Graph Algorithms

References:

- Algorithms, Jeff Erickson, Chapter 5 Basic Graph Algorithms, Chapter 6, Depth-First Search
- Algorithm Desgin Manual, Chapter 5.

A simple and incredibly versatile and useful data structure to represent pair-wise relationships.

Formally, a (simple) graph is a pair of sets

$$(V, E)$$

where V is the set of **vertices** (or **nodes**), and E is set of pairs of elements in V, which we call **edges**. In **undirected** graph, the edge is unordered pair, i.e. $(u, v) = (v, u)$, whereas in **directed** graph, the pair is ordered. In the textbook, unordered pair is usually shortened as uv, and ordered pair is $u \longrightarrow v$.

Depending on context, the book also uses V, E, as the **number** of vertices and edges. E.g. in a statement such as $0 \leqslant E \leqslant V(V-1)$

Without qualification, we usualy mean ***simple*** graph which

- No parallel edges: E is a **set** of pairs
- No loop (undirected edge with self)

In undirected graph, if $uv \in E$, then we say v is a ***neighbor*** of u, and u, v are ***adjacent***.The ***degree*** of a node is the number of neighbors.

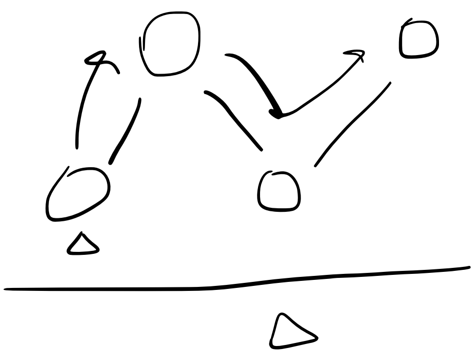
In directed graph, we distinguish two kind of neighbors: for edge $u \longrightarrow v$, we call u the ***predecessor*** of v, and v the ***successor*** of u. The ***in-degree*** of a node is the number of predecessors, and ***out-degree*** is the number of sucessors.

A graph $G' = (V', E')$ is a ***subgraph*** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A ***proper subgraph*** is any subgraph that is not G itself.

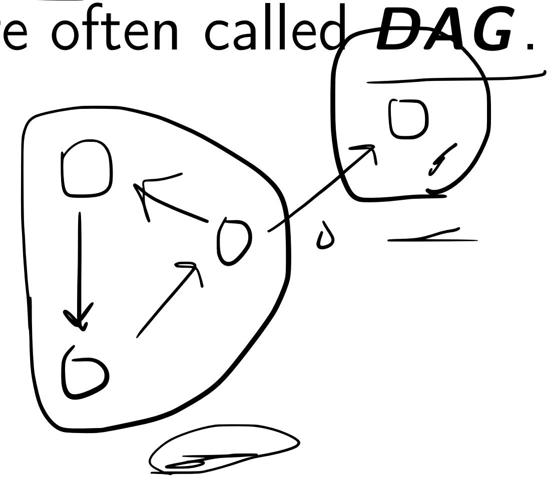
A ***walk*** in an undirected graph G is a *sequence of vertices*, where each adjacent pairs in the sequence is also adjacent in the graph. Also a walk can mean the sequence of edges. A walk is called a ***path*** if it visits each vertex at most once. For any two vertices u, v, we say v is ***reachable*** from u is G contains a walk (and therefore a path) between u and v. A graph is ***connected*** if every vertex is reachable from every other vertex. Every undirected graph consists of one or more ***components***, which are maximal connected subgraphs; two vertices are in the same component iff there is path between them.

A walk is ***closed*** if it starts and ends in the same vertex; a ***cycle*** is a closed walk that enters and leaves each vertex at most once. An undirected graph is ***acyclic*** if no subgraph is a cycle; acyclic graphs are also called ***forests***. A ***tree*** is a connected acyclic graph (one component of a forest). A ***spanning tree*** of G is a subgraph that is a tree and contains every node of G. (Only) A connected graph contains spanning tree. A ***spanning forest*** of G is a collection of spanning trees, one for each component.

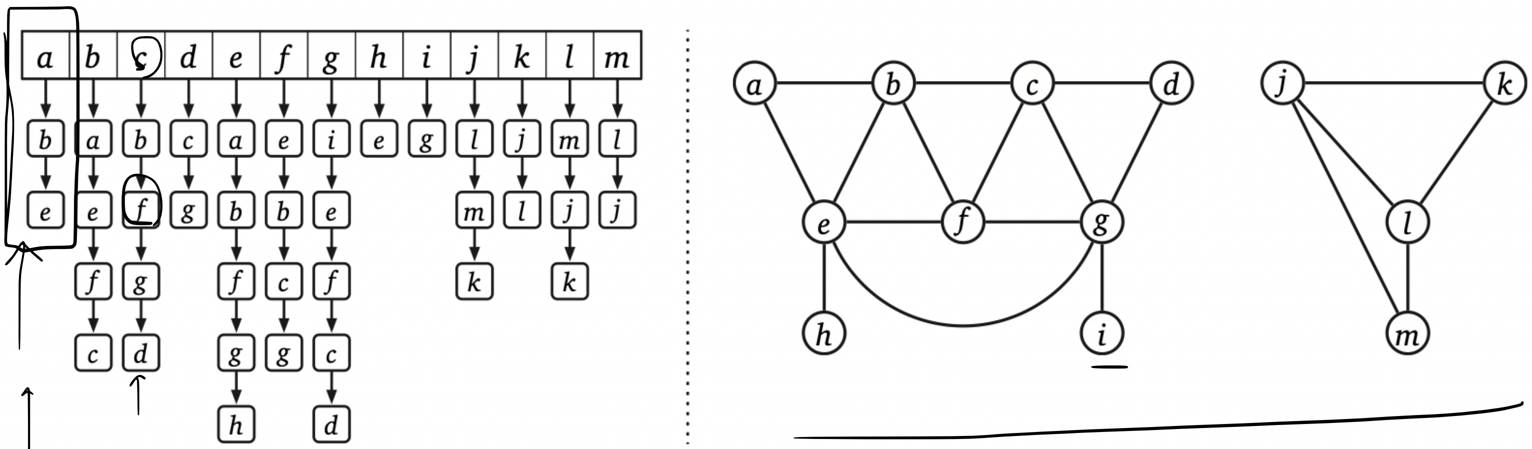
In directed graph the definitions are similar, with the distinction that the edges, and therefore walk and paths are *directed*. A directed graph is ***strongly connected*** if every vertex is reachable from every other vertex; A directed graph is ***acyclic*** if it does not contain a directed cycle. directed acyclic graph are often called ***DAG***.



$n-1$



Adjacency list:



Here the list of neighbors for each vertex is stored in a linked list; but it does not have to be linked list;

A dynamic array (vector), binary search tree, or hash table could also be used. Different data structures have different performance characteristics.

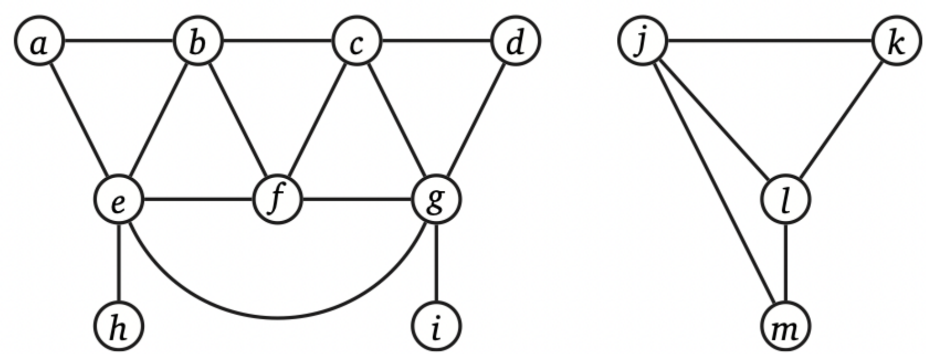
Adjacency matrix: a $V \times V$ matrix A of 0s and 1s:

- For undirected graph, $A[u, v] = 1$ iff $uv \in E$
- For directed graph, $A[u, v] = 1$ iff $u \rightarrow v \in E$

The adjacency matrix for undirected graph is symmetric, and the diagonal is always 0. For directed graph it's not necessarily symmetric, and the diagonal could be non-zero.

