APA: Advanced Programming, Algorithms and Data Structures

Emre Güney, PhD

Master in Bioinformatics for Health Sciences
Universitat Pompeu Fabra

Lectures 7-8

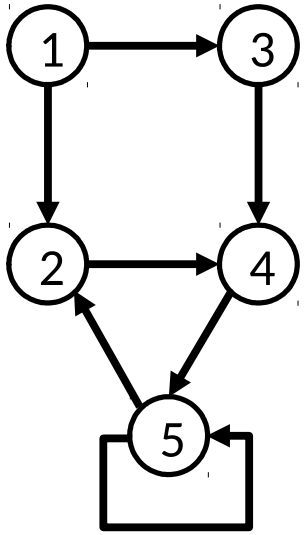
October 15-22, 2019

Previously on APA

- Computational complexity
  - Correctness of an algorithm
  - Efficiency of an algorithm
  - Time complexity in terms of input size
  - Space complexity
- Sorting
  - Buble sort, selection sort, insertion sort, merge sort
  - Exhaustive search (brute force) vs divide & conquer
  - Quick sort
  - Heap sort
- Data structures
  - Linked lists, stacks, queues
  - Priority queues and heaps
  - Trees, binary search trees
  - Balanced trees (AVL trees)
- Tree traversals
  - Preorder, postorder, inorder

Graph definition

A graph is specified by a set of vertices (or nodes) V and a set of edges E.



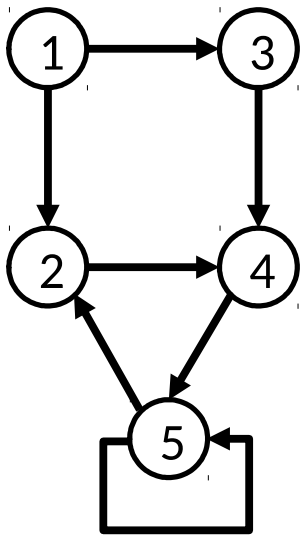
$$V = \{1,2,3,4,5\}$$

$$E = \{(1,2), (1,3), (2,4), (3,4), (4,5), (5,2), (5,5)\}$$

Graphs can be directed or undirected.
Undirected graphs have a symmetric relation.

Graph representation: adjacency matrix

A graph with $n = |V|$ vertices $v_1,\ldots,v_n$, can be represented by an $n \times n$ matrix with:



$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

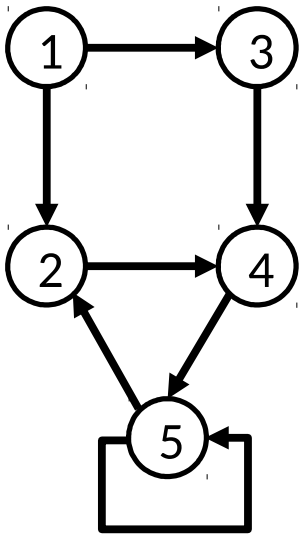
$$a = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

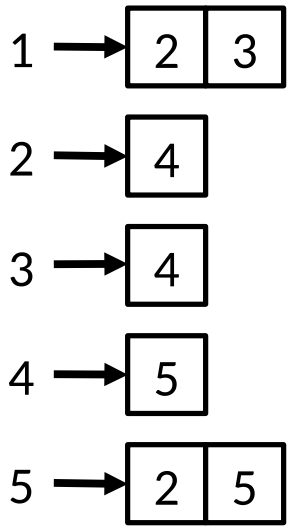
Space: $O(n^2)$

Undirected graphs: the matrix is symmetric.

Graph representation: adjacency list

A graph can be represented by $|V|$ lists, one per vertex. The list for vertex u holds the vertices connected to the outgoing edges from u.





The lists can be implemented in different ways (vectors, linked lists, ...)

