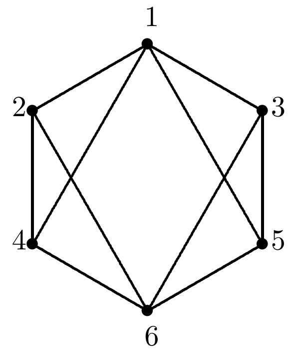
informally, a graph consists of "vertices" joined together by "edges," e.g.,:

example graph G_0:



formally a graph is a pair (V, E) where
 V is a finite set of elements, called *vertices*
 E is a finite set of pairs of vertices, called *edges*
if H is a graph, we can denote its vertex & edge sets as $V(H)$ & $E(H)$ respectively

if the pairs of E are unordered, the graph is *undirected*
if the pairs of E are ordered the graph is *directed*, or a *digraph*

two vertices joined by an edge are *adjacent*, also *neighbors*

Size of a Graph

n always denotes the number of vertices of the graph, i.e., $n = |V|$
m always denotes the number of edges, $m = |E|$
 G_0 has $n = 6$, $m = 10$

a graph is *complete* if it has every possible edge, so $m = n(n-1)/2$ if it's undirected

an *isolated vertex* is not on any edge
we usually assume there are no isolated vertices
 since in most applications isolated vertices are trivial
in this case $m \geq n/2$

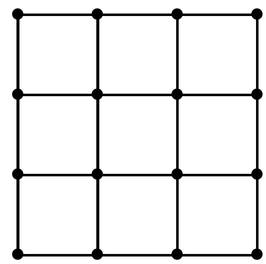
thus in general, $\underline{m = O(n^2)}$, and in most applications, $\underline{m = \Omega(n)}$
 these bounds hold for digraphs too

sometime's we're sloppy and assume $m = \Omega(n)$, eg, we write $O(m)$ rather than $O(m+n)$

these bounds can also be seen from the
Handshaking Lemma. *In an undirected graph the degrees sum to* $2m$.

graphs with $\Theta(n)$ edges are called *sparse*, those with $\Theta(n^2)$ edges are *dense*

the *grid graph* is sparse:



Connectivity

$G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ & $E' \subseteq E$

a *path* is a sequence of vertices $v_0, v_1, \ldots, v_\ell$, $\ell \geq 0$, with $(v_i, v_{i+1}) \in E$ for $i = 0, \ldots, \ell - 1$
it's *simple* if all vertices are distinct
a path can have length 0

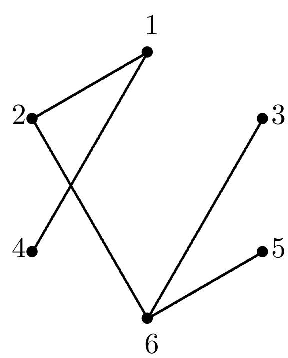
a *cycle* is a path with $\ell \geq 3$, $v_0 = v_\ell$ and all other vertices & edges distinct

see CLRS B.4 for other basic terms like *degree*
note the above definitions differ slightly from CLRS

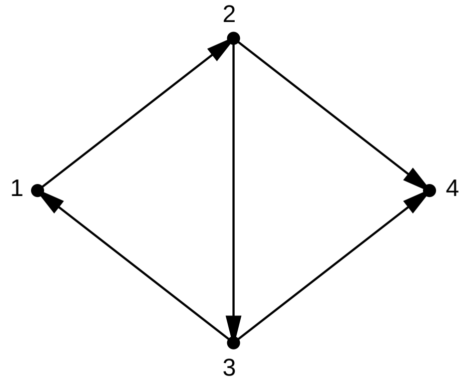
an undirected graph is *connected* if it has a path joining any 2 vertices
a *tree* is a connected undirected graph with no cycles

a connected undirected graph G has a *spanning tree*, i.e.,
a subgraph that is a tree containing every vertex of G

spanning tree T of G_0:



Digraphs



CLRS allows *loops* (x, x) in digraphs but not in undirected graphs

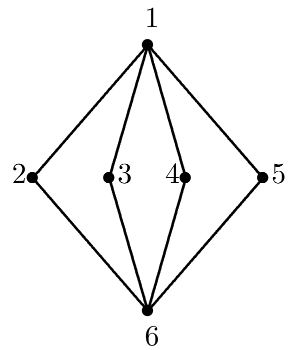
an edge (u, v) goes *from* u *to* v
v is the *head* & u is the *tail*

vertex v is *reachable* from vertex u if there is a path from u to v
vertex 1 can reach all others, but vertex 4 can reach only itself

Graph Operations

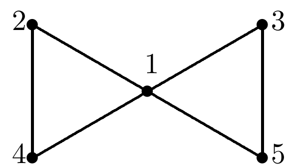
deleting edge e from graph G means forming the graph $\underline{G - e}$
having all edges of G except e

$G_0 - \{(2,4),(3,5)\}$:



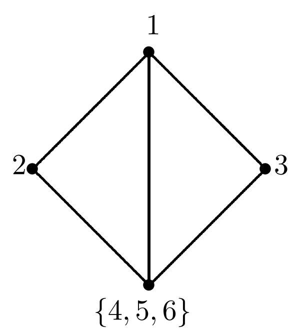
deleting vertex v from G means forming the graph $\underline{G - v}$
having all vertices of G except v and all edges of G except those incident to v

$G_0 - 6$ is the "bowtie graph":

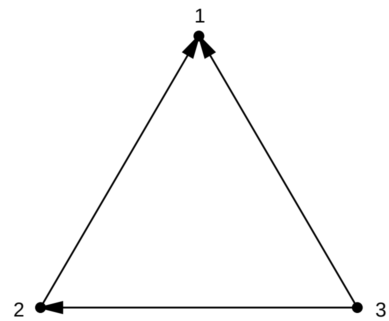


contracting a set of vertices S means forming the graph $\underline{G/S}$
where the vertices S are replaced by a new vertex Σ, adjacent to every neighbor of S

$G_0/\{4,5,6\}$:

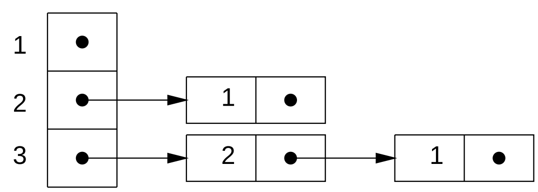


the first step of any graph algorithm is to read the graph into a graph data structure
 the input graph is usually presented as a list of edges, with the vertices numbered from 1 to n



Example digraph G_0

1. Adjacency lists



Adjacency list representation of G_0

the *adjacency list* of a vertex v is a list of all vertices w with $(v, w) \in E$

the *adjacency list representation* for a graph (directed or undirected)
 consists of an adjacency list for every vertex
 plus an array of list heads

the adjacency list representation uses space $\Theta(m + n)$

Variations
adjacency lists are sometimes doubly-linked, to facilitate deletions
in undirected graphs the 2 nodes for an edge may be linked to each other

An implementation of adjacency lists for static graphs
2 parallel arrays `LINK` & `VERTEX` give pointer and vertex information, respectively

	LINK	VERTEX
1	0	
2	4	
3	5	
4	0	1
5	6	2
6	0	1

Parallel arrays representing G_0.

in general:
for $1 \le i \le n$, $\texttt{LINK}[i]$ is the head of i's adjacency list,
for $n+1 \le i$, $\texttt{LINK}[i]$ & $\texttt{VERTEX}[i]$ form a node on an adjacency list –
  $\texttt{LINK}[i]$ points to next node, $\texttt{VERTEX}[i]$ gives the vertex
  $\texttt{LINK}[i] = 0$ if the node at i is the last on its adjacency list
for digraphs $i \le m+n$; for undirected graphs $i \le 2m+n$

Remark. For some applications we can use sequentially allocated adjacency lists.

Problem. Calculate the out-degree of each vertex in a digraph.
The digraph is given by an adjacency list representation.

Solution 1: Low-level algorithm
  /$*$ this code sets $d[v]$ to the out-degree of v, for each vertex v $*$/
  for $v \leftarrow 1$ **to** n **do** {
    $d[v] \leftarrow 0$;
    $i \leftarrow \texttt{LINK}[v]$;
    while $i \ne 0$ **do** {
      increase $d[v]$ by 1;
      $i \leftarrow \texttt{LINK}[i]$; } }

this code calculates all out-degrees in time $\Theta(m+n)$
  the **for** loop iterates n times
  the body of the **while** loop is executed once for each edge

Solution 2: High-level algorithm
we omit the details of pointer manipulation in walking down adjacency lists

  for $v \in V$ **do** {
    $d[v] \leftarrow 0$;
    for each edge (v, w) **do**
      increase $d[v]$ by 1; }

the inner **for** loop walks down v's adjacency list, as in Solution 1

the time for this algorithm is $\Theta(m+n)$

> *Timing Principle*: A graph algorithm uses time $O(m+n)$ if it does $O(1)$ work on each vertex or edge.

Important Special Case:
an algorithm that walks down every adjacency list uses linear time
if the remaining work can be "charged" to work by the walk

Exercises.
1. Criticize this reasoning: In Solution 2 the loop
for each edge (v, w) **do**
iterates $O(n)$ times for a given vertex v. There are n vertices. Hence the total time is $O(n^2)$.