V^2

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*
a	0	1	0	0	1	0	0	0	0	0	0	0	0
b	1	0	1	0	1	1	0	0	0	0	0	0	0
c	0	1	0	1	0	1	1	0	0	0	0	0	0
d	0	0	1	0	0	0	1	0	0	0	0	0	0
e	1	1	0	0	0	1	1	1	0	0	0	0	0
f	0	1	1	0	1	0	1	0	0	0	0	0	0
g	0	0	1	1	1	1	0	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	1	1	1
k	0	0	0	0	0	0	0	0	0	1	0	1	0
l	0	0	0	0	0	0	0	0	0	1	1	0	1
m	0	0	0	0	0	0	0	0	0	1	0	1	0



Comparison:

	Standard adjacency list (linked lists)	Fast adjacency list (hash tables)	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Test if $uv \in E$	$O(1+\min\{\deg(u),\deg(v)\}) = O(V)$	$O(1)$	$O(1)$
Test if $u \to v \in E$	$O(1+\deg(u)) = O(V)$	$O(1)$	$O(1)$
List v's (out-)neighbors	$\Theta(1+\deg(v)) = O(V)$	$\Theta(1+\deg(v)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$O(1)$	$O(1)^*$	$O(1)$
Delete edge uv	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$	$O(1)$

Why three representations?

- If the graph is dense $E \approx O(V^2)$, the adj matrix is both simple and efficient.
- Adj with linked list is usually good enough
- Many problems have **implicit** graph representation, and they can be modeled by either adj list or adj matrix.

Comparison	Winner
Faster to test if (x, y) is in graph?	adjacency matrices
Faster to find the degree of a vertex?	adjacency lists
Less memory on small graphs?	adjacency lists $(m + n)$ vs. (n^2)
Less memory on big graphs?	adjacency matrices (a small win)
Edge insertion or deletion?	adjacency matrices $O(1)$ vs. $O(d)$
Faster to traverse the graph?	adjacency lists $\Theta(m + n)$ vs. $\Theta(n^2)$
Better for most problems?	adjacency lists

Figure. Comparison in cost of adj list/matrix.

From Algorithm Design Manual, Skiena.

We always assume adj list with linked list as the graph representation, unless explicitly stated otherwise.

Whatever-First Search

Reachability problem: given vertex s in undirected graph G, which vertices are reachable from s? We must search through the graph to find out the answer.

The most natural reachability algorithm is (for those of us who are enchanted by the magic of recursion) ***depth-first search***. This algorithm can be written either recursively or iteratively:

```
RecursiveDFS(v):
  if v is unmarked
    mark v
    for each edge vw:
      RecursiveDFS(w)
```

```
IterativeDFS(s):
  push(s)
  while the stack not empty
    v ← pop stack
    if v is unmarked
      mark v
        for each edge vw
          push(w)
```

DFS is just one of the whatever-first-search method. The generic algorithm stores a set of candidate edges in some data structure called "bag". The only important thing about "bag" is that you can put something in and later get it out. A stack is such a "bag", and it leads to DFS (the iterative version).

```
WhateverFirstSearch(s):
    put s into the bag
    while the bag is not empty
        take v from the bag
        if v is unmarked
            mark v
            for each edge vw
                put w into the bag
```

The generic WhateverFirstSearch algorithm marks every node reachable from s, and nothing else. It visits every node in the connected graph *at least* once. To help prove it and analyze its performance, let's consider the slightly modified algorithm (modification in red):

```
WhateverFirstSearch(s):
    put (∅, s) into the bag
    while the bag is not empty
        take (p, v) from the bag                          (take-out)
        if v is unmarked
            mark v
            parent(v) ← p
            for each edge vw                              (neighbor-loop)
                put (v, w) into the bag                   (put-in)
```

Note that the modified version does exactly the same thing as before, but additionally **records** parent node (who put me in the bag the first time?) along the way. This parent relationship helps us understand and prove the claim that the search algorithm indeed visits all reachable nodes at least once.

Lemma. WhateverFirstSearch(s) *marks **every** vertex reachable from s and only those vertices. Moreover, the set of all pairs* $(v, \text{parent}(v))$ *with* $\text{parent}(v) \neq \emptyset$ *defines a **spanning tree** of the component containing* s.

Sketch proof:

- First, we argue that the algorithm marks every reachable vertex v from s.

  Prove by induction on the shortest-path length from s to vertex v.

- Second, we argue that the pairs (v,parent(v)) forms a spanning tree of the component containing s.

  Claim: for every vertex v, the parent path: $v \longrightarrow \text{parent}(v) \longrightarrow \text{parent}(\text{parent}(v)) \longrightarrow \cdots \longrightarrow$ eventually leads to s.

  Claim: all the parent edges form a tree

Analysis. The running time of the whatever-first-search depends on what data structures we use for the "bag", but we can make a few general observations.

Suppose putting a node into the "bag" or getting one out takes T time.

- The (neighbor-loop) is executed **exactly once** for each marked vertex v, and therefore at most V times.

- Each edge uv in the component is put into bag **exactly twice**; once as (u, v) and once as (v, u). So the (put-in) statement is executed at most $2E$ times.

- For the (take-out) statement, we can't take more out than we put in, so it's executed at most $2E$ times.

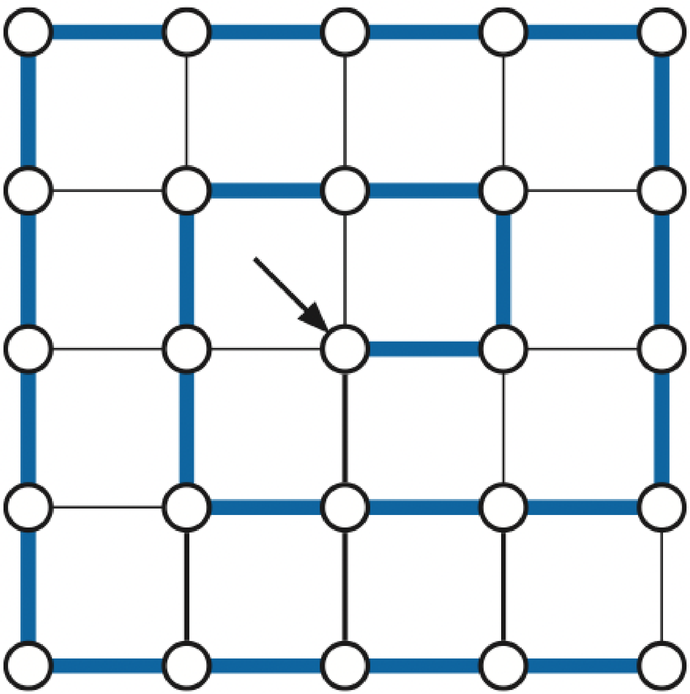
So, WhateverFirstSearch() takes $\boldsymbol{O(V + ET)}$ time. (If graph is represented as adj matrix then it's $O(V^2 + ET)$. Why?