Space: $O(|E|)$

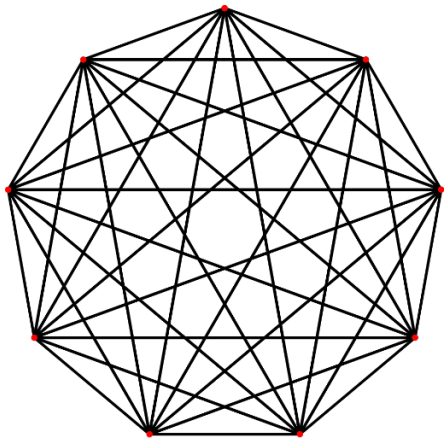
Undirected graphs: use bi-directional edges

Dense and sparse graphs

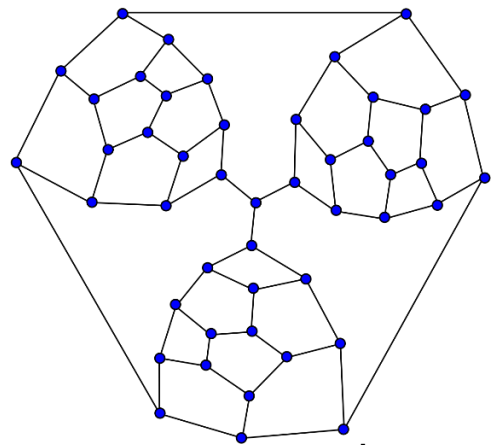
A graph with |V| vertices could potentially have up to $|V|^2$ edges (all possible edges are possible).

We say that a graph is **dense** when |E| is close to $|V|^2$. We say that a graph is **sparse** when |E| is close to |V|

How big can a graph be?