2. Criticize this pseudocode to calculate the in-degree of each vertex.
for $v \in V$ **do** {
$d[v] \leftarrow 0$;
for each edge (w, v) **do**
increase $d[v]$ by 1; }

2. Adjacency matrices

	1	2	3
1	0	0	0
2	1	0	0
3	1	1	0

Adjacency matrix representation of G_0

an *adjacency matrix* is an $n \times n$ matrix with
$A[x, y] = 1$ if (x, y) is an edge, else $A[x, y] = 0$

an adjacency matrix has size $\Theta(n^2)$

to calculate all out-degrees for a digraph represented as an adjacency matrix:
for $v \leftarrow 1$ **to** n **do** {
$d[v] \leftarrow 0$;
for $w \leftarrow 1$ **to** n **do** $d[v] \leftarrow d[v] + A[v, w]$; }

this algorithm takes time $\Theta(n^2)$

Conclusion.
for sparse graphs, adjacency list representations are preferable to adjacency matrices:
they use less space, & (consequently) lead to faster algorithms
e.g., the bound $\Theta(m+n)$ is superior to $\Theta(n^2)$, since it improves on sparser graphs

Remark. Ex.22.1-6 gives what's essentially the only nontrivial problem that can be solved in $o(n^2)$ time using adjacency matrices.

many basic algorithms for graphs use dfs
e.g., time $O(m+n)$ algorithms for

connected & biconnected components of an undirected graph
strong components of a digraph
} Tarjan, *SICOMP* '72

planarity testing (Hopcroft & Tarjan, *J. ACM* '74)
triconnected components of an undirected graph (Hopcroft & Tarjan, *SICOMP* '73)
approximating smallest well-connected subgraphs (Khuller & Vishkin, *J. ACM* '94,...)

throughout this handout $G = (V, E)$ is a graph,
either <u>undirected</u> or <u>directed</u>

a *search* of a graph "scans" all the edges (e.g., breadth-first search, CLRS 22.2)

idea of dfs:
repeatedly scan an edge from
the most recently discovered vertex with unscanned edges

recursive implementation of dfs:

procedure `DFS`(v) {D:

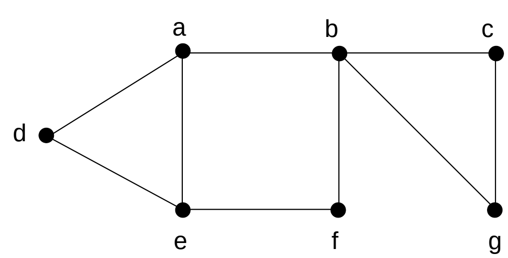
for each edge (v, w) **do**
{S:
if w has not been discovered **then** `DFS`(w);
S$'$:}

F:}

v is *discovered* at point D
v is *finished* at point F
(v, w) is *scanned* at point S (& perhaps S$'$)

the main routine starts the search by calling `DFS`(s) for an arbitrary vertex s

we illustrate, showing 2 ways to conceptualize dfs

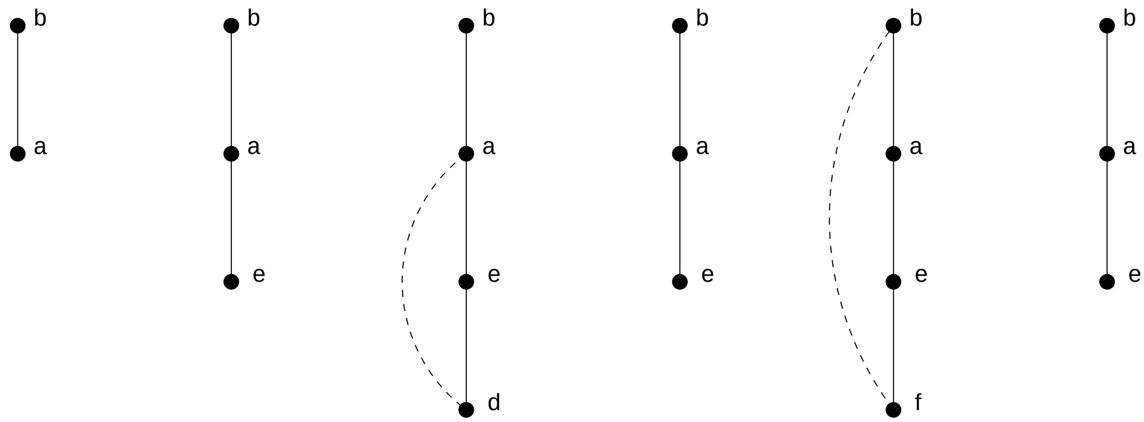


Example graph G_0

1. Path view of dfs

a *dfs path* is the path of edges the search traverses to discover a vertex v

"dfs grows a sequence of dfs paths"



The first 6 paths in `DFS`(b).
Path edges are drawn solid.

at any point in time, the sequence of vertices in the dfs path corresponds to
the vertices in the recursion stack of `DFS`

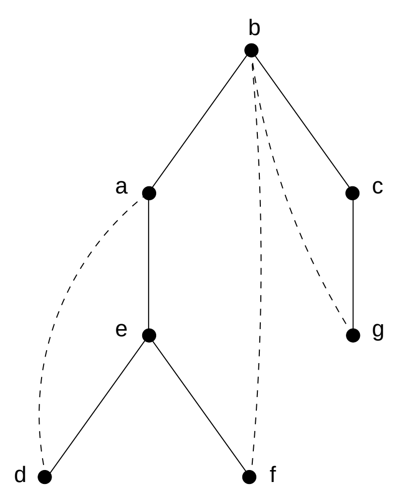
2. Tree view of dfs

all the dfs paths together can be represented by a tree

the dfs tree consists of every edge that leads to the discovery of a vertex

the children of a node are discovered in left-to-right order

"dfs constructs a dfs tree"



Dfs tree for `DFS`(b).
Tree edges are drawn solid.

the dfs tree is the recursion tree of `DFS`

i.e., v is the parent of w if `DFS`(w) is called from `DFS`(v)

Note (a, d) can be scanned from both a and d; (a, e) can be scanned from both a and e.

in many applications, this 2nd scan – from ancestor to descendant – is a `NOOP`

Remarks

1. in general to ensure the search explores the entire graph, the main routine of a dfs is:

> **for** each vertex v **do**
>   **if** v has not been discovered **then** $\texttt{DFS}(v)$

Example.

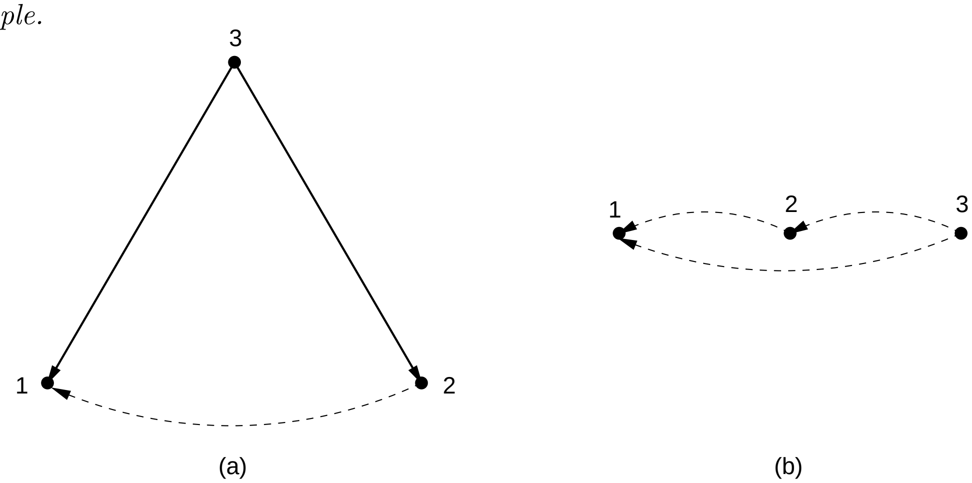


Fig. 1. 2 dfs's of digraph G_0 of Handout#3.
(a) $\texttt{DFS}(3)$ explores the whole graph.
(b) G_0 explored by $\texttt{DFS}(1)$; $\texttt{DFS}(2)$; $\texttt{DFS}(3)$. We get a dfs forest consisting of 3 dfs trees.

2. dfs can be implemented in time $\underline{O(m+n)}$

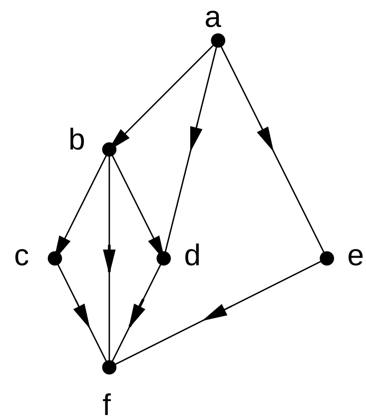
highlights:
use a boolean array $\texttt{discovered}[v]$
each recursive call $\texttt{DFS}(v)$ uses $O(1)$ time to manage the recursion stack
walking through the adjacency structure takes $O(m+n)$ time

3. we can test if an undirected graph is connected in $O(m+n)$ time by dfs
if not connected, we find the connected components
use an array $\texttt{component}[v]$
each tree of the dfs is a connected component

a collection of trees is called a *forest*

we can test if all vertices of a digraph are reachable from s in $O(m+n)$ time by dfs
just check if all vertices are discovered in $\texttt{DFS}(s)$

a *dag* is a directed acyclic graph
a dag can always be drawn so all edges are directed down:



Example dag G_0

dags model combinational circuits, prerequisite graphs, makefile & program dependencies,
arithmetic expressions, spreadsheet evaluation, Bayesian networks, neural networks, . . .