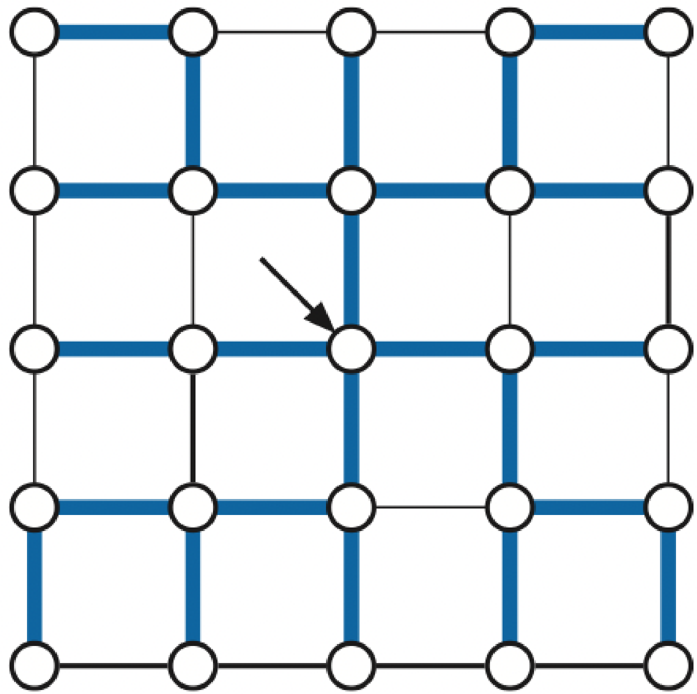
Important Variants:

Stack: Depth-First

Queue: Breadth-First

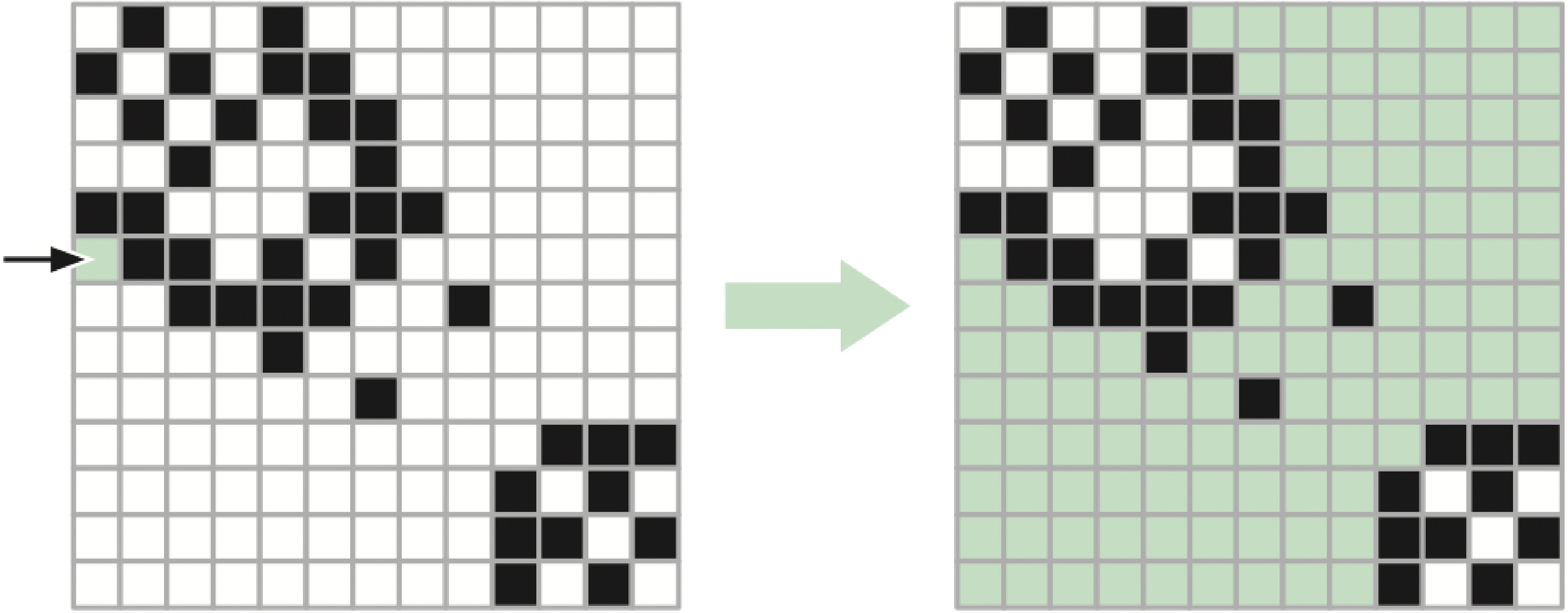
Priority-queue: Best-First





Flood Fill

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



Data Structures for Graph.

```
class Graph {
public:
    enum EdgeClass {Unknown, TreeEdge, BackEdge, ForwardEdge, CrossEdge};
    int V; // # nodes
    int E; // # edge;
    int clock; // for DFS
    bool directed;
    vector<vector<int>> edgelists; // adj list
    vector<bool> marked; // for DFS/BFS
    vector<int> parent; // for DFS/BFS
    vector<int> pre;   // for DFS;
    vector<int> post;   // for DFS;
    vector<int> tree_out_degree;
    vector<int> earliest_reach;
    Graph(int n) : directed(false), V(n), clock(0), edgelists(n,vector<int>())
                , marked(n, false), parent(n, -1), pre(n, -1), post(n,-1)
                , tree_out_degree(n,0), earliest_reach(n,-1){}
};
```

BFS for undirected graph.

```
void bfs(Graph &G, int start) {
    queue q;
    int v, y;
    q.push(start);
    G.marked[start] = true;
    while(!q.empty()) {
        auto v = q.back();
        q.pop();
        process_vertex_early(v);
        q.marked[v] = true;
        for(auto y : G.edgelists[v]) {
            if (!G.marked[y]) {
                G.marked[y] = true;
                q.push(y);
                G.parent[y] = v;
            }
        }
        process_vertex_late(v);
    }
}
```

Applications of BFS:

1. Connected components
2. Shortest path from a source s

Depth First Search

DFS has some special properties. It's a variant of Whatever-First-Search, but it's usually implemented in recursion. And we can modify the algorithm a bit so that we check the mark of w *before* we put into bag, so every reachable vertex w is put in bag exactly once:

```
DFS(v):
    mark v
    PreVisit(v)
    for each edge vw
        if w is unmarked                          #modified!
            parent(w) ← v
            DFS(w)
    PostVisit(v)
```

and we have the two magic unspecified blax-box subroutines called PreVisit and PostVisit pre/post the recursion. By putting computations in the two subroutines we can solve many problems.

For undirected graph, we already know that DFS visits the component containing v, and the parent() relation defines a spanning tree.

If the graph is not connected, we can wrap around DFS like this to visit all vertices (two equivalent formulations)

```
DFSAll(G):
  Preprocess(G)
  for all vertices v
    unmark v
  for all vertices v
    if v is unmarked
      DFS(v)
```

```
DFSAll(G):
  Preprocess(G)
  add vertex s
  for all vertices v
    add edge s → v
    unmark v
  DFS(s)
```

Preorder and Postorder

Hopefully you already have some experience with preorder/posorder traversals of rooted *trees*, both can be computed with DFS.

Similary traversals can be defined for arbitrary directed graphs

Preprocess(G):	PreVisit(v):	PostVisit(v):
$\text{clock} \leftarrow 0$	$\text{clock} \leftarrow \text{clock} + 1$	$\text{clock} \leftarrow \text{clock} + 1$
	$v.\text{pre} \leftarrow \text{clock}$	$v.\text{post} \leftarrow \text{clock}$

Now each vertex is timestamped by two clock readings: v.pre and v.post.

