G (each of them once, each done in $O(1)$ time).

Hence, DFS and BFS will result in

$\Theta(n+m) = \Theta(\max(n, m))$ worst-case running time.

Here is an example worst-case digraph:



(every vertex has an edge to every other vertex)

 is an abbreviation of 

There is n vertices and $4(n-1)$ edges to examine here.

Special cases.

a. G is any graph on n vertices

In this case, the worst-case number of edges is $n(n-1) \in \Theta(n^2)$

So, the worst-case running times for DFS and BFS are $\Theta(\max(n, n(n-1))) = \Theta(n(n-1)) = \Theta(n^2)$.

b. G has limited number of edges by $m \leq c \cdot n$ (where $c \geq 1$ is constant for all such G's)

In this case, the worst-case running times for DFS and BFS are $\Theta(\max(n, c \cdot n)) = \Theta(c \cdot n) = \Theta(n)$.

Space usage

1. Adjacency matrix implementation $|adjMx(n)|$

For any graph on n vertices, its adjacency matrix has $n \times n$ elements, each of them requiring at least 1 bit. All other space either takes up to $c \cdot n$ bits (where c is a constant) or d bits (where d is a constant). This results in $\Theta(n^2)$ space used.

$|adjMx(n)| \in \Theta(n^2)$

The size (say, in bytes) of adjacency matrix implementation of a digraph on n vertices.

2. Adjacency lists implementation.

Headers of adjacency lists (one for each of n vertices of G) take $\Theta(n)$ space.

The lists themselves (one entry for each edge of G) take $\Theta(m)$ space. Other (auxiliary) variables take $\Theta(1)$ space.

This results in

$\Theta(n + m + 1) = \Theta(\max(n, m))$ space used.

$|\text{adjLISTs}(n)| \in \Theta(\max(n, m))$

The size of adjacency LISTs implementation of digraph on n vertices.

So, for digraphs G that satisfy a limitation

$$m(G) \in o\left(n^2(G)\right), \quad \left|adjLists(n)\right| \in o\left(\left|adjMx(n)\right|\right)$$

for instance, if $m(G) \in O(n(G))$ or $m(G) \in O(n \lg n(G))$

that is,

$$\lim_{n\to\infty} \frac{\left|adjLists(n)\right|}{\left|adjMx(n)\right|} = 0 \text{ or } \lim_{n\to\infty} \frac{\left|adjMx(n)\right|}{\left|adjLists(n)\right|} = \infty$$

This shows how much more memory is needed for adjacency matrix implementations than for adjacency lists implementation.

Therefore, adjacency matrix implementation can only be space-efficient if and only if $m(G) \notin o(n^2(G))$, that is, iff

$$m(G) \in \Theta(n^2(G)).$$

Graphs that do <u>not</u> belong to that category include acyclic graphs, because an acyclic graph on n vertices can have at most $n-1$ edges, and $n-1 \in o(n^2)$.