Basic dag concepts

a *source* (*sink*) of a dag is a vertex with in-degree (out-degree) 0
every dag has one or more sources and one or more sinks
since a maximal length path starts at a source & ends at a sink

Example. G_0 has sink f; $G_0 - f$ has sinks c, d, e

Topological sort

a *topological numbering* of a digraph assigns an integer to each vertex so that
each edge is directed from lower number to higher number
usually we use the numbers $1..n$, but this isn't required

Example. dag G_0 has topological order a, b, c, d, e, f
we'll use an array I to record a topological numbering,
so this order would be $I[a] = 1$, . . ., $I[f] = 6$

Topological Order Theorem. *A digraph has a topological numbering* $\Longleftrightarrow$ *it's a dag.*

Proof.
$\Longrightarrow$: topological numbers increase along any path, so we can't have a cycle

$\Longleftarrow$: assign the highest number n to some sink s
then delete s and recursively number all remaining vertices $\square$

the proof suggests the following high level algorithm:

repeat until G has no vertices:
  grow the dfs path P until a sink s is reached
  set $I[s] = n$, decrease n by 1 and delete s from P & G

Remarks.
1. each iteration grows P by starting with the previous P and extending it, if possible as in Handout#4
2. s is a sink in the *current* graph G

Implementation
data structure: it's convenient to use array $I[1..n]$ for 2 purposes:

$$I[v] = \begin{cases} t & \text{if } v \text{ has been deleted and assigned topological number } t \\ 0 & \text{if } v \text{ is still in } G \text{ (a special case is } v \text{ undiscovered – see Ex. 1)} \end{cases}$$

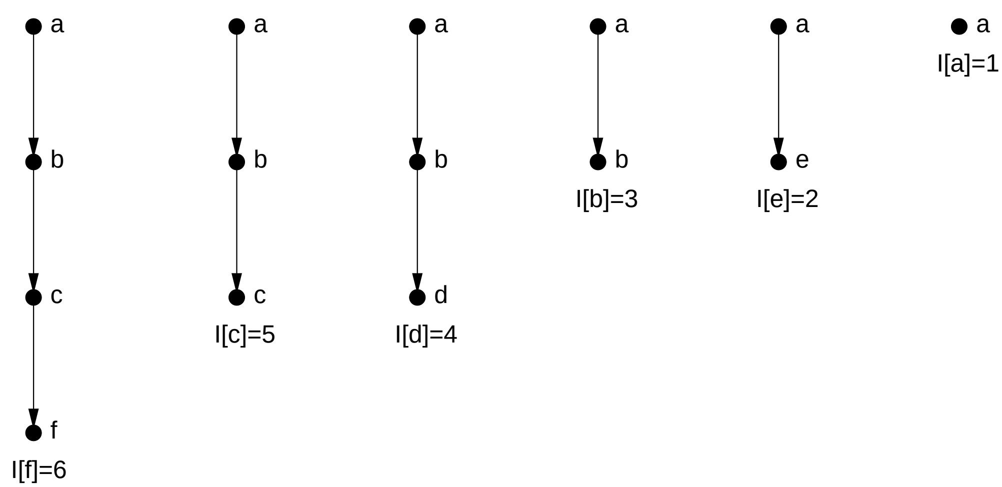
Topological Numbering Pseudocode

procedure `TOP`(G) {
$num = n$;
for each vertex v **do** $I[v] = 0$;
for each vertex v **do if** $I[v] = 0$ **then** `DFS`(v) }

procedure `DFS`(v) {
for each edge (v, w) **do**
  if $I[w] = 0$ **then** `DFS`(w);
$/*$ v is now a sink in the high level algorithm $*/$
$I[v] = num$; decrease num by 1; $/*$ v is now deleted $*/$ }

Remark. deletion of v is accomplished using the I array – we don't modify the adjacency structure

Example. G_0 could give topological order a, e, b, d, c, f, as in this execution:



Question. Explain how the pseudocode processes edge (b, f).

Timing
`TOP` uses time $\underline{O(m + n)}$

Exercises.

1. The algorithm uses the test $I[v] = 0$ to check if v has been discovered. But we can have $I[v] = 0$ when v is currently in P. Explain why this is irrelevant – whenever the algorithm examines $I[v]$, we have

$$I[v] = 0 \text{ if and only if } v \text{ has not been discovered.}$$

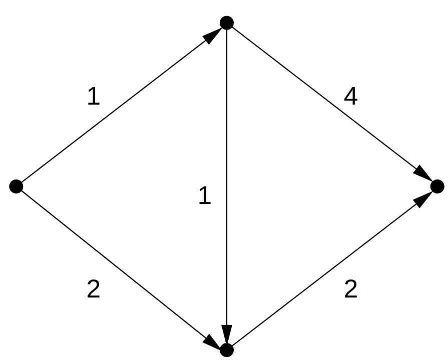
2. (a) Explain why we've shown that numbering the vertices in order of decreasing finish times in a dfs gives a topological numbering. (b) Would numbering in order of increasing discovery times give a valid topological numbering?

3. Give an algorithm for topological numbering that works by repeatedly deleting a source. The algorithm maintains a queue of sources, as well as the in-degree of each vertex. Your algorithm runs in time $O(n + m)$, although it makes two passes over the graph.

Remark. Dag algorithms often propagate information from higher topological numbers to lower, after scanning each edge (v, w) or at the end of `DFS`(v). Propagating in the opposite direction is also possible.

Algorithms on dags

suppose each edge e of a dag G has a real-valued length $\ell[e]$
we can find the longest path in G, in time $\underline{O(m + n)}$



longest path length 5

Idea
we'll set $d[v]$ to the length of a longest path starting at v
we'll compute $d[v]$ values for v in reverse topological order, using the formula
$(*)$ $\qquad d[v] = \max\{0,\ \ell[v, w] + d[w] : (v, w) \in E\}$

we calculate the values specified by $(*)$ by modifying our dfs code:

procedure LONGEST(G) {
/* we maintain the invariant, $d[v] = -1$ iff v is undiscovered */
for each vertex v **do** $d[v] = -1$;
for each vertex v **do if** $d[v] = -1$ **then** DFS(v) }

procedure DFS(v) {
$d[v] = 0$;
for each edge (v, w) **do** {
  if $d[w] = -1$ **then** DFS(w);
  /* at this point $d[w]$ equals its correct final value */
  $d[v] = \max\{d[v],\ \ell[v, w] + d[w]\}$ } }

Correctness
Prove by induction that $d[v]$ has the correct value at the end of DFS(v).

Exercise. Give a small digraph where the recurrence $(*)$ fails. Notes:
- for general digraphs "longest path" means longest simple path.
- Don't use negative edges.
- $(*)$ does not refer to topological numbers.

Remarks
1. the longest paths in a dag are known as the "critical paths"
   when we're doing *critical path scheduling* (CLRS p.594)

2. finding the longest path in a general graph is NP-complete!

3. a similar algorithm finds a path whose lengths have the greatest possible product
   e.g., find a maximum probability path
   this is the basis of Viterbi's algorithm for speech recognition (CLRS Pr.15-5, pp.367–68)

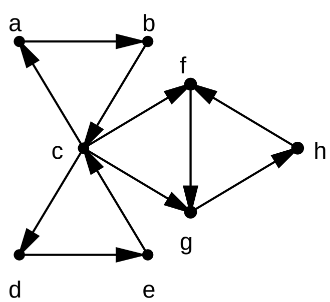
4. similar algorithms can be used to calculate the longest path from s to t
   or shortest paths from a vertex s, etc.

the analog of connectivity in digraphs is strong connectivity,
the fundamental concept on the structure of directed graphs

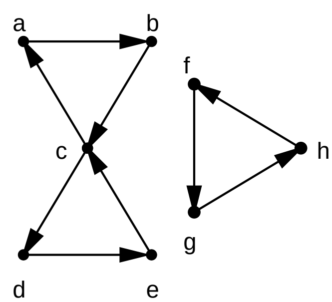
a digraph $G = (V, E)$ is *strongly connected* if every vertex can "reach" every other vertex
i.e., $(\forall u, v \in V)$ ($\exists$ a uv-path)

a quicker test: G is strongly connected if $\exists r \in V \ni$
r can reach every vertex, & every vertex can reach r

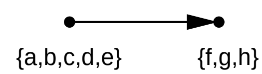
Strongly connected components (SCs) of a digraph G:
we partition the vertices into SC's according to this definition:
u & v are in the same SC $\Longleftrightarrow$ they can reach each other, i.e., $\exists$ a uv-path & $\exists$ a vu-path
(this is an equivalence relation – see CLRS p.1076, Theorem B.1)



Example digraph G_0



Strong components:
$\{a, b, c, d, e\}$, $\{f, g, h\}$



SC Graph

for any digraph, contracting each SC to a vertex gives the *strong component graph* ("SC graph")

Basic Facts

Lemma 1. *Let C be a cycle.*
(i) All vertices of C are in the same SC.
(ii) G and G/C have the same SC graph.

Proof Idea, part (ii):
show a path in G gives a path in G/C, & vice versa $\square$

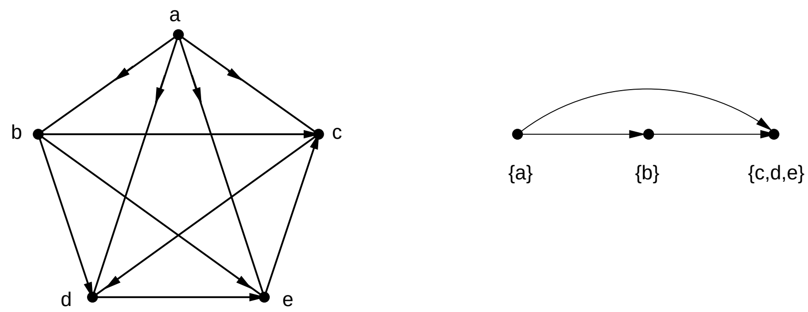
Lemma 2. *(i) Any dag is its own SC graph.*
(ii) Any SC graph is a dag.