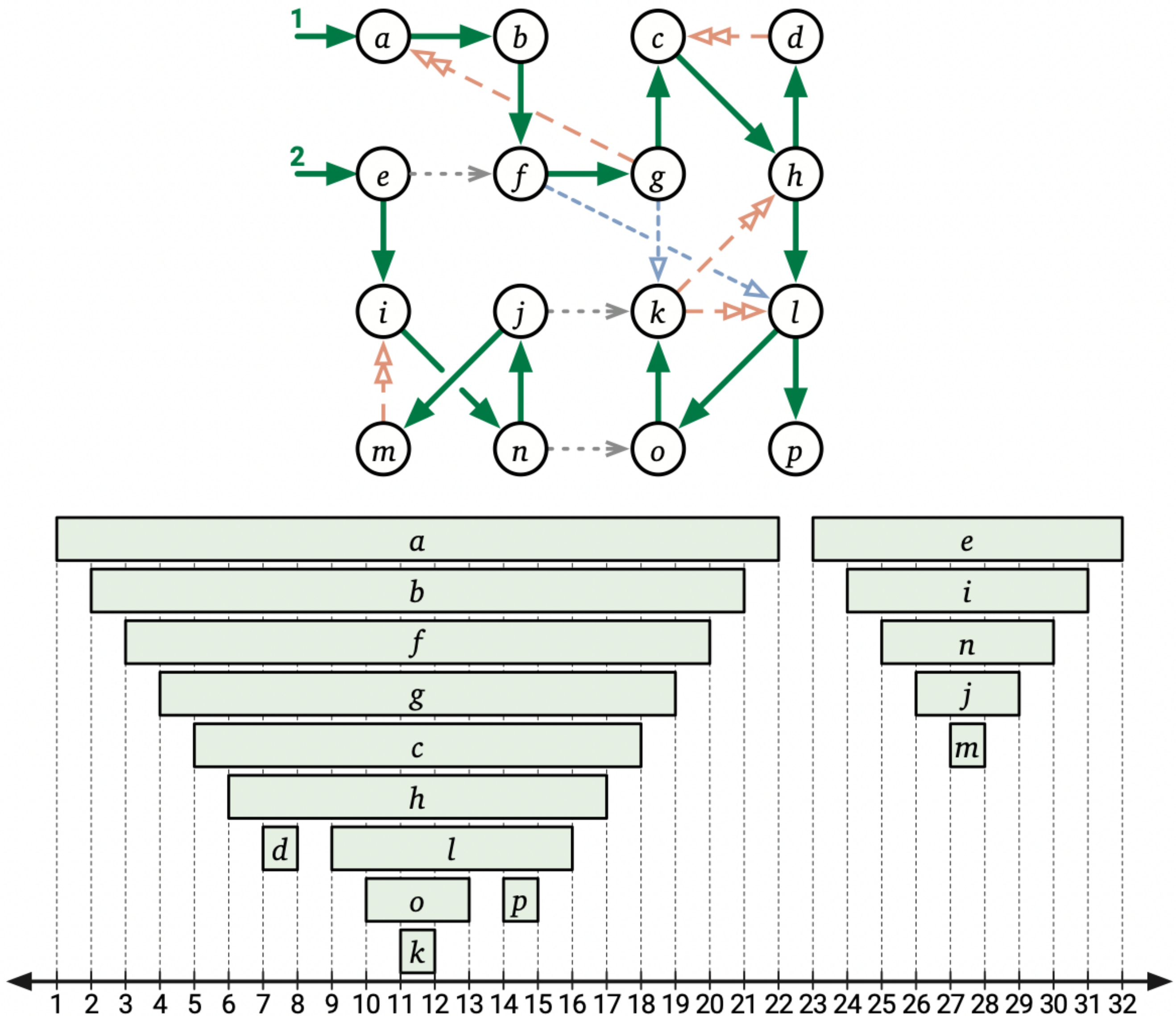
- v.pre: the time point that DFS "enters" node v
- v.post: the time point that DFS "exits" node v.

The timestamps delineates each node v into three states at any time:

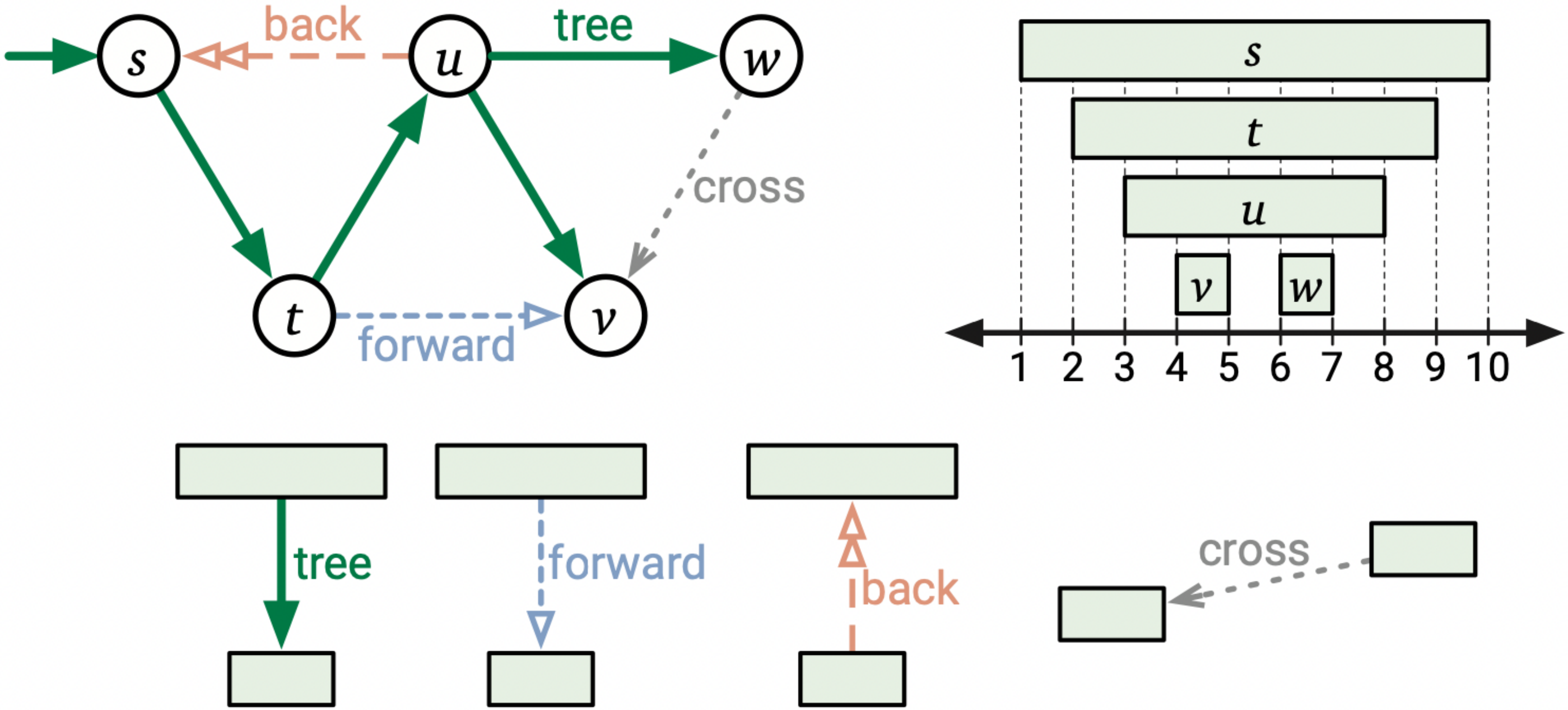
- **new**: $\text{clock} < v.\text{pre}$: DFS(v) has not yet been called.
- **active**: $v.\text{pre} \leqslant \text{clock} < v.\text{post}$ DFS has entered but not exited note v.
- **finished**: $v.\text{post} \leqslant \text{clock}$, DFS(v) has returned.

A couple of interesting properties of the pre/post time:

- A node v is active iff v is on the current recursion stack.
- For two nodes u, v, their clock intervals $[u.\text{pre}, u.\text{post}]$ and $[v.\text{pre}, v.\text{post}]$ must either be disjoint or nested. They cannot just overlap.
- $[u.\text{pre}, u.\text{post}] \subset [v.\text{pre}, v.\text{post}]$ means that v is descendent of u.
- disjoint $[u.\text{pre}, u.\text{post}]$ and $[v.\text{pre}, v.\text{post}]$ means u and v are not descendents of each other.