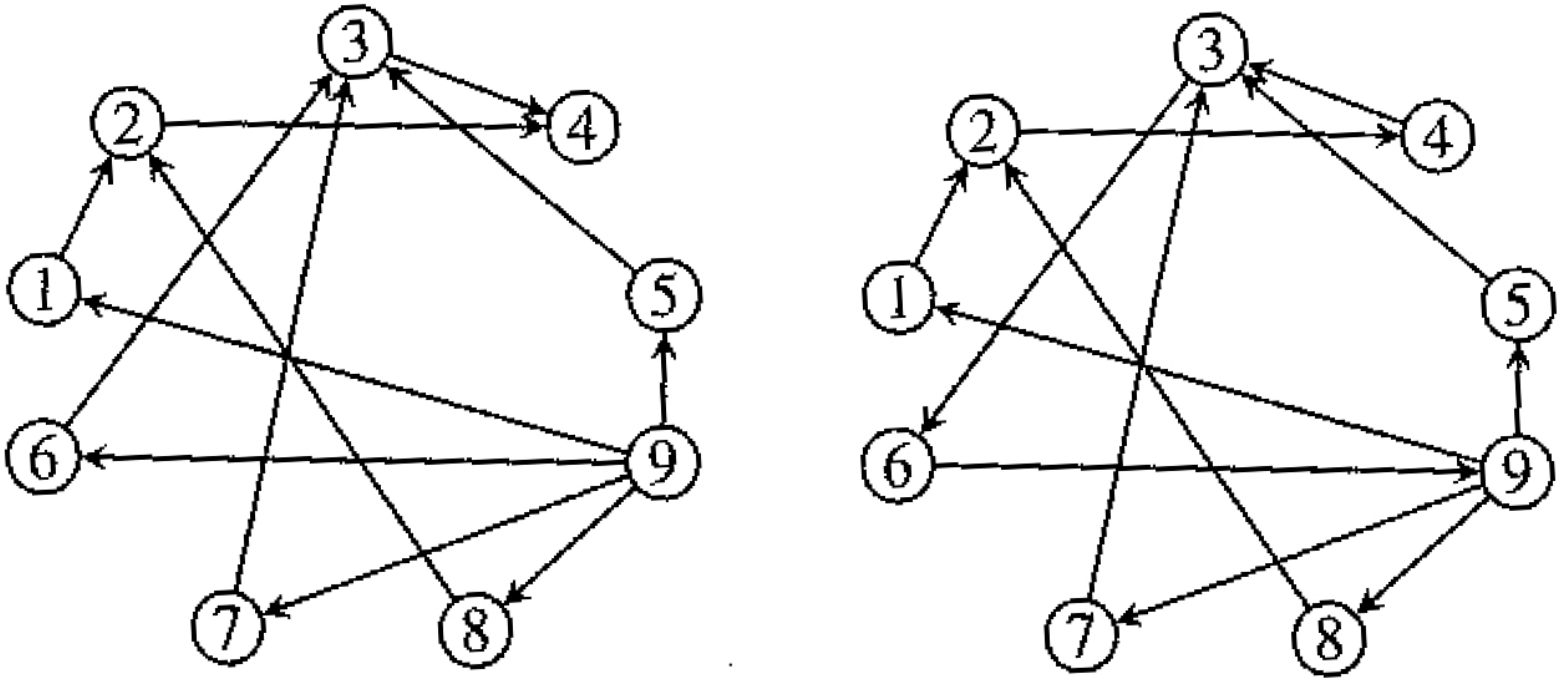


Figure 7.18 Two directed graphs. Which one is acyclic?

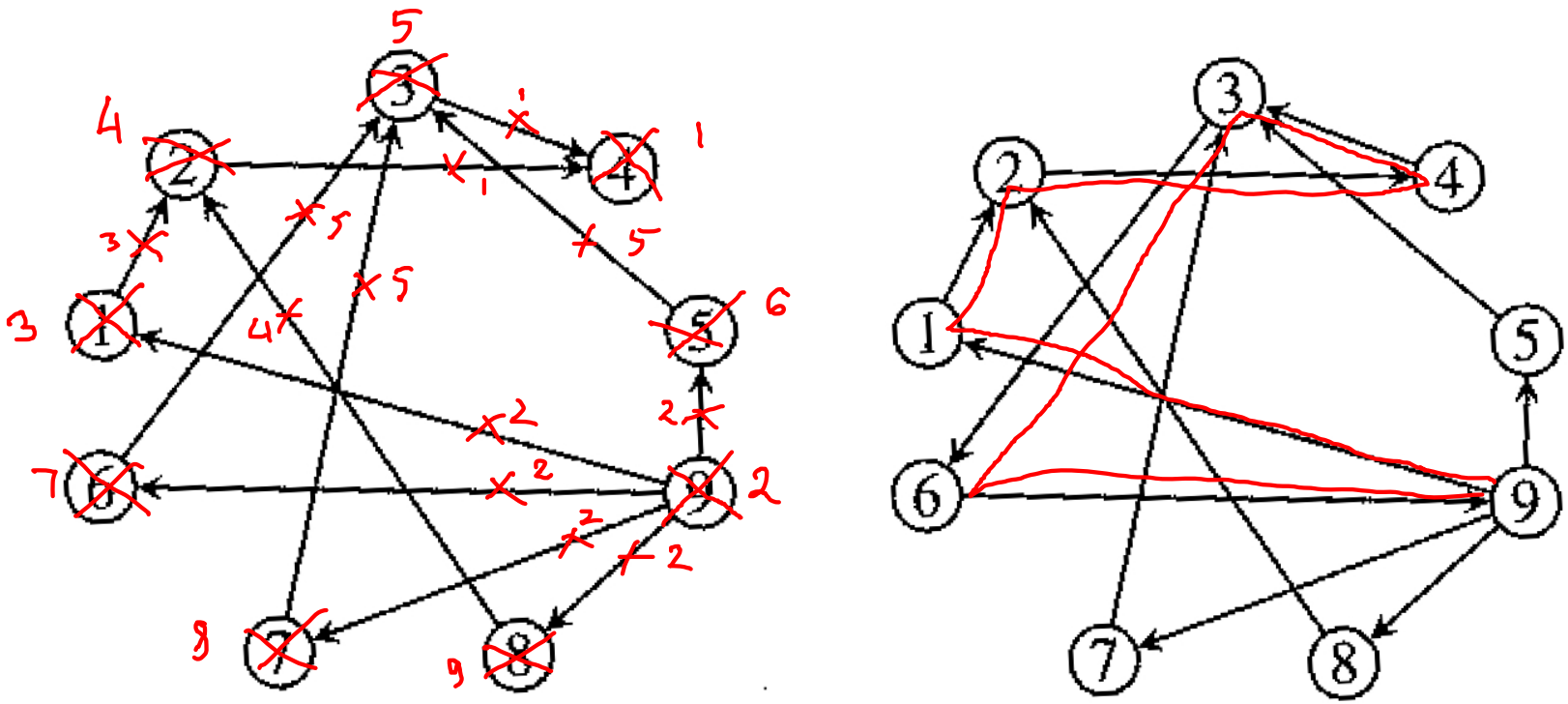


Figure 7.18 Two directed graphs. Which one is acyclic?