Proof Idea, part (ii):
repeatedly contract cycles, until a dag is formed
apply Lemma 1(ii), and part (i) $\square$

Applications

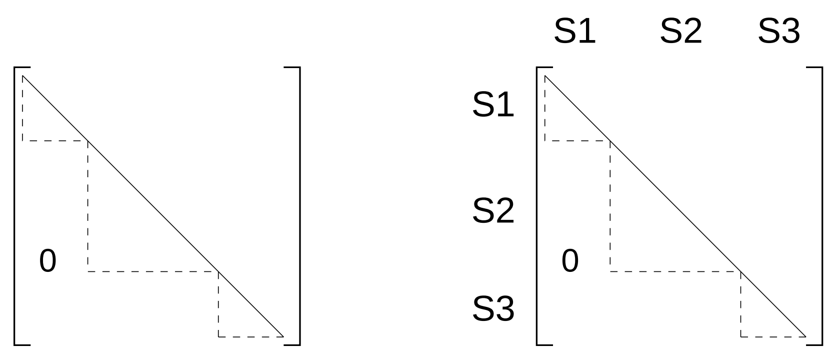
1. a Markov chain is *irreducible* $\Longleftrightarrow$
   the graph of its (nonzero) transition probabilities is strongly connected

2. in a *tournament* (i.e., a digraph where each pair of vertices is joined by exactly 1 edge)
   the strong component graph is a "complete dag", and so ranks the players



tournament SC graph

3. a *block diagonal matrix* is a sparse matrix
   whose entries below the diagonal are partitioned into blocks – Fig.(a):



(a) (b)

Gaussian elimination is efficient on block diagonal matrices –
there is limited fill-in

a common heuristic in sparse matrix packages uses strong components
to rearrange a given matrix to block diagonal
it is based on this principle:

let G be a digraph, with strong components $S_1, \ldots, S_n$ in topological order
(i.e., no edge goes from S_i to S_j, $j < i$)
number the vertices by strong component: first come the vertices of S_1, then S_2, etc.
the adjacency matrix for this numbering is block diagonal – see Fig.(b)

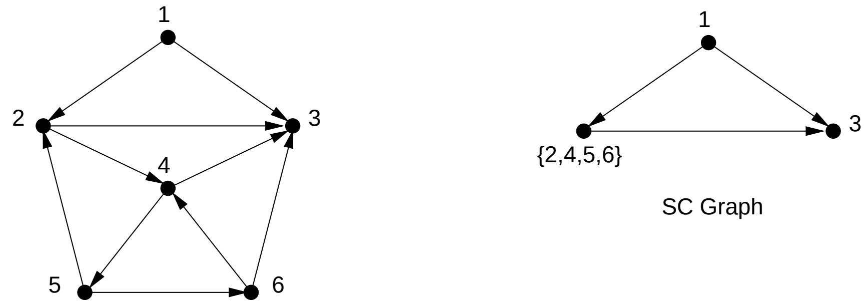
(Special Case: a dag has an upper triangular adjacency matrix)

Exercise. Explain why this is the "best" way to make a matrix block diagonal: In any permutation of an adjacency matrix, any block S_i is a union of SC's.

4. in a Buchi automaton, an infinite execution sequence is accepting
if it visits some accepting state infinitely often
i.e., some accepting state is in a nontrivial SC

it's easy to compute the strong components of a digraph in time $O(n(m+n))$
for each vertex v, find all the vertices it can reach
this is called the "transitive closure" (CLRS p.632)

using dfs we'll find the SCs in time $\underline{O(m+n)}$



Example graph G_0 & its SC graph

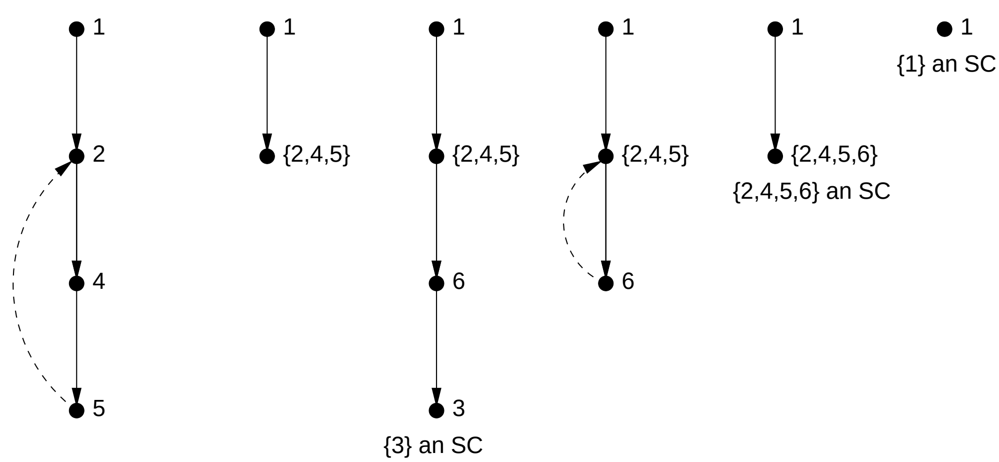
Basic ideas

all vertices on a cycle are in the same SC
in fact, the SC graph is formed by repeatedly contracting cycles

a sink s is a vertex of the SC graph
in fact, the SC's are $\{s\}$ and the SC's of $G - s$

High level algorithm

repeat until G has no vertices:
grow the dfs path P until a sink or a cycle is found
sink s: mark $\{s\}$ as an SC & delete s from P & G
cycle C: contract the vertices of C



Execution of the high-level algorithm on G_0

Implementing the high-level algorithm

we use stacks S with these operations (CLRS, p.201):

`TOP`(S): returns value of top of stack pointer (e.g., $S[\texttt{TOP}(S)]$ is the entry at top of stack)
`PUSH`(v, S): pushes element v onto stack S
`POP`(S): pops stack S; returns the value popped

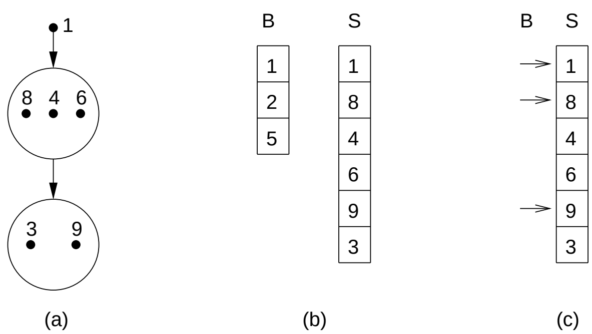
$S[1]$ is the lowest entry in the stack, not $S[0]$

3 data structures represent the dfs path P:

stack S contains the sequence of vertices in P
stack B contains the boundaries between contracted vertices

more precisely, S & B correspond to the dfs path $P = (v_1, \ldots, v_k)$ where $k = \texttt{TOP}(B)$
and for $i = 1, \ldots, k$, $v_i = \{S[j] : B[i] \le j < B[i+1]\}$
(when $i = k$, interpret $B[k+1]$ to be ∞)

at all times both S & B have $\le n$ entries



Stacks S & B.
(a) The search path in the high level algorithm has 3 vertices –
1 and contracted vertices $\{8, 4, 6\}$, $\{3, 9\}$.
(b) 2 arrays represent the search path. $\texttt{TOP}(B) = 3$, $\texttt{TOP}(S) = 6$.
(c) Our pictorial notation for the arrays: B contains the arrows to the left of S.

array $I[1..n]$ stores stack indices, for vertices in P
& it stores the strong component number of a vertex when that number is known

more precisely for a given vertex v at any point in time,

$$I[v] = \begin{cases} 0 & \text{if } v \text{ has never been in } P \text{ (i.e., } v \text{ undiscovered)} \\ j & \text{if } v \text{ is currently in } P \text{ and } S[j] = v \\ c & \text{if the SC containing } v \text{ has been deleted and numbered as } c \end{cases}$$

we number the SC's starting at $n + 1$
so the 3 cases correspond to $I[v] = 0$, $0 < I[v] \le n$, $n < I[v]$ respectively

Remark. using I for multiple purposes gave a 20% speed gain

Example. in Fig.1(c) below, $I[3]$ changes from its stack index 6 to its component number 7

Pseudocode for Strong Components Algorithm

procedure $\texttt{STRONG}(G)$ {
empty stacks S and B;
for $v \in V$ **do** $I[v] = 0$;
$c = n$;
for $v \in V$ **do if** $I[v] = 0$ **then** $\texttt{DFS}(v)$ }

procedure $\texttt{DFS}(v)$ {
$\texttt{PUSH}(v, S)$; $I[v] = \texttt{TOP}(S)$; $\texttt{PUSH}(I[v], B)$; /* add v to the end of P */
for edges $(v, w) \in E$ **do**
    if $I[w] = 0$ **then** $\texttt{DFS}(w)$
    else /* the following loop does contractions, when necessary */
        /* it handles deleted vertices too */
        while $B[\texttt{TOP}(B)] > I[w]$ **do** $\texttt{POP}(B)$;
if $B[\texttt{TOP}(B)] = I[v]$ **then** { /* number vertices of the next SC */
    $\texttt{POP}(B)$; increase c by 1;
    while $\texttt{TOP}(S) \ge I[v]$ **do** $I[\texttt{POP}(S)] = c$ }; }

Timing

$\underline{O(m + n)}$, since we spend $O(1)$ time on each vertex & edge
note every vertex gets pushed & popped exactly once from both S & B

Questions.

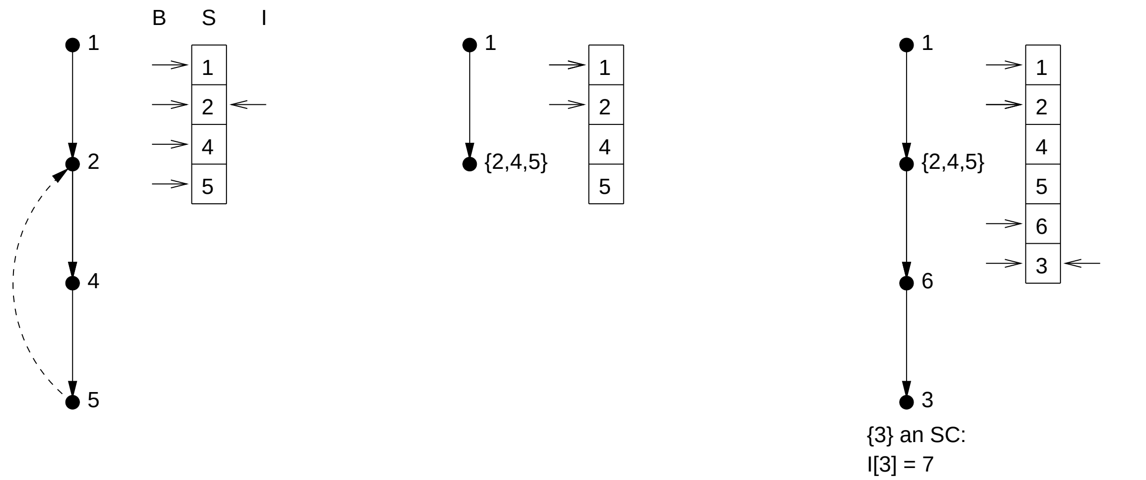
1. We could number the SC's starting at $n + 1$ and descending (perhaps as low as 2). Explain why this works.