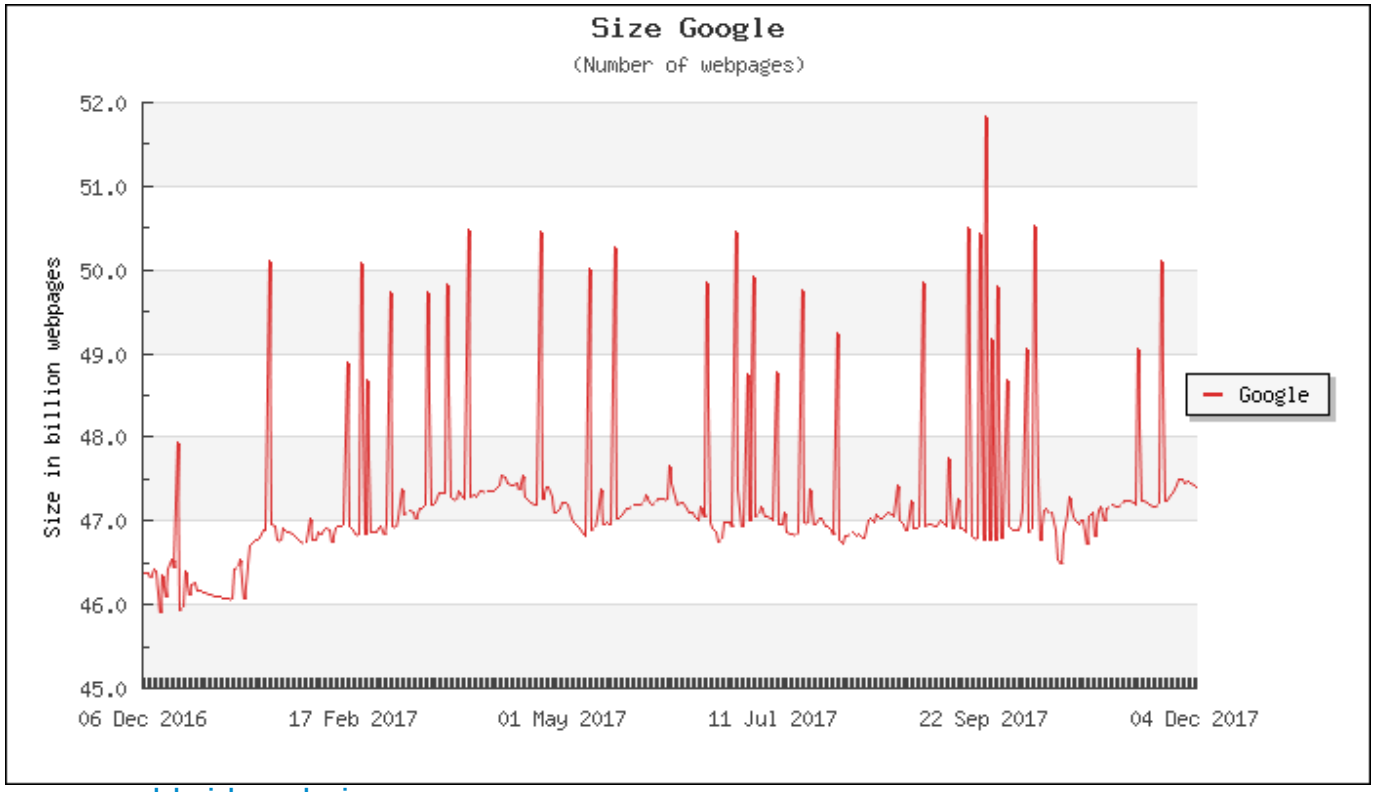
Dense graph



Sparse graph

Source: UPC - J. Cortadella & J. Petit

Size of the World Wide Web



www.worldwidewebsize.com

- December 2017: 50 billion web pages (50×10^9).
- Size of adjacency matrix: 25×10^{20} elements.
  (not enough computer memory in the world to store it).
- Good news: The web is very sparse. Each web page has about half a dozen hyperlinks to other web pages.

Source: UPC - J. Cortadella & J. Petit

Adjacency matrix vs. adjacency list

- Space:

  Adjacency matrix is $O(|V|^2)$

  Adjacency list is $O(|E|)$

- Checking the presence of a particular edge (u,v):

  Adjacency matrix: constant time

  Adjacency list: traverse u's adjacency list

- Which one to use?

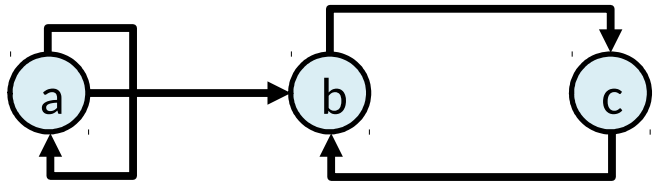
  For dense graphs: adjacency matrix

  For sparse graphs: adjacency list

- For many algorithms, traversing the adjacency list is not a problem, since they require to iterate through all neighbors of each vertex. For sparse graphs, the adjacency lists are usually short (can be traversed in constant time)

Graph usage: example

```
# Declaration of a graph that stores
# a string (name) for each vertex
G = Graph()

# Create the vertices
a = G.addVertex("a")
b = G.addVertex("b")
c = G.addVertex("c")

# Create the edges
G.addEdge(a,a)
G.addEdge(a,b)
G.addEdge(b,c)
G.addEdge(c,b)
```



	info	succ	pred
0	"a"	{0,1}	{0}
1	"b"	{2}	{0,2}
2	"c"	{1}	{1}

```
# Print all edges of the graph
for src in range(G.numVertices()): # all vertices
    for dst in G.succ(src):        # all successors of src
        print(G.info(src), " -> ", G.info(dst))
```