The naïve acyclicity test runs in $\Theta(n^2)$ worst-case running time.

DFS acyclicity test runs in the same time as DFS does, that is

$$\Theta(\max(n, m))$$

digraph

$G = (V, E)$ ← binary relation

Transitive closure E^c of E

For each $v, w \in V$,

$(v, w) \in E^c$ iff there is a path from v to w in G

E^c is a transitive binary relation

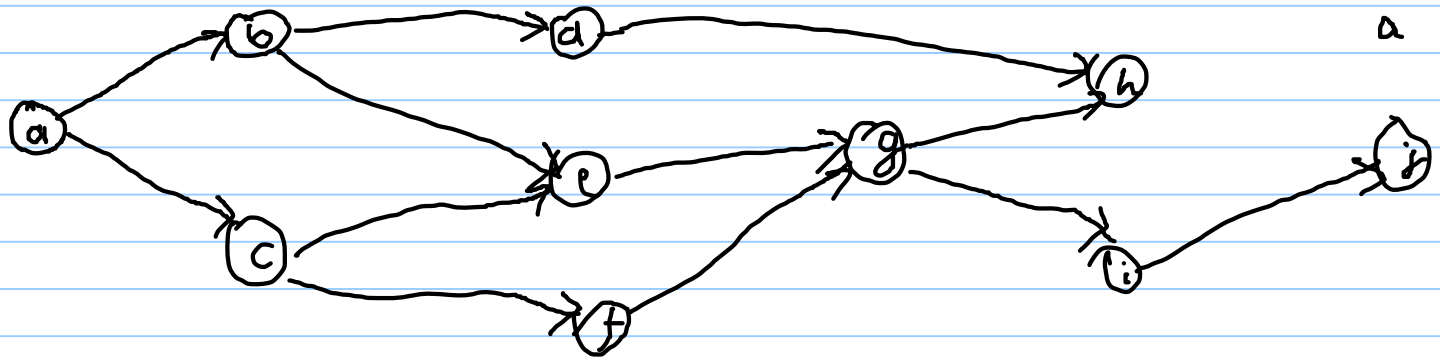
$(v, w) \in E^c$ & $(w, t) \in E^c \Rightarrow (v, t) \in E^c$ — (

Moreover, E^c is a reflexive relation

For each $v \in V$, $(v,v) \in E^c$

DAG – directed acyclic graph

– acyclic digraph



a DAG

If G is a DAG then

E^c is antisymmetric binary relation

For each $v, w \in V$, if $(v,w) \in E^c$ & $v \neq w$

$\Rightarrow (w,v) \notin E^c$

Therefore, G is a DAG

iff E^c is a partial ordering

relation.

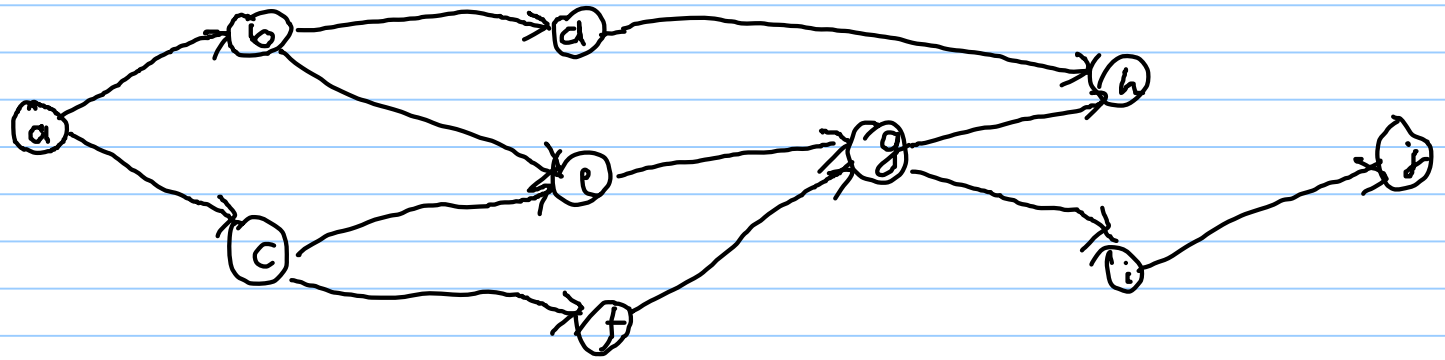
Theorem. Every partial ordering R on a set D extends to a total ordering T on D.
("extends" means $R \subseteq T$). (a.k.a. linear)

(Proof requires material outside of the scope of this course and will not be given here.)