2. The middle line in the definition of I says when $v \in P$, $I[v]$ points to v's entry in S. Explain why this is crucial to achieving the linear time.

3. Explain why the SC graph has vertex set $n + 1, \ldots, c$ and edge set $\{(I[v], I[w]) : (v, w) \in E,\ I[v] \ne I[w]\}$. How should $\texttt{STRONG}$ be changed so the vertices are topologically numbered?

Fig. 1. Execution of strong components algorithm on G_0

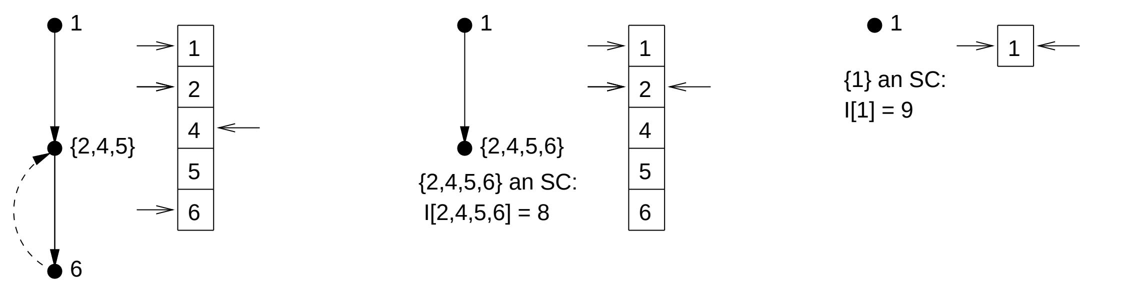
Key: B & I are indicated by the arrows into S. The entries of I examined by the algorithm are to the right of S. E.g., in Fig.1(d), $\mathtt{TOP}(B) = 3$, $B[\mathtt{TOP}(B)] = 5$, $I[4] = 3$.



(a) Since $I[2] = 2$, $(5, 2)$ completes a cycle.

(b) Cycle gets contracted.

(c) Before $\mathtt{DFS}(3)$ exits, $\mathtt{TOP}(B) = 4$ & $B[4] = I[3]$ indicate 3 starts an SC.



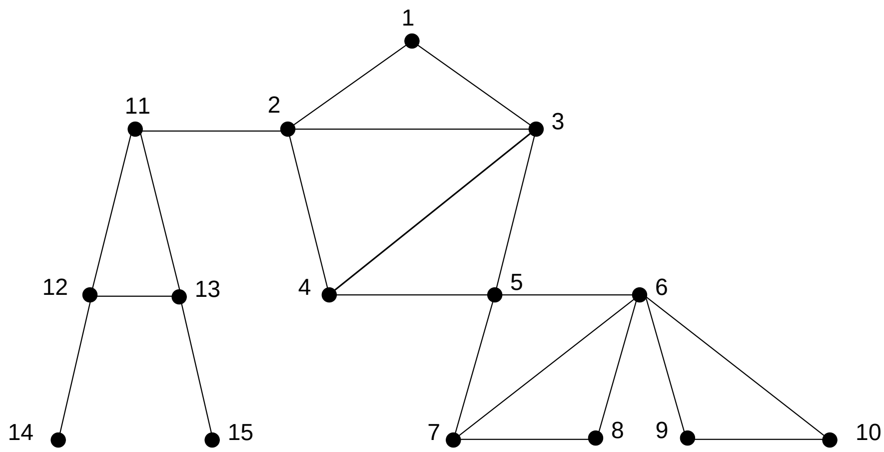
(d) Since $I[4] = 3$, $(6, 4)$ completes a cycle.

(e) Before $\mathtt{DFS}(2)$ exits, $\mathtt{TOP}(B) = 2$ & $B[2] = I[2]$ indicate 2 starts an SC.

(f) Before $\mathtt{DFS}(1)$ exits, $\mathtt{TOP}(B) = 1$ & $B[1] = I[1]$ indicate 1 starts an SC.

Exercise. Make sure you understand the pseudocode by describing how edges $(2, 6)$ (if it existed) and $(2, 3)$ get processed in Fig.1(e).

in this handout $G = (V, E)$ is a connected undirected graph



Example graph G_0 with 3 bridges & 6 cutpoints

edge e is a *bridge* of G if $G - e$ is not connected
vertex v is an *articulation point* (*cutpoint*) of G if $G - v$ is not connected

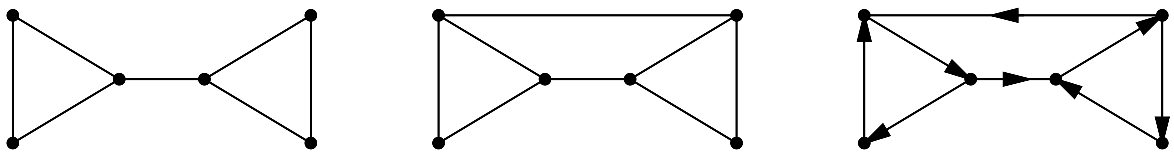
a graph is *bridgeless* if it has no bridges
a graph is *biconnected* if it has no cutpoint

Lemma. *e is a bridge $\iff$ it's not in any cycle.*
Proof. (v, w) is not a bridge $\iff$ some vw-path avoids (v, w) $\iff$ (v, w) is on a cycle $\square$

Applications

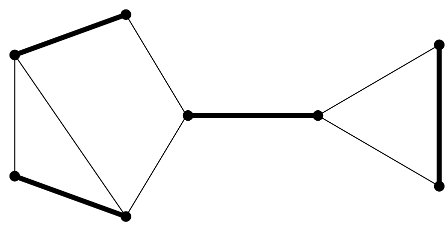
1. if a communications network (e.g., Internet) has a bridge,
   that link's failure disables communication
   similarly if it has an articulation point, that site's failure disables communication

2. **Robbins' Theorem** '39. *A connected undirected graph has a strongly connected orientation* $\Longleftrightarrow$ *it is bridgeless.*



Connected graph — Bridgeless graph — Strongly-connected orientation

3. **Kotzig's Theorem** '59. *A unique perfect matching contains a bridge of the graph.*



Graph & unique perfect matching.

Kotzig's Theorem can be used to find a unique perfect matching in time $O(m \log^4 n)$
(Gabow et.al., '01); see Handout#36

4. **Theorem.** (Whitney, '32). *A graph is planar* $\Longleftrightarrow$ *each biconnected component is planar.*

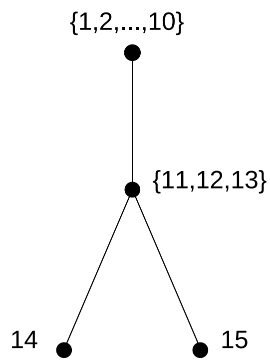
next handout shows how to find all the bridges in linear time
a similar algorithm (Handouts#41–42) finds all the cutpoints in linear time

as before assume G is a connected undirected graph
also continue to use G_0 of Handout#8 as our example graph

Bridge components

let B be the set of all bridges of G
the *bridge components* (BCs) of G are the connected components of $G - B$
i.e., a BC is a maximal set of vertices,
any of which can reach any other without crossing a bridge

contracting each BC to a vertex gives the *bridge tree*
Question. Explain why it's a tree, i.e., it has no cycle.



Bridge tree of graph G_0.

in this handout a contraction operation retains parallel edges
e.g. in $G_0/\{5,6,7\}$, 2 parallel edges join 8 & $\{5,6,7\}$

note G_0 & $G_0/\{5,6,7\}$ have the same bridges
Lemma. *If C is a cycle, G & G/C have the same bridges & the same bridge tree.*

Exercises.
1. Correct a small error in this proof of the Lemma: A nonbridge of G/C gives a nonbridge of G, and a nonbridge of G gives a nonbridge of G/C.
2. Explain why the proof of the lemma dictates that contraction must retain parallel edges.

we'll compute the bridges & bridge tree of a connected undirected graph in time $\underline{O(m+n)}$
the algorithm is almost identical to `STRONG`