1
2
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32


back
tree
cross
forward
s
u
w
t
v
1 2 3 4 5 6 7 8 9 10
tree
forward
back
cross

The edges of the input graph falls into four different classes, depending on how there active intervals intersect. Fix edge $u \longrightarrow v$:

- If v is new when DFS(u) starts, then DFS(v) must be called *sometime* when u is active (u must be ancestor of v)

  i DFS(u) calls DFS(v) directly, in which case the edge $u \longrightarrow v$ is called **tree edge** (because it's in the DFS tree).

  ii Otherwise, $u \longrightarrow v$ is called **forward edge**.

- If v is active when DFS(u) starts, then v is already on the recursion stack, which implies $[u.\text{pre}, u.\text{post}] \subset [v.\text{pre}, v.\text{post}]$ (v is ancestor of u). Edge $u \longrightarrow v$ is called **back edge**.

- If v is finished when DFS(u) starts, we immediately have $[u.\text{pre}, u.\text{post}] \geqslant [v.\text{pre}, v.\text{post}]$ (disjoint interval; v is first). Edge $u \longrightarrow v$ is called **cross edge**.

The following statements are equivalent:

- u is an ancestor of v
- $[v.\text{pre}, v.\text{post}] \leqslant [u.\text{pre}, u.\text{post}]$ (or $v.\text{post} \leqslant u.\text{pre}$)
- Just after DFS(v) is called, u is active
- Just before DFS(u) is called, there's a path from u to v in which every vertices (including u, v) are new.

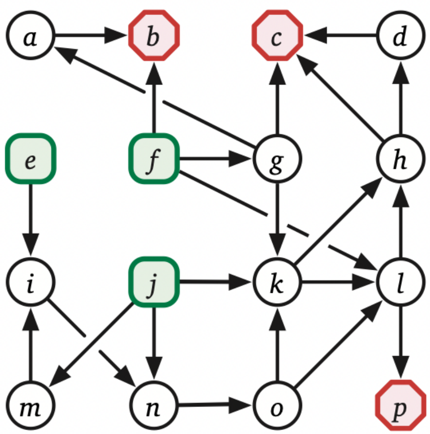
Sketch Proof:

Note: this is for directed graph. Undirected graph DFS tree does not have forward edge or cross-edge! It's very powerful precisely because it classifies edges into two classes: tree/back edges.

Question: why no forward/cross edges in undirected DFS?

Detecting Cycles

A DAG (directed acyclic graph) or dag is a directed graph with no cycles. Any vertex in a DAG with no incoming edges is called **source**; and vertex in a DAG with no outcoming edges is called **sink**. An isolated vertex (no incoming/outcoming edges) is both source and sink. A DAG must have at least one source and one sink, but can have more.



Green: sources

Red: sinks.

How do we detect cycles in a directed graph? The key idea is an observation:

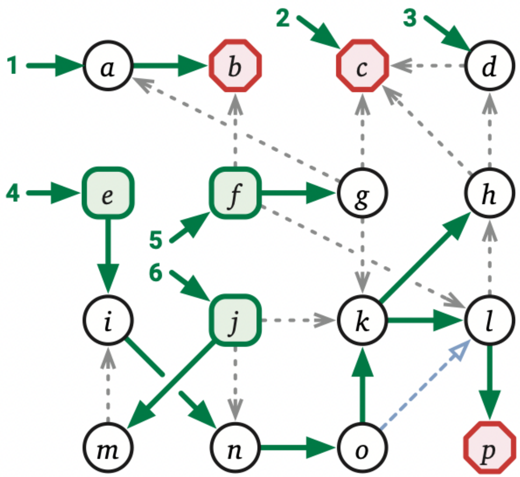
If and edge $u \rightarrow v$ but u finishes earlier than v ($u.\text{post} < v.\text{post}$), then there is directed path from v to u, which means cycle.

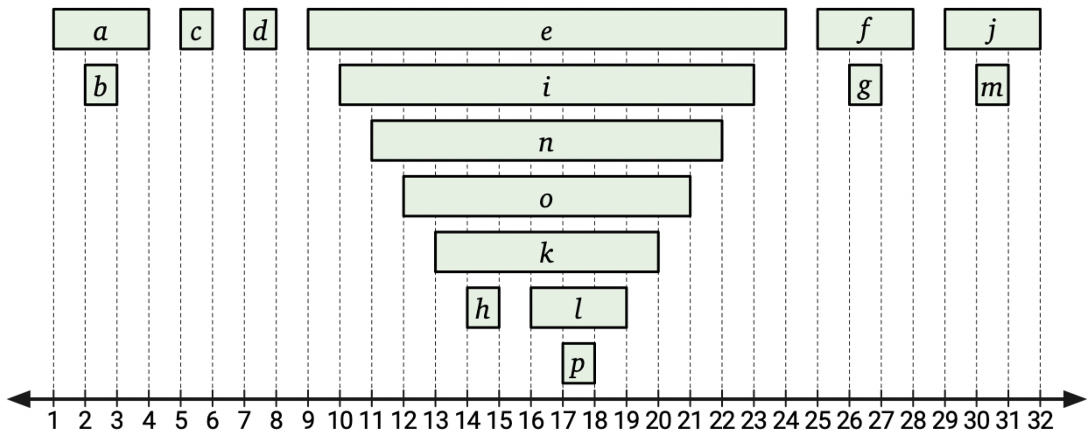
Why? Because if $u.\text{post} < v.\text{post}$ that means $u \rightarrow v$ is a **back edge**, therefore v is ancestor of u.

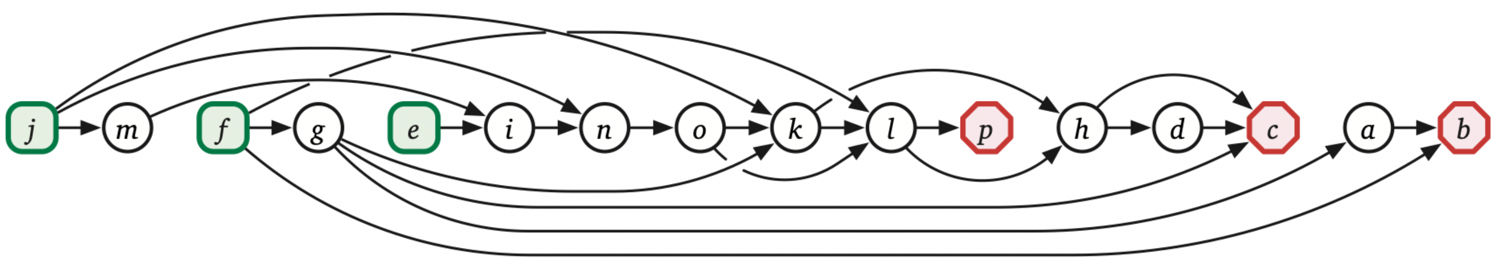
The reverse is true too; namely if we have cycle then we have back edge, which means some edge $u \rightarrow v$ with u finishes earlier. So we have

Detecting cycles $\Leftrightarrow$ detecting back edges in DFS

We can generate postorder of the nodes and look for condition $u \rightarrow v \wedge u.\text{post} < v.\text{post}$. Or we could simply embed the back edge detection logic into the DFS algorithm.







Topological ordering is an total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \rightarrow v$. Visually, topological ordering places nodes on a line such that no edges point from right to left.

Topological ordering is impossible for cyclic graph! (how do you place a cycle on a line without right-to-left edge?)

But for every acyclic directed graph we have topological ordering. For example, any **reversed** post-ordering is a valid topological ordering. Computing a topological ordering of an acyclic graph is called ***topological sorting***.