Definition. A relation E^t is called a **topological order** for a digraph $G = \langle V, E \rangle$ iff E^t is a total order that extends E^c.

For finite digraphs, topological order is isomorphic with enumeration $v_1, v_2, \ldots v_n$ of all vertices of G such that there is no edge (v_i, v_j) in G such that $i > j$.

Example. E is a prerequisite relation and topological ordering is a valid plan of study with one course per semester.



Example of topological ordering

$a < c < f < b < e < g < i < j < d < h$

Theorem. A digraph $G = \langle V, E \rangle$ has a topological ordering E^t iff G is a DAG (iff G is acyclic).

Proof follows from the preceding discussion.

Construction of topological ordering:

Execute DFS (or BFS) and enumerate vertices of the digraph in the order they were visited.

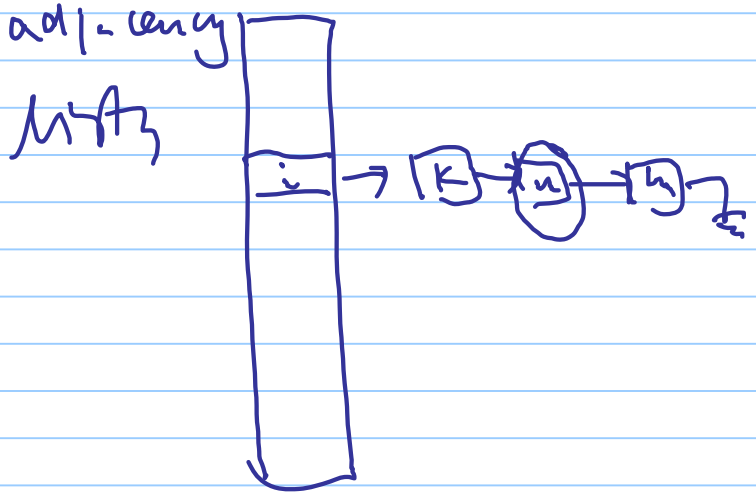
This can be done in $\Theta(\max(n, m))$ time if an adjacency LISTS implementation of G is used.

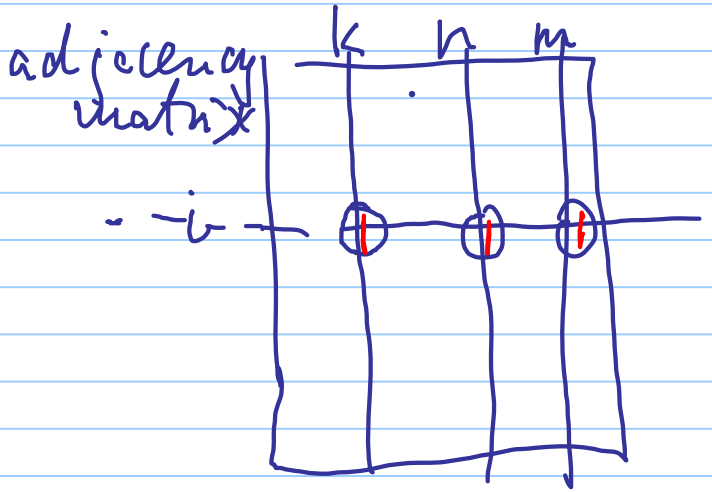
Fact. The inverse of topological ordering is a linear ordering, too.

It can be found without reversing edges of G.

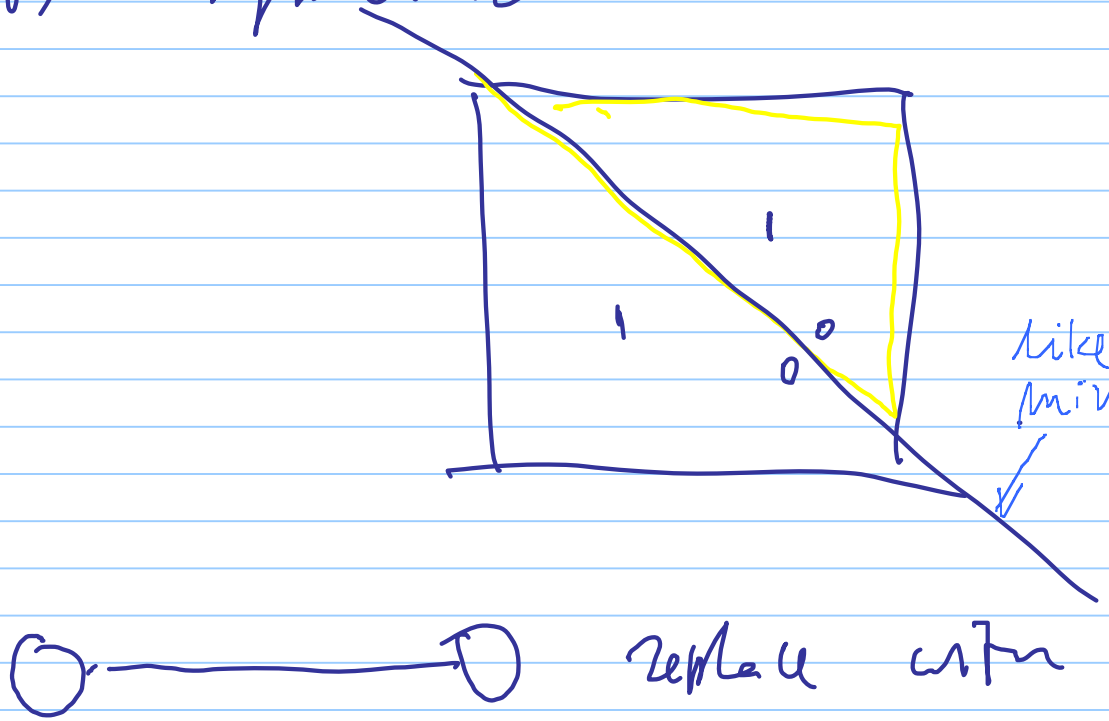
Converting an (undirected) graph G onto "equivalent" digraph G'