note that two parallel edges form a cycle

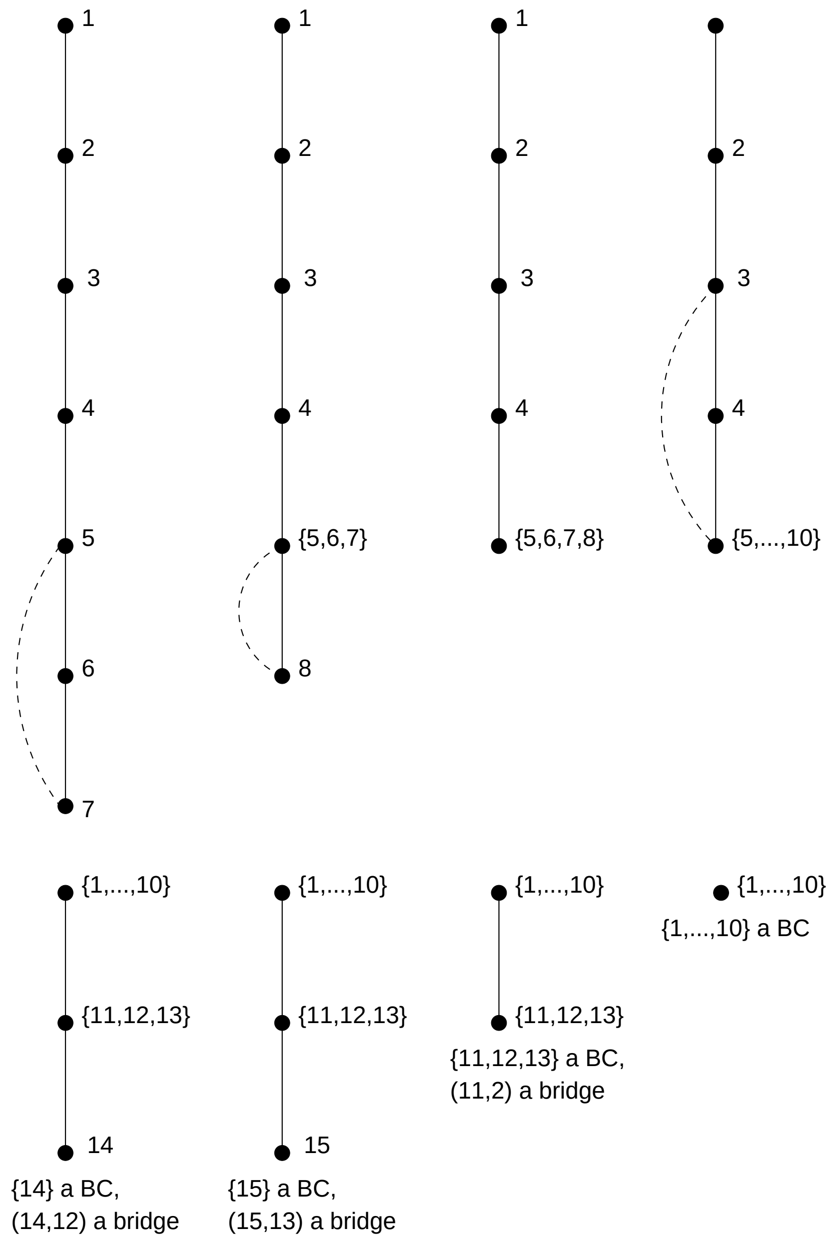
Basic ideas

all vertices on a cycle are in the same BC
in fact, the bridge tree is formed by repeatedly contracting cycles

a vertex x of degree ≤ 1 is a vertex of the bridge tree
in fact, the BC's are $\{x\}$ and the BC's of $G - x$

High level algorithm
say the last vertex x of a dfs path is a *dead end* if x has degree ≤ 1

repeat until G has no vertices:
grow the dfs path P until a cycle is found or a dead end is reached
cycle C: contract the vertices of C
dead end x: mark $\{x\}$ as a BC & delete x from P & G
if x has degree 1, mark its edge as a bridge (of the original graph)



Execution of the high-level algorithm on G_0.
Not all paths are shown. In the 2nd panel, parallel edges from 8 prevent a false bridge being marked.

Implementing the high-level algorithm

as in STRONG,

we represent the dfs path P using stacks S & B, & array I

we number the BC's starting at $n+1$

to conveniently identify the bridges, DFS will have two arguments DFS(v, u):

u is the vertex that calls DFS(v, u)

i.e., the search is exploring edge (u, v)

Pseudocode for Bridge Algorithm

the new code is underlined

for simplicity we assume the given graph does not have parallel edges

```
procedure BRIDGE(G) {
empty stacks S and B;
for v ∈ V do I[v] = 0;
c = n;
for v ∈ V do if I[v] = 0 then DFS(v, 0);
/* no need for a loop if G is known to be connected */ }

procedure DFS(v, u) {
PUSH(v, S); I[v] = TOP(S); PUSH(I[v], B); /* add v to the end of P */
for edges (v, w) ∈ E do
    if I[w] = 0 then DFS(w, v)
    else if w ≠ u then /* possible contract */ while B[TOP(B)] > I[w] do POP(B);
if B[TOP(B)] = I[v] then { /* number vertices of the next BC */
    POP(B); increase c by 1;
    while TOP(S) ≥ I[v] do I[POP(S)] = c;
    if u ≠ 0 then mark (u, v) as a bridge; } }
```

Exercises.

1. The test $w \neq u$ before contracting is crucial. Explain why omitting the test causes `BRIDGE` to *always* return with just 1 BC & no bridges.

2. Modify the code so it works for a multigraph, still in time $O(m+n)$.

3. Use the high level bridge algorithm to prove Robbins' Theorem. *Hint.* Run the BC algorithm. It shrinks the graph to 1 vertex. Orient the path edges down and the cycle edges up. Now the SC algorithm shrinks the oriented graph to 1 vertex. A bonus of this proof is that it gives a linear-time algorithm to strongly orient a bridgeless graph (see Exercise #5).

4. Here's a generalization of Robbin's Theorem. A *mixed graph* G is one that can have both directed and undirected edges. G is *traversable* if for every ordered pair of vertices u, v, there is a path from u to v that has all its directed edges pointing in the forward direction. (So if G is undirected, G is traversable $\Longleftrightarrow$ it's connected; if G is directed, G is traversable $\Longleftrightarrow$ it's strongly-connected.) A *bridge* of G is an undirected edge that is a bridge of the undirected graph formed by ignoring edge directions in G. An *orientation* of G assigns a unique direction to each undirected edge.

Theorem [Boesch & Tindell]. *A traversable mixed graph has a strongly-connected orientation* $\Longleftrightarrow$ *it has no bridge.*

Prove Boesch & Tindell's Theorem by giving a high-level dfs algorithm to orient the graph.

5. Implement the algorithm of #3 efficiently. The time should be $O(m+n)$ plus the time to maintain a data structure for set merging (Handout#35). This is $O(m+n)$ if the data structure of Gabow & Tarjan (CLRS p.522) is used.

6. As illustrated in Exercises 3–4, dfs is a powerful tool for proving theorems about graphs. Use dfs to prove this fact: A bipartite graph with a unique perfect matching has a vertex of degree 1.

Proof of the Lemma (page 1)
Lemma. *If C is a cycle, G & G/C have the same bridges & the same bridge tree.*

we'll use this fact:
Fact. *Contracting an edge of a cycle gives a cycle in the contracted graph.*
This depends on contractions retaining parallel edges – if they didn't, the Fact would fail when we contracted an edge of a triangle.

we'll prove Lemmas 1 & 2:
Lemma 1. *All vertices of a cycle belong to the same BC.*
Lemma 2. *If vertices x and y are in the same BC of G, then G and $G/\{x,y\}$ have the same bridges and bridge components.*
the Lemma follows easily from these 2 –
repeatedly contract 2 vertices that are consecutive in the cycle
[we're using the Fact here]

Proof of Lemma 1
Let C be the cycle. No edge of C is a bridge. So any 2 vertices of C can reach each other without crossing a bridge, ie, they're in the same BC. $\Box$

Proof of Lemma 2
We prove G and $G/\{x,y\}$ have the same bridges, in two steps, (i) & (ii):

(i) A bridge e of G is a bridge of $G/\{x,y\}$.
Proof. x & y are in the same connected component of $G-B$. So they're in the same connected component of $G-e$. So contracting x,y doesn't combine any connected components of $G-e$. Thus $G/\{x,y\}-e$ is not connected, ie, e is a bridge of $G/\{x,y\}$. $\diamond$

(ii) A nonbridge of G is a nonbridge of $G/\{x,y\}$.
We need to show that if e is on a cycle C of G, then e is on a cycle of $G/\{x,y\}$.
If the contraction actually changes the cycle C, it's because both x & y are in C. So the contraction simply shortcuts the cycle into another cycle containing e. $\diamond$
[We've used the Fact here.]

let $G = (V, E)$ be a connected bridgeless graph
we want to find a bridgeless subgraph $H = (V, F^*)$ of G
with as few edges as possible, i.e., $|F^*|$ is minimum

we usually write OPT instead of F^*

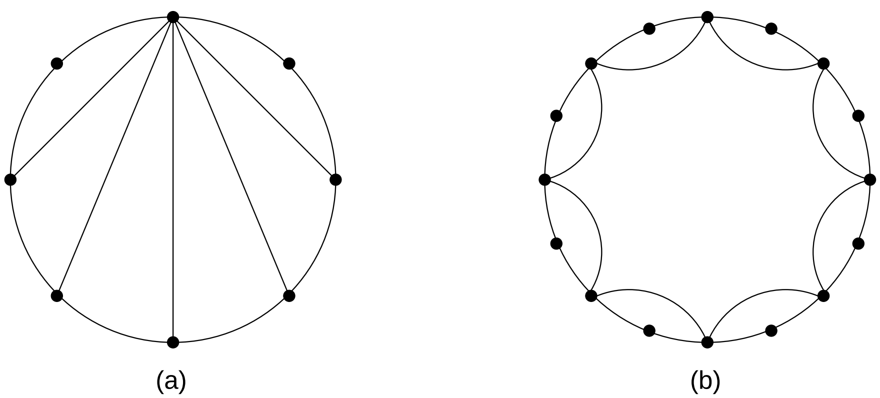


Fig.1. In both graphs OPT is a Hamiltonian cycle.

the problem is NP-hard

we'll give a "2-approximation algorithm" (i.e., it finds a bridgeless subgraph with $\le 2|OPT|$ edges)
& also a $\frac{3}{2}$-approximation algorithm

Factor 2 Approximation Algorithm

use the (high level) bridge algorithm
the solution graph contains all dfs path edges, and all edges causing a contraction

Proof of the Approximation Ratio

our solution graph has $n-1$ dfs path edges, and $\le n-1$ cycle edges
since every contraction decreases the number of vertices by ≥ 1

so it has $\le 2n$ edges

we'll use the *degree lower bound*: $|OPT| \ge n$
obviously the degree lower bound implies we have a 2-approximation

Proof of the Degree Lower Bound

any vertex in a bridgeless graph (with $n \ge 2$) has degree ≥ 2
so the Handshaking Lemma implies $2m \ge 2 \times n$, $m \ge n$ $\square$

Exercise. Show our bound is tight: On the graph of Fig.1(a) it's possible that our algorithm returns a solution graph with $2n-3$ edges. As $n \to \infty$, the approximation ratio $\frac{2n-3}{n}$ approaches 2.