Why? Based on the analysis of cycle detection, we know that in a *acyclic* graph:

> for edge $u \to v$, we must have $u.\text{post} > v.\text{post}$

We could do topological sorting with DFS with post-order timestamp in $O(V + E)$ time. But typically an application visits the nodes in *implicit* topological order; in this case DFS with post-precessing is the right choice of tool:

```
PostProcessDFS(v):
    v.status ← active
    for each edge v → w
        if w.status = new
            PostProcessDFS(w)
        else if w.status = active
            report error: "cycles detected!"
    v.status ← finished
```

This algorithm does not need precise clocks, instead it only relies on the **new** $\rightarrow$ **active** $\rightarrow$ **finished** states of the nodes. Notice that this PostProcessDFS(v) processes nodes in topological order. Because of PostProcessDFS is so common, we can shorten the postorder processing of DAG as

```
PostProcessDAG(G):
    for all vertices v in postorder
        Process(v)
```

Topological Sort: Real Code


```
vector<vector<int>> adj;
int n, m; // #nodes, #edges
vector<bool> discovered;
vector<bool> processed;
vector<bool> parent;

vector<int> pre;
vector<int> post;
bool directed;
int clock;

enum EdgeClass {Unknown, TreeEdge, BackEdge, ForwardEdge, CrossEdge};
```

```
// topological sort a DAG. Report error if directed graph has cycle.
void topsort() {
    int n = adj.size(); // # nodes
    for (int i=0; i<n; i++) {
        if (discovered[i] == false)
            dfs(i);
    }
    reverse(top_sorted.begin(), top_sorted.end());
}
```

```cpp
void dfs(int v) {
    discovered[v] = true;
    clock++;
    pre[v] = clock;
    pre_visit(v);

    for (auto w : adj[v]) {
        if (!discovered[w]) { // new vertex
            parent[w] = v;
            process_edge(v,w);
            dfs(w);
        } else if ((!processed[w]) || directed) {
            // for undirected: only process back edge but not forward edge;
            // for directed; process every edge.
            process_edge(v,w);
        }
    }

    clock++;
    post[v] = clock;
    processed[v] = true;
    post_visit(v);
}
```

```
void pre_visit(int v) {

}
void post_visit(int v) {
    top_sorted.push_back(v);
}
void process_edge(int v, int w) {
    auto ec = edge_classify(v, w);
    if (ec == Graph::BackEdge) {
        cout << "Warning! directed cycle detected, not a DAG" << endl;
    }
}

Graph::EdgeClass edge_classify(int v, int w) {
    if (parent[w] == v) return Graph::TreeEdge;
    if (discovered[w] && !processed[w]) return Graph::BackEdge;
    if (processed[w] && pre[w] > pre[v]) return Graph::ForwardEdge;
    if (processed[w] && pre[w] < pre[v]) return Graph::CrossEdge;
    return Graph::Unknown;
}
```

Remember in dynamic programming, one of the important task is to identify the dependency graph of the subproblems, and find an evaluation order that repsects the dependency.

Recursion in dynamic programs gives us DAG, and such evaluation order we seek is precisely *reverse* topological ordering of the DAG!

```
MEMOIZE(x):
    if value[x] is undefined
        initialize value[x]

        for all subproblems y of x
            MEMOIZE(y)
            update value[x] based on value[y]
        finalize value[x]
```

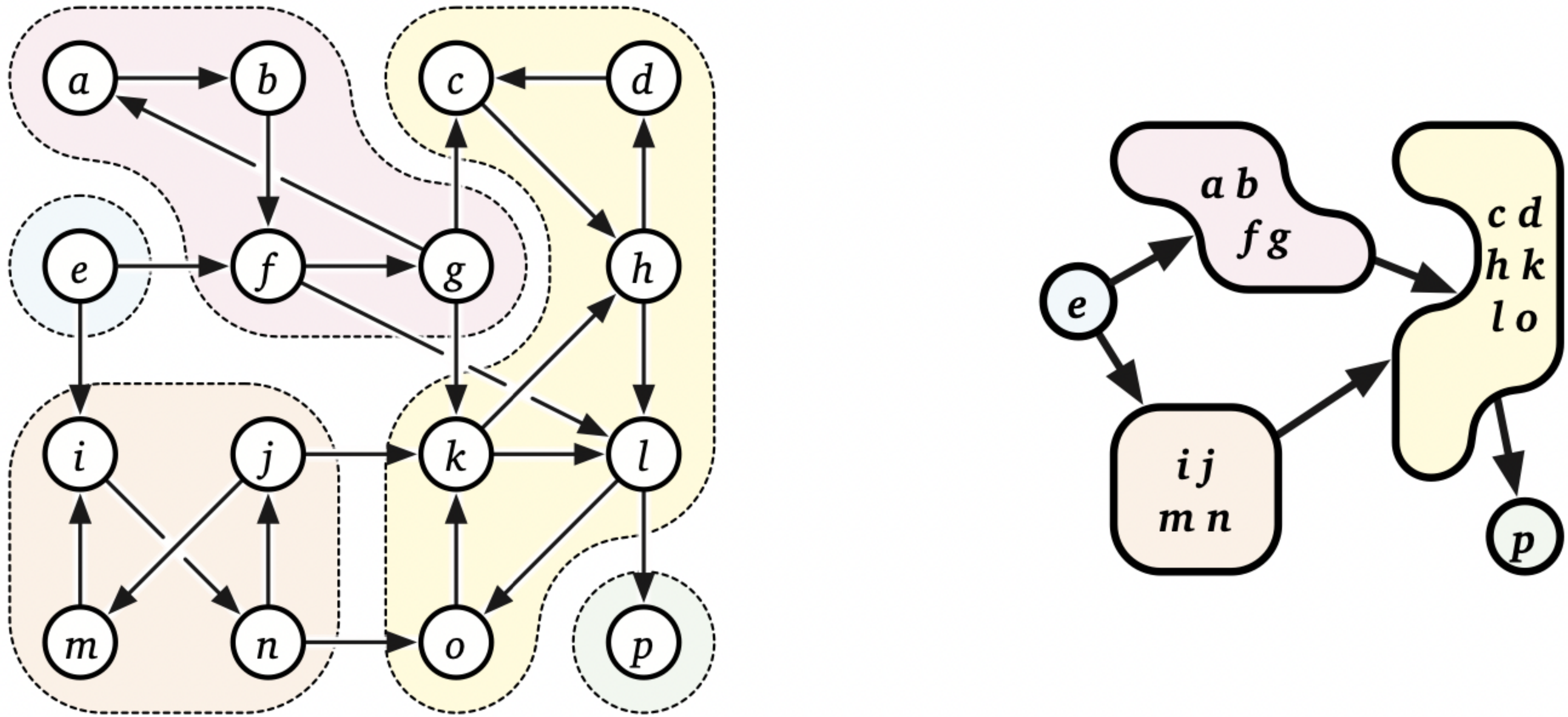
```
DFS(v):
    if v is unmarked
        mark v
        PREVISIT(x)
        for all edges v→w
            DFS(w)

        POSTVISIT(x)
```

Strong Connectivity

For directed graph, there's stronger version of connectivity called **strongly connected**. For two vertices u, v, if there is a directed path from u to v, and also a directed path from v to u, then we call the vertices u,v strongly connected.

Strong connectivity defines an equivalence relation over the nodes; the equivalence class is called the **strongly connected component** (SCC). A SCC of graph G is a maximal strongly connected subgraph. A directed graph is strongly connected iff it has exactly 1 SCC. At the other extreme, G is a DAG iff every SCC is single vertex.

The **strong component graph** scc(G) is another directed graph obtained by contracting each SCC of G into a single node and collapsing the parallel edges. the scc(G) is always a DAG.



Now how do we compute the SCC of a graph? We start with computing one SCC of a single node v.