1. Produce adjacency matrix of G.
   if G is represented as adjacency lists.

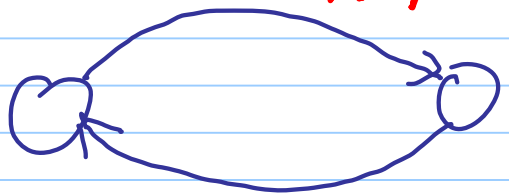




Make sure the adjacency matrix is symmetric



Not a cycle in undirected graph



replace with

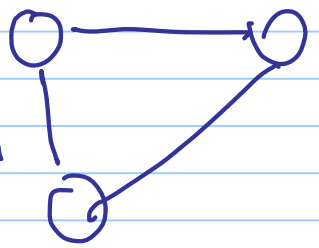
3. Convert adjacency matrix onto adjacency lists

Let G be an (undirected) graph.

there are some typical questions and problems related to G:

1. Is G cyclic or acyclic?

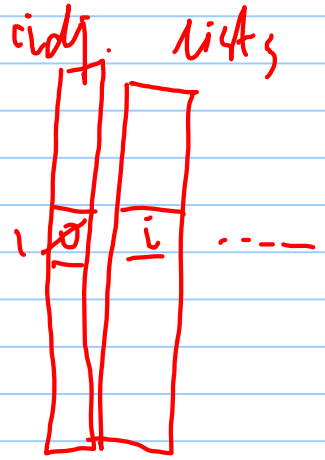
a shortest cycle in a graph



Not a cycle!

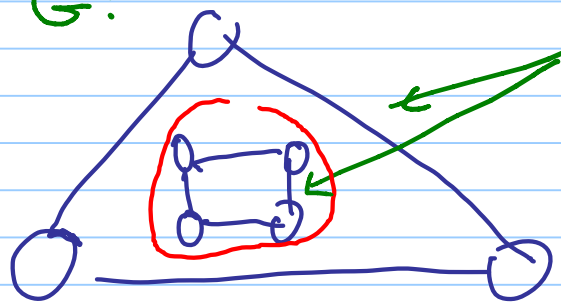
2. Is G connected or disconnected

3. Find all connected components of G



↑ "visited" marks (one for each vertex, may be either true or false).

G:



These are 2 connected components of G

Running DFS(G, v) on graph G and its vertex v will produce a list of all vertices that belong to the connected component of G that contains v. Also, it will mark all such vertices "visited".

Running DFS(G, w) for any unvisited (so far) vertex w of G, if such w exists, will produce a list of all vertices that belong to a second connected component of G.

The following algorithm will list all vertices of graph G in groups: one group of vertices for one connected component of G.

See DFS posted on class website.

Optional

Mr. Sum and Mr. Product Puzzle

Two Mathematicians: Mr. Sum and Mr. Product

Two natural numbers: $2 \leq a \leq b \leq 100$

Mr. Sum knows the value of $a + b$

Mr. Product knows the value of $a \times b$

That's **all** they know.

This is their common knowledge

The following dialogue takes place.

P: "I don't know the value of a and b".

S: "I knew you wouldn't. I don't know them, either".

P: "But now, I do know the values of a and b".

S: "Now, I know them, too".

What are these values?

$$C\varphi \overset{\text{def}}{\equiv}$$

$$\varphi \wedge K_S\varphi \wedge K_P\varphi \wedge$$

$$K_S K_P \varphi \wedge K_P K_S \varphi \wedge$$

$$K_S K_P K_S \varphi \wedge K_P K_S K_P \varphi \wedge$$

... etc ... an infinite conjunction of statements like these.