The Carving Algorithm (Khuller & Vishkin, *J. ACM* '94)

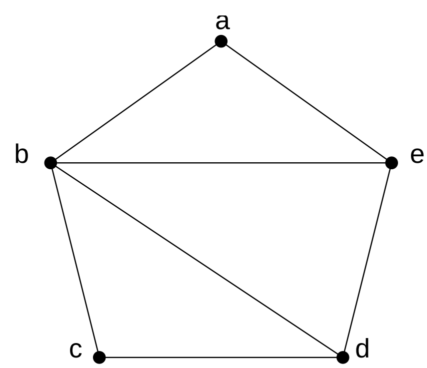
an obvious improvement is to use cycle edges that contract as many vertices as possible
this improves the performance bound to 3/2

Algorithm

F denotes the edges of the algorithm's solution
initially $F = \emptyset$

repeat until G has 1 vertex:
- grow the dfs path P until its endpoint x has all neighbors belonging to P
- let y be the neighbor of x closest to the start of P
- let C be the cycle formed by edge (x, y) & edges of P
- add all edges of C to F
- contract the vertices of C

Example:



Execution 1: $P = a, b, c, d, e$; contract for edge (e, a).
This gives $|F| = 5 = |OPT|$.

Execution 2: $P = a, b, c, d, e$; contract for edge (e, b).
Invalid: b doesn't satisfy the condition for y.

Execution 3: $P = a, b, d, c$; contract for edge (c, b);
$P = a, \{b, c, d\}, e$; contract for edge (e, a).
This gives $|F| = |OPT| + 1$.

Execution 4: $P = a, b, c, d$; contract for edge (b, d).
Invalid: d doesn't satisfy the condition for x.

Proof of the Approximation Ratio

let c be the number of cycles contracted by the algorithm
the key fact is the *Carving Lower Bound*:

$$|OPT| \geq 2c$$

first note the carving lower bound implies a 3/2 approximation ratio:
as before, $|F| = (n-1) + c$

using the Degree & Carving Bounds we get

$$|F| = (n-1) + c \leq |OPT| + |OPT|/2 = (3/2)|OPT|$$

Proof of the Carving Lower Bound

Basic Principle: In a bridgeless graph, any set of vertices S, $S \neq \emptyset, V$,
has ≥ 2 edges leaving it.

let x be an endpoint of P giving a contraction, as in the algorithm
OPT contains ≥ 2 edges leaving each x

all the edges leaving x disappear after the graph is contracted
so no edge of the original graph G leaves two x's

$\therefore$ OPT contains $\geq 2c$ edges $\Box$

Remark. The main issue in this proof is being sure we don't "double-count", i.e., count an edge of OPT twice. The contraction ensures we don't double count.

Example: (cont'd)

Execution 1: $c = 1$. The proof says OPT contains ≥ 2 edges incident to e.

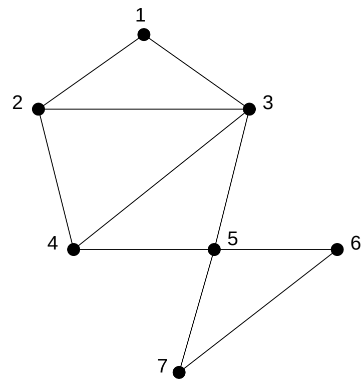
Execution 3: $c = 2$. The proof says OPT contains ≥ 2 edges incident to c, plus ≥ 2 edges incident to e; no edge is incident to both c & e.

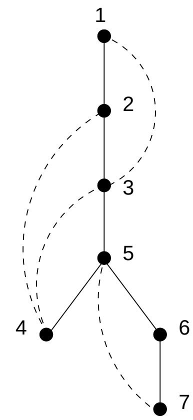
Exercise. Show our bound is tight:
(a) On the graph of Fig.1(b) it's possible that our algorithm returns a solution graph with $\frac{3}{2}n - 1$ edges.
(b) Give a more devastating example: Delete 1 vertex from Fig.1(b), and show the algorithm can give a solution with $3\frac{n-1}{2}$ edges that's *minimal*, i.e., no edge can be deleted

Jothi, Raghavachari & Varadrajan (*SODA '03*)
use a more involved DFS to achieve performance ratio 5/4
they also use a better lower bound: D_2, the smallest subgraph with minimum degree 2

Remark. These algorithms illustrate the importance of *good lower bounds* in designing approximation algorithms.

advanced dfs algorithms (e.g., planarity) use the depth-first spanning tree





Graph G_0 & its dfs tree

General remarks

1. *Tree-Drawing Convention*
   when drawing a dfs-tree,
   the children of a vertex are drawn left-to-right in the order they are discovered

2. discovery of v is often called the *preorder visit* of v
   finish of v is the *postorder visit* of v

   these correspond to preorder and postorder traversals of the df forest

Undirected graphs

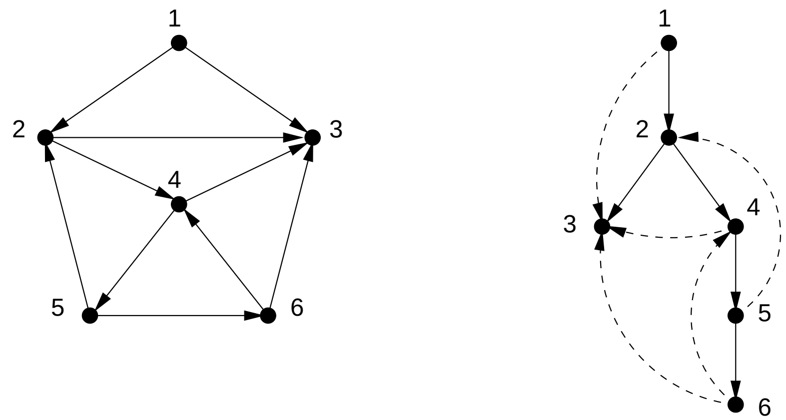
2 vertices in a tree are *related* if one is an ancestor of the other

in a dfs of an undirected graph the nontree edges are called *back edges*

the key fact:
any back edge joins 2 vertices that are related in the dfs tree
i.e., there are no "cross edges"

intuitively, *dfs makes a graph look like a tree*

Digraphs



Graph G_0 of Handout#7 & its dfs tree

a nontree edges is either
- *forward* (directed from ancestor to descendant),
- *back* (directed from descendant to ancestor), or
- *cross* (joins 2 unrelated vertices)

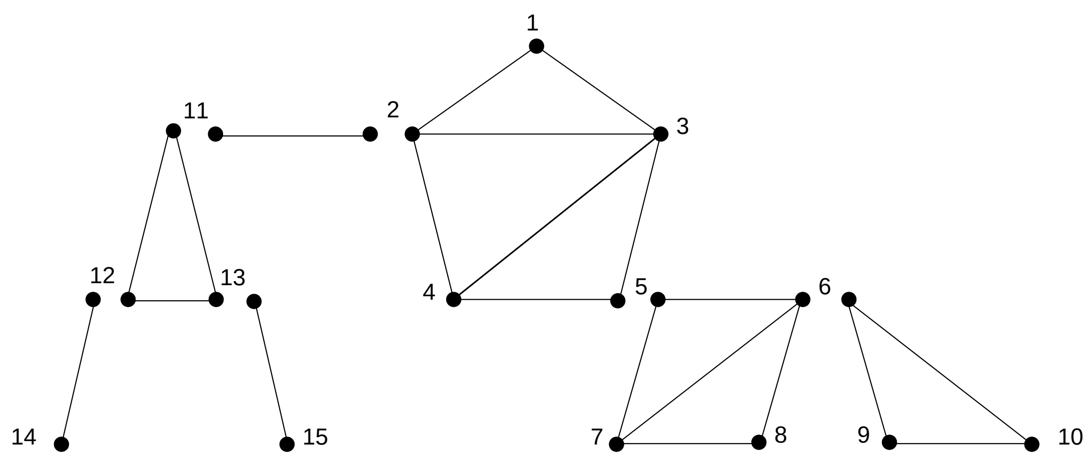
key fact:
any cross edge is directed from right to left

assume G is a connected undirected graph

Biconnected components/Blocks

the *biconnected components (blocks)* are the maximal biconnected subgraphs of G
i.e., a block is a maximal set of edges,
any 2 of which are in a common cycle

Exercise. Prove the 2 definitions are equivalent. (Note, we'll only use the 2nd definition.)

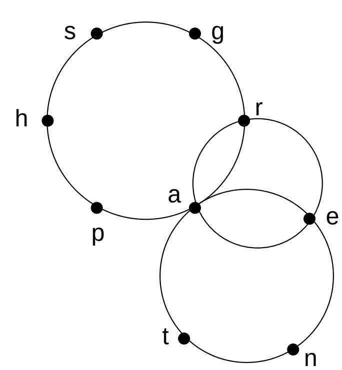


The 7 blocks of graph G_0 of Handout#8

we'ld like a succinct representation of the blocks
Question. Explain why there's no obvious representation based on contracting each block. In this respect blocks don't behave like BCs or SCs.