First, we compute the $\text{reach}(v)$ by WhateverFirstSearch. Then we compute $\text{reach}^{-1}(v) = \{u\colon u \text{ can reach } v\}$ by searching the reversal of G: $\text{reach}^{-1}(v) = \text{reach}(v) \text{ in } \text{rev}(G)$. Finally, the SCC of v is the intersection $\mathsf{reach}(v) \cap \mathsf{reach}^{-1}(v)$. This takes linear time $O(V+E)$.

To compute all the SCCs, we can wrap the SCC of single node around a outer loops that computes SCC of every potential node. However, the resulting algorithm runs in $O(VE)$ time, instead of linear. (Why? There are at most $O(V)$ SCCs, and each one needs $O(E)$ time to discover), even if the graph is a DAG! We can do better.

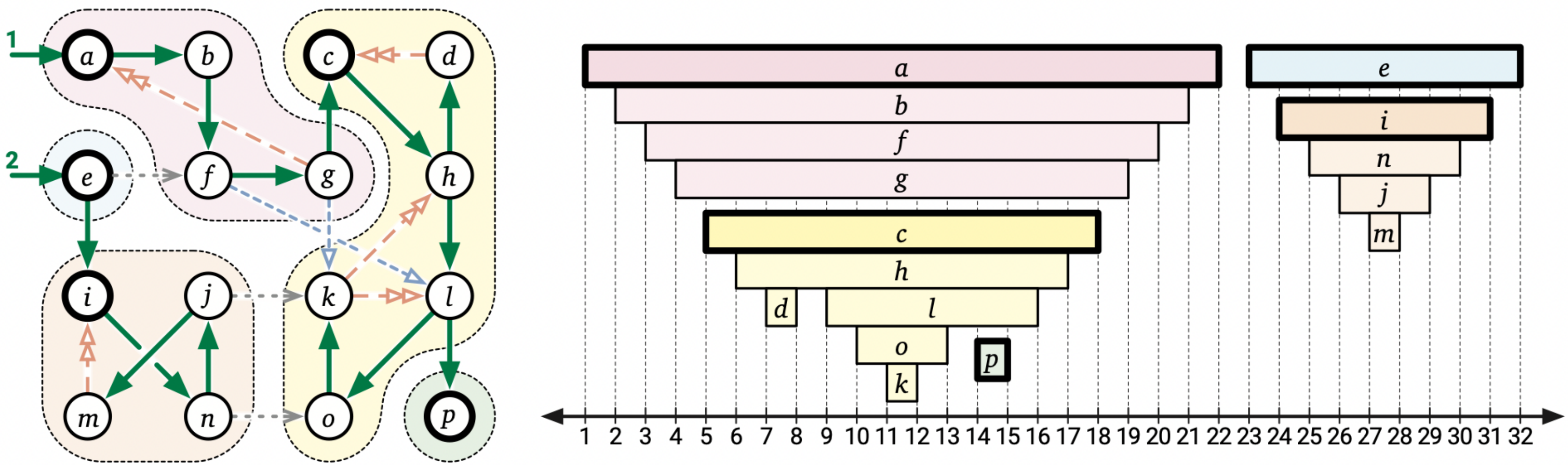
All linear algorithm of finding all SCC rely on the observation:

Fix **DFS traversal** of directed graph G. Each strong component C contains **exactly** one node which does not have parent in C.

Why?

- Consider a strong component C of G and *any* directed path from one vertex $v \in C$ to another $w \in C$. The whole path must belong to C (why?)
- Let v be the vertex in C with the earliest starting (entering) time, v.pre. Then v has no parent in C.

- (uniqueness). Suppose *w* is another vertex. Just before DFS(v), every vertex in *C* is new, so there's path of new vertices from v to w. Thus *w* must have parent in C.



The observation implies that each strong component defines a connected subtree of any depth-first forest. In particular, the node in *C* with the earliest starting time is the **root** of *C*. And strong components are contiguous in the depth-first forest. Now we are ready to describe a linear time algorithm!

```
StrongComponents(G):
    count ← 0
    while G is non-empty
        C ← ∅
        count ← count + 1
        v ← any vertex in a sink component of G     #magic!
        for all vertices w in reach(v)
            w.label ← count                          #we found a SCC
            add w to C
        remove C and its incoming edges from C.
```

This algorithm works by identifying a **sink component** (sink in the scc(G)). The sink component can only reach itself! We compute its reach, and remove it from our graph, and recurse. Gradually we find all the strong components one by one.

But how do we find any vertex in a **sink component** of G?

Finding vertex in sink component does not seem easy; but finding vertex in a source component is easy enough. In fact:

The last vertex of any post-ordering of G lies in a source component of G.

(why?)

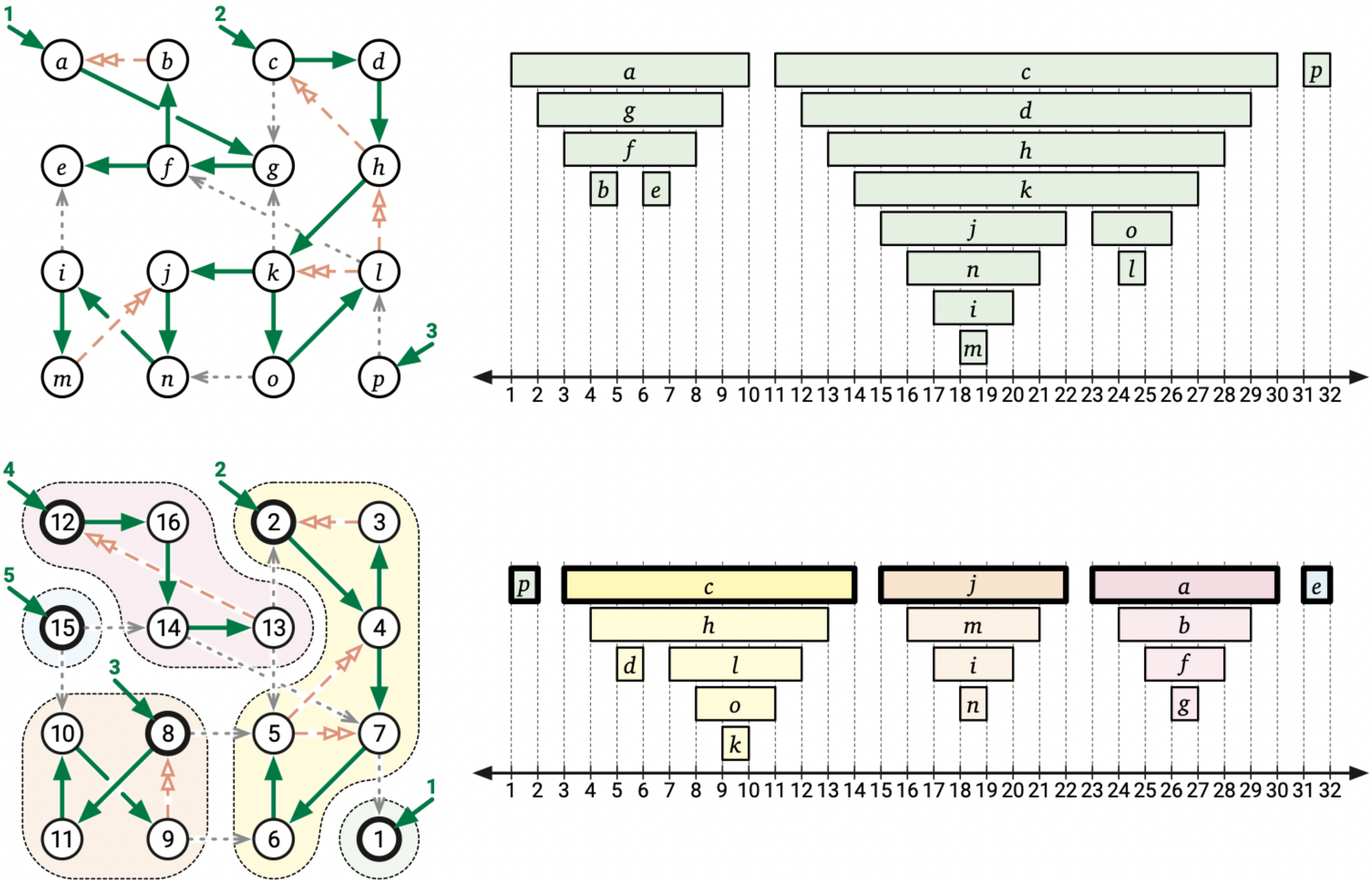
Noting that post-ordering of rev(G) gives us souce component in rev(G), which is sink component in G! So we do two passes:

- First, DFS traverse rev(G) and record post-order.
- DFS Traverse G, remove the sink components one at a time.

```
KosarajuSharir(G):
    S ← new empty stack
    for all vertices v
        unmark v
        v.root ← None
    ⟨⟨Phase 1: Push in postorder in rev(G)⟩⟩
    for all vertices v
        if v is unmarked
            PushPostRevDFS(v, S)
    ⟨⟨Phase 2: DFS again in stack order⟩⟩
    while S is non-empty
        v ← Pop(S)
        if v.root = None
            LabelOneDFS(v, v)
```

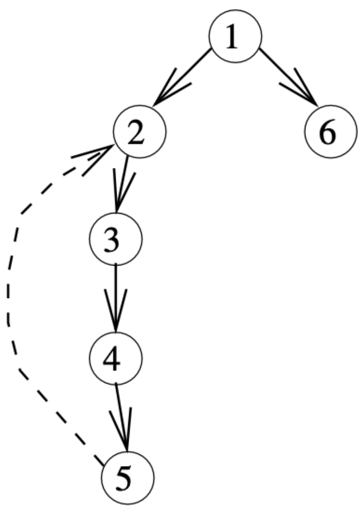
```
PushPostRevDFS(v, S):
    mark v
    for each edge u→v    ⟨⟨Reversed!⟩⟩
        if u is unmarked
            PushPostRevDFS(u, S)
    Push(v, S)
```

```
LabelOneDFS(v, r):
    v.root ← r
    for each edge v→w
        if w.root = None
            LabelOneDFS(w, r)
```



1
2
3
a b c d
e f g h
i j k l
m n o p
a c p
g d
f h
b e k
j o
n l
i
m
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
4 2 5 3 1
12 16 2 3
15 14 13 4
10 8 5 7
11 9 6 1
p c j a e
h m b
d l i f
o n g
k
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

In a connected undirected graph, a node is called cut-node is called **cut-node** or **articulation vertex**, iff removing the node disconnects the graph.



In the left graph we have the **DFS tree** of an undirected connected graph.

node 1,2 are cut-node; other nodes are not.

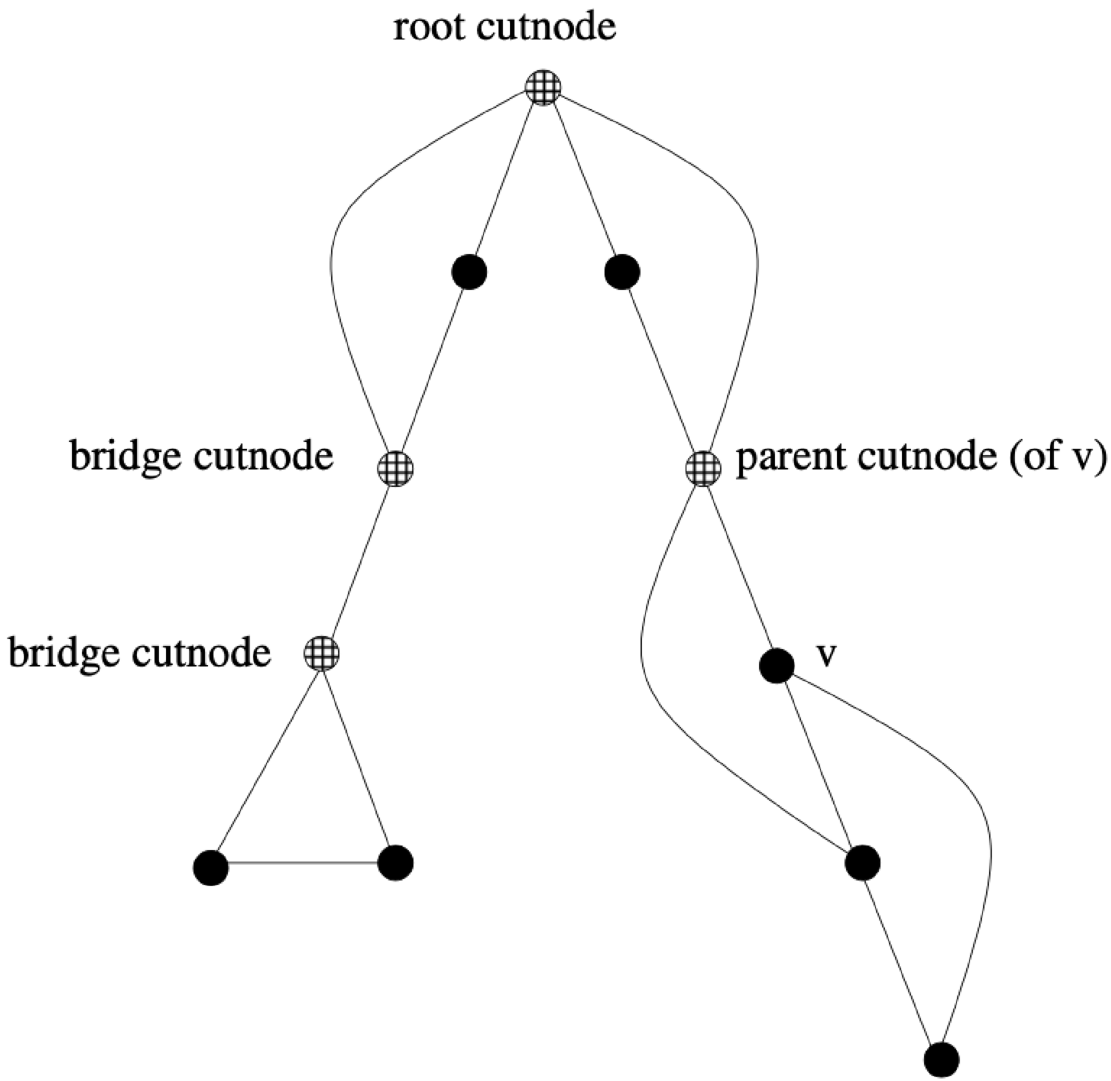
How do we find out which nodes are cut-nodes?

Clearly, the back edges ($5 \longrightarrow 2$) play special role, because it makes 3,4 NOT cut-node.

The back edge inspires us to compute something called `earliest_reach[v]` for every vertex, which means the earliers (in terms of entering time u.pre) vertex that v can reach through tree edges and back edges. E.g. `earliest_reach[4]`=2, because 4 can reach 2 through 5, and that's as early as it can go.

Why do we care about `earliest_reach[v]`? Because it tells exactly what nodes are cut-nodes. Several observations:

- Root is cut-node, if it has $\geqslant 2$ children. (in DFS tree of undirected graph, there is no cross-edge, or forward edge).
- Leaves are always not cut-nodes (the spanning tree is still connected, therefore the graph G is also connected)
- If earliest_reach[v]=v, then both v and parent(v) are cut nodes
- If earliest_reach[v]=parent(v), then parent(v) is cut node.
- If earliest_reach[v]<parent(v), then v,parent(v) are not cutnode.


root cutnode
bridge cutnode
parent cutnode (of v)
bridge cutnode
v

To put the observation in algorithm, we must know the following information along the DFS tree.

- earliest_reach[v]: the earliest reachable ancestor of the whole subtree[v].
- is_root[v]: whether v is the root
- is_leaf[v]: whether v is a leaf

Let's put them together into C++ program: `cutnode.cpp`