Optional

Because every model $\mathcal{M}$ is a subset of the Cartesian product I^N, we may represent it as a mesh, e.g., (for $N = 2$ and $I = \{0, 1, 2, 3, 4, 5\}$):

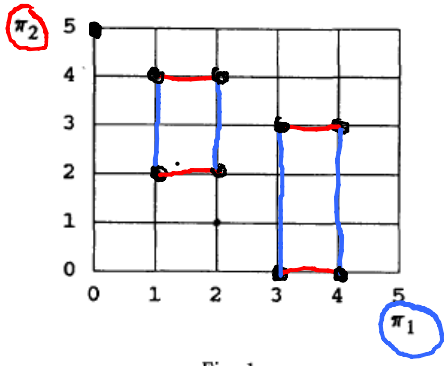


Fig. 1.

where joints • represent possible worlds. The above mesh visualises model $\mathcal{M} = \{[0, 5], [1, 2], [1, 4], [2, 1], [2, 2], [3, 0], [3, 3], [4, 0], [4, 3]\}$. If the actual world is [2,1] then A_1 knows that $(\pi_1 = 2) \wedge ((\pi_2 = 1) \vee (\pi_2 = 2))$ and A_2 knows that $(\pi_1 = 2) \wedge (\pi_2 = 1)$, i.e. $\mathcal{M} \models K_1((\pi_1 = 2) \wedge ((\pi_2 = 1) \vee (\pi_2 = 2)))[2, 1]$ and $\mathcal{M} \models K_2((\pi_1 = 2) \wedge (\pi_2 = 1))[2, 1]$. Consequently, if the actual world is [2,2], A_2 does not know $(\pi_1 = 2)$, but also A_1 does not know that A_1 does not know that $(\pi_1 = 2)$, i.e., $\mathcal{M} \models \neg K_2(\pi_1 = 2)[2, 2]$ and $\mathcal{M} \models \neg K_1 \neg K_2(\pi_1 = 2)[2, 2]$.

By connecting those joints of this mesh which lie on the same coordinate line ($\pi_1 = const$ or $\pi_2 = const$) one obtains a graph whose connected components provide the semantics for the common knowledge modality C. Namely, $\mathcal{M} \models C\varphi[\vec{i}]$ iff for each $\vec{j} \in \mathcal{M}$ which belongs to the same connected component as $\vec{i}$ does, $\mathcal{M} \models \varphi[\vec{j}]$. For $\mathcal{M}$ of Fig. 1, $\mathcal{M} \models C(\pi_1 = 1 \vee \pi_1 = 2)[1, 4]$, but $\mathcal{M} \not\models C(\pi_1 = 1)[1, 4]$.

$\mathcal{M} \models C(((\pi_1 = 1) \vee (\pi_1 = 2)) \wedge ((\pi_2 = 2) \vee (\pi_2 = 4)))[1,2]$

although $\mathcal{M} \models K_1(\pi_1 = 1) \wedge K_2(\pi_2 = 2)\ [1,2]$

Optional

Example for Sum and Product.

If Sum knows 8 then Sum knows that Product knows 12, 15, or 16, therefore Sum knows that Product knows that Sum knows 8, 7, 8, 10 (= 7, 8 or 10) therefore, Sum knows that Product knows that Sum knows that Product knows

10, 12, 12, 15, 16, 16, 21, 24, 25

(= 10, 12, 15, 16, 21, 24 or 25) ... etc

It turns out that respective connected component is ∞!
(infinite)

Optional

So, there is very little common knowledge (between Sum and Product) except for that stated in the puzzle!

1'. A version of the acyclicity test. for undirected graphs

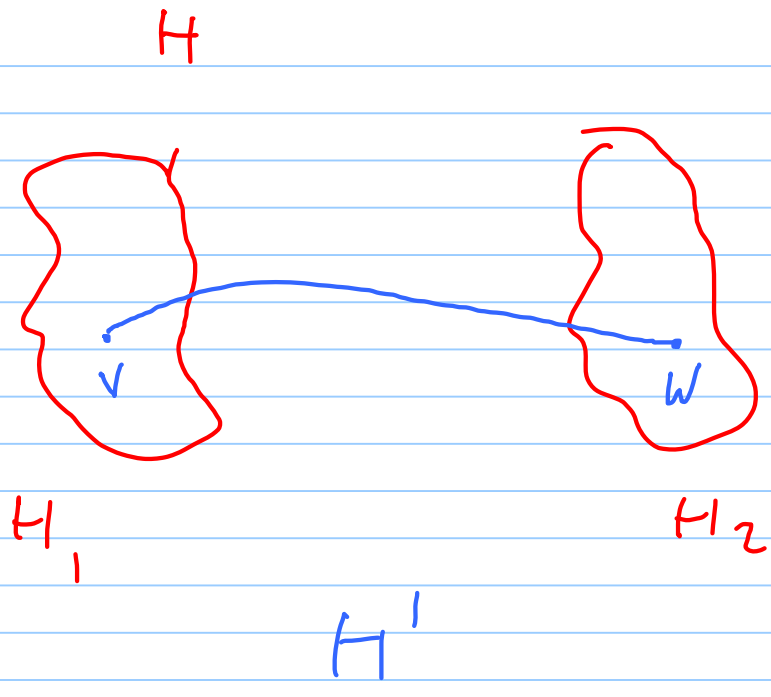
a. Find all connected components of G, counting the number of vertices n_i and the number of edges m_i in each connected component i of G.

b. Verify that for each connected component i of G, $n_i = m_i + 1$.