Hypergraphs

a *hypergraph* $H = (V, E)$ consists of a finite set V of *vertices*
& a finite set E of *edges*, where each edge is a subset of V
we sometimes call an element of E a *hyperedge*



a *path* in H is a sequence $v_1, e_1, \ldots, v_k, e_k$, $k \ge 1$,
of distinct vertices v_i and distinct edges e_i, $1 \le i \le k$,
where $v_1 \in e_1$ and $v_i \in e_{i-1} \cap e_i$ for every $1 < i \le k$
by convention a sequence of one vertex v_1 is also a path
the set of vertices in P is denoted $V(P) = \cup_{i=1}^{k} e_i$

a *cycle* is a path with the additional properties that $k > 1$ and $v_1 \in e_k$

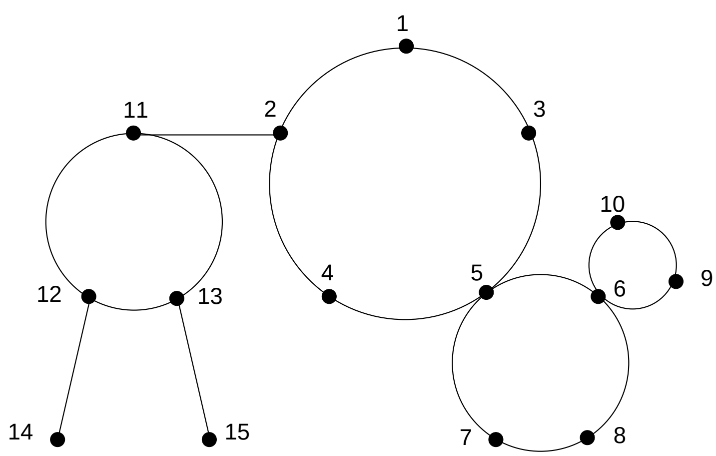
a hypergraph is *acyclic* if it contains no cycle

to *merge* edges e_i, $i = 1, \ldots, k$,
add a new edge $\cup_{i=1}^{k} e_i$ and delete every edge properly contained in it (e.g., e_i)
a *merging* of hypergraph H is a hypergraph formed by doing zero or more merges on H

Block Hypergraph

the *block hypergraph* H of G is the hypergraph formed by merging the edges of each block of G
H is an acyclic hypergraph

Question. Explain why (a) H is acyclic; (b) any 2 hyperedges of H share at most 1 common vertex.

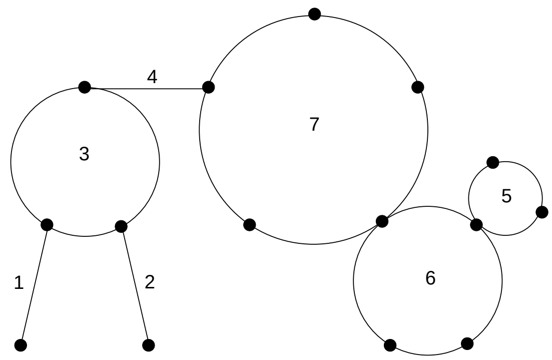


The block hypergraph of G_0 of Handout#8 a convenient way to represent the blocks is to number the hyperedges of H bottom-up

i.e., choose a hyperedge of H as the root

this implicitly defines the "child" hyperedges of each hyperedge of H

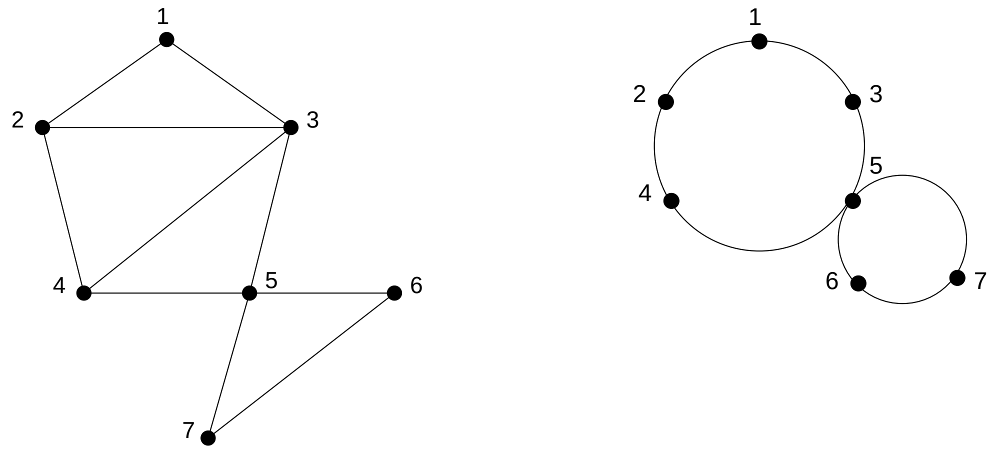
assign a unique number to each hyperedge, that is larger than the number of any child



for each $v \in V$ let $I[v]$ be the largest number of a hyperedge containing v

any edge $(v, w) \in E$ belongs to the block numbered $\min\{I[v], I[w]\}$

we compute the blocks and articulation points of a connected undirected graph in time $\underline{O(m+n)}$



Example graph G_0 & its block hypergraph

Basic ideas

all edges on a cycle are in the same block

    in fact, the block hypergraph is formed by repeatedly merging cycles

a *pendant edge* has ≤ 1 vertex in another edge

a pendant edge e is an edge of the block hypergraph

    in fact, the blocks are e and the blocks of $H - e$

a *dfs path* in a hypergraph is defined just like in a graph –

    we keep on adding an edge to the end of the path

High level algorithm

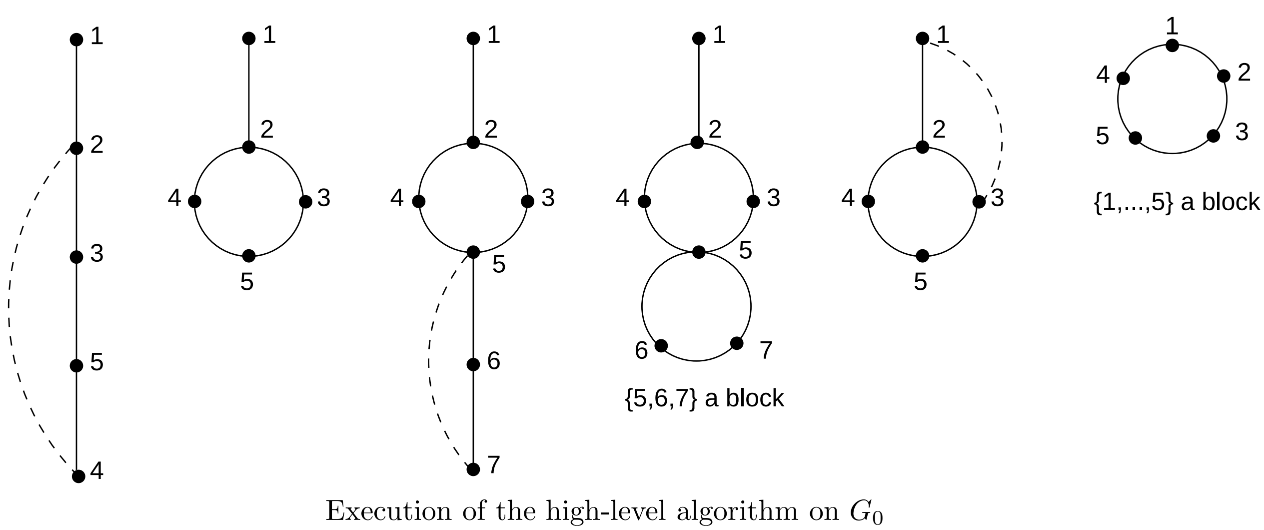
repeat until G has no edges:

  grow the dfs path P until a pendant edge or a cycle is found

    pendant edge e: mark e as a block;

      delete e & its preceding vertex from P; delete e from G

    cycle C: merge the edges of C



Execution of the high-level algorithm on G_0

Implementing the high-level algorithm

we represent the dfs path P using stacks S & B, & array I
we number the blocks starting at $n+1$

stack S gives the sequence of vertices in P, as before

stack B gives the boundaries between hyperedges of P, 2 vertices per boundary

more precisely, S & B correspond to the dfs path $P = (v_1, e_1, \ldots, v_k, e_k)$, $k \ge 1$, where $\mathtt{TOP}(B) = 2k$
and for $i = 1, \ldots, k$,
$v_i = S[B[2i-1]]$
$e_i = v_i \cup \{S[j] : B[2i] \le j < B[2i+2]\}$

when $i = k$, interpret $B[2k+2]$ to be ∞
when $k \ge 1$ we have $B[i] = i$ for $i = 1, 2$
at certain points P is a path (v), in which case $S[1] = v$, $\mathtt{TOP}(S) = 1$ and $\mathtt{TOP}(B) = 0$

array I is similar to `STRONG`:

$$I[v] = \begin{cases} 0 & \text{if } v \text{ has never been in } P \\ j & \text{if } v \text{ is currently in } P \text{ and } S[j] = v \\ c & \text{if the last block containing } v \text{ has been output \& numbered as } c \end{cases}$$

Pseudocode for Block Algorithm

procedure $\mathtt{BLOCKS}(G)$ {
empty stacks S and B;
for $v \in V$ **do** $I[v] = 0$;
$c = n$;
for $v \in V$ **do if** $I[v] = 0$ and v is not isolated **then** $\mathtt{DFS}(v)$ }

procedure $\mathtt{DFS}(v)$ {
$\mathtt{PUSH}(v, S)$; $I[v] = \mathtt{TOP}(S)$; **if** $I[v] > 1$ **then** $\mathtt{PUSH}(I[v], B)$; /*v is the second boundary vertex*/
for edges $(v, w) \in E$ **do** {
  if $I[w] = 0$ **then** { $\mathtt{PUSH}(I[v], B)$; $\mathtt{DFS}(w)$ /* v is the first boundary vertex */ }
  else /* possible merge */ **while** $I[v] > 1$ and $I[w] < B[\mathtt{TOP}(B) - 1]$ **do** {$\mathtt{POP}(B)$; $\mathtt{POP}(B)$}
if $I[v] = 1$ **then** $I[\mathtt{POP}(S)] = c$
else if $I[v] = B[\mathtt{TOP}(B)]$ **then** {
  $\mathtt{POP}(B)$; $\mathtt{POP}(B)$; increase c by 1;
  while $\mathtt{TOP}(S) \ge I[v]$ **do** $I[\mathtt{POP}(S)] = c$ } }

Exercise. Modify the pseudocode so it marks the articulation points. Do the same for the bridges.

Execution of block algorithm on G_0