4. Contraction of a maximal acyclic Supergraph H of an acyclic graph G.

a. $H = G$

b. While H is not connected
{ select vertices v and w from G that belong to different connected components of H;
$E_H = E_H \cup \{\{v, w\}\}$ // E_H is the set of edges of H }



$$E_{H'} = E_H \cup \{\{v, w\}\}$$

How do we know H' is acyclic?
H_1 and H_2 are acyclic and connected.
H' is connected, too.

Let n_1 be the number of nodes of H_1,

m_1 be the number of edges of H_1,

n_2 be the number of nodes of H_2, and

m_2 be the number of edges of H_2.

By tree properties:

$m_1 = n_1 - 1$ and $m_2 = n_2 - 1$. So,

$m_1 + m_2 = n_1 + n_2 - 2$.

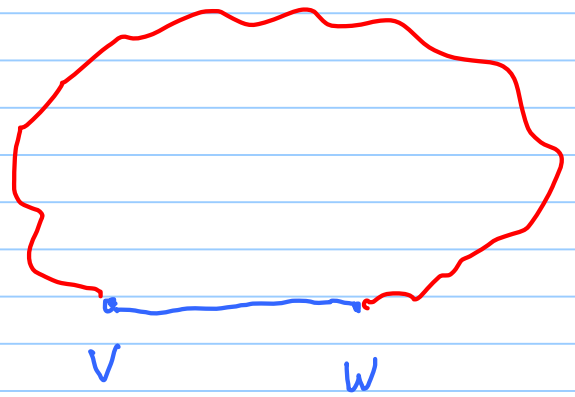
Thus H' has $m_1 + m_2 + 1 = n_1 + n_2 - 1$ edges, so H' is a tree.

Here is another proof.

Suppose that H' is cyclic

Then there must be a cycle in H' that contains edge $\{v, w\}$

It means that there is a path P from v to w in H'



P does not traverse edge $\{v, w\}$, so

P must be a path in H.
So v and w must belong to the same connected component of H. This contradicts the choice of v and w from different connected components of H.
So, our assertion that H' has a cycle leads to a contradiction.
Therefore, H' is acyclic.

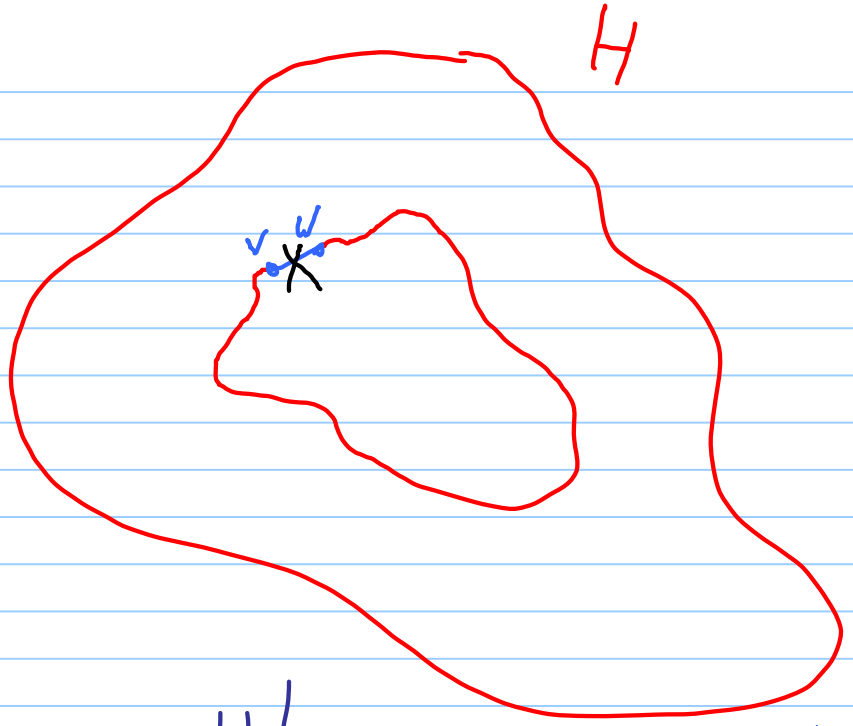
Conclusion. After exiting the while loop of the above algorithm, H is a maximal acyclic supergraph of G. H is connected because otherwise the said loop would continue.

Note. One can modify the above algorithm to assume that H is a subgraph of a connected graph G' that is a supergraph of G and has the same set of vertices as G has.

5. Construction of minimal connected subgraph H of a connected graph G.
(a costly one - we will see a better one, later)

a. $H = G$

b. While H has a cycle // can use DFS to find out
{ select vertices v and w from G
such that the edge $\{v, w\}$ is a
part of a cycle in H;
$E_H = E_H \setminus \{\{v, w\}\}$ // remove the
edge (v, w) from the
set of edges of H }



H' ← set-theoretic difference (between sets)

suppose that $H' = H - \{\{v, w\}\}$ is not connected.

If so then v and w must belong to different connected components of H'

But, there is still a path from v to w. All the edges of that path are in H', so v and w belong to the same connected component of H'.

Contradiction. Hence, H' is connected.

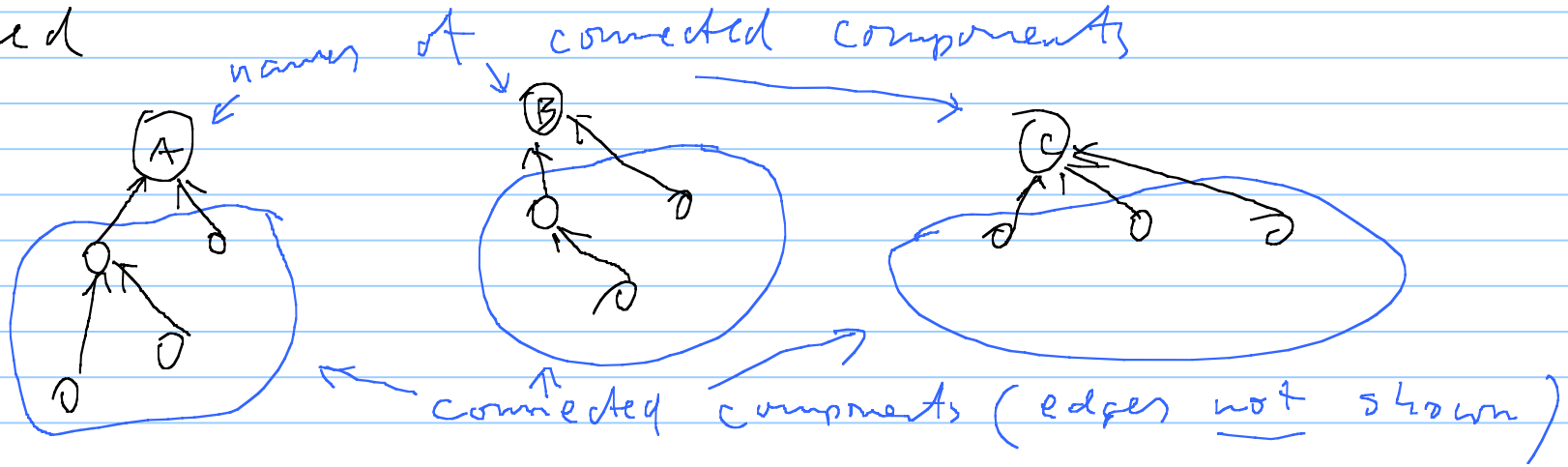
Conclusion. After exiting the while-loop in the above algorithm, graph H is a connected acyclic graph. Removing any edge from H would result in breaking H

onto two connected components. (An argument used in the proof of the previous algorithm proves it. Exercise: Prove that removing any edge from H would make H a disconnected (not connected, that is) graph.)

So, H is a minimal connected subgraph of G that contains all vertices of G.

In order to make the above algorithms efficient, special data structure may be needed.

For instance, a tree of connected components (a.k.a. **find-and-merge set**) may be used



Test 2 coverage
ends here.

Note regarding acyclic digraph. (DAG)

Since a digraph G is acyclic iff there exists topological ordering of its vertices, it follows that the maximum number of edges in a DAG on n vertices is $\frac{n(n-1)}{2}$

Proof. Let G be a DAG on n vertices. Let $v_1, v_2, \ldots, v_n$ be a topological ordering of vertices of G. Because topological ordering is a linear ordering that extends E^c, for every edge $(v_k, v_m) \in E$, $k < m$. Because there are $\frac{n(n-1)}{2}$ pairs of the form (k, m), where $1 \le k < m \le n$ (prove it!),

the number of edges of G is not larger than

$$\frac{n(n-1)}{2}.$$

On the other hand, adding to the set of edges E of graph G all missing pairs of the form (V_k, V_m) where $1 \le k < m \le n$ will result in a DAG G' (prove it!) with the same vertices as G and a set of edges $E' \supseteq E$. So, G' is an acyclic supergraph of G and has exactly $\frac{n(n-1)}{2}$ edges. This way we have proved the following

Theorem. Every DAG G can be extended, by means of adding 0 or more edges, to a DAG G' that has $\frac{n(n-1)}{2}$ edges. Moreover, any digraph H on n vertices that has more than $\frac{n(n-1)}{2}$ edges has a cycle.

Any such digraph G' is called a maximal DAG on n vertices.