Source: UPC - J. Cortadella & J. Petit

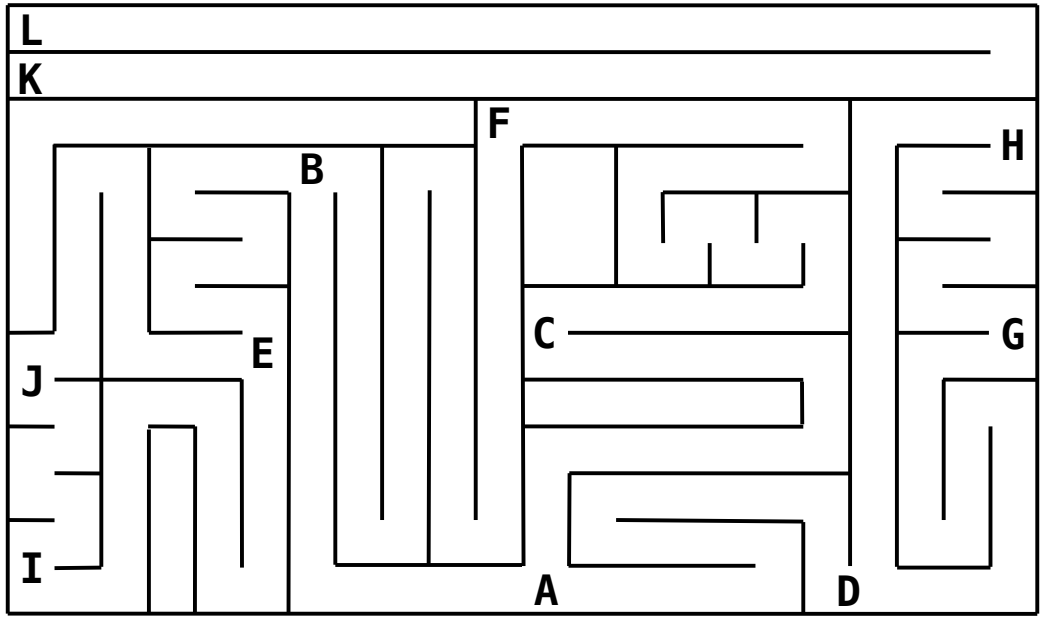
Graph implementation

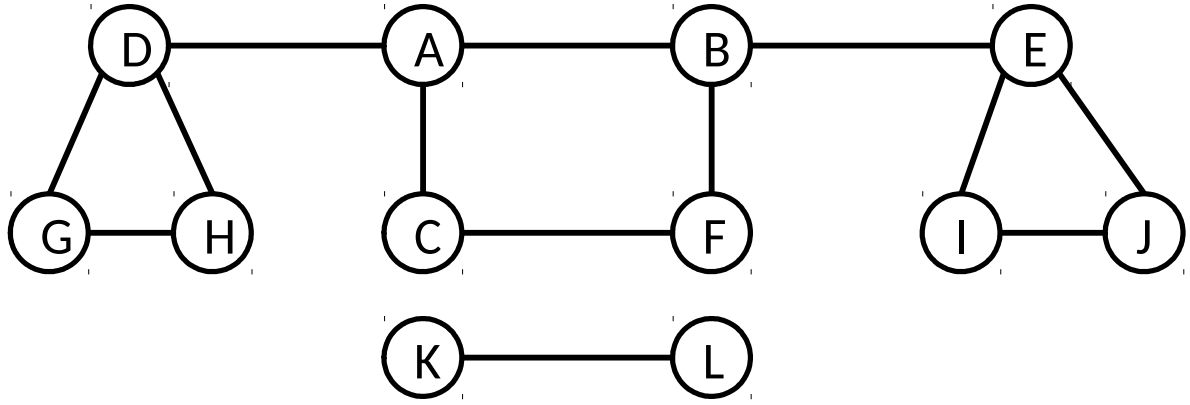
```
class Graph:
    # -------- private _Vertex class -----------
    class _Vertex:
        def __init__(self, info):
            self._info = info # Information of the vertex
            self._succ = []   # List of successors
            self._pred = []   # List of predecessors

    # -------- public Graph methods -----------
    def __init__(self):
        self.__vertices = []  # List of vertices

    # Adds a vertex with information associated to
    # the vertex. Returns the index of the vertex
    def addVertex(self, info):
        self.__vertices.append(self._Vertex(info))
        return len(self.__vertices) - 1;
```

Graph implementation

```
# Adds an edge src -> dst
def addEdge(self, src, dst):
    self.__vertices[src]._succ.append(dst)
    self.__vertices[dst]._pred.append(src)

# Returns the number of vertices of the graph
def numVertices(self):
    return len(self.__vertices)

# Returns the information associated to vertex v
def info(self, v):
    return self.__vertices[v]._info

# Returns the list of successors of vertex v
def succ(self, v):
    return self.__vertices[v]._succ

# Returns the list of predecessors of vertex v
def pred(self, v):
    return self.__vertices[v]._pred
```

Reachability: exploring a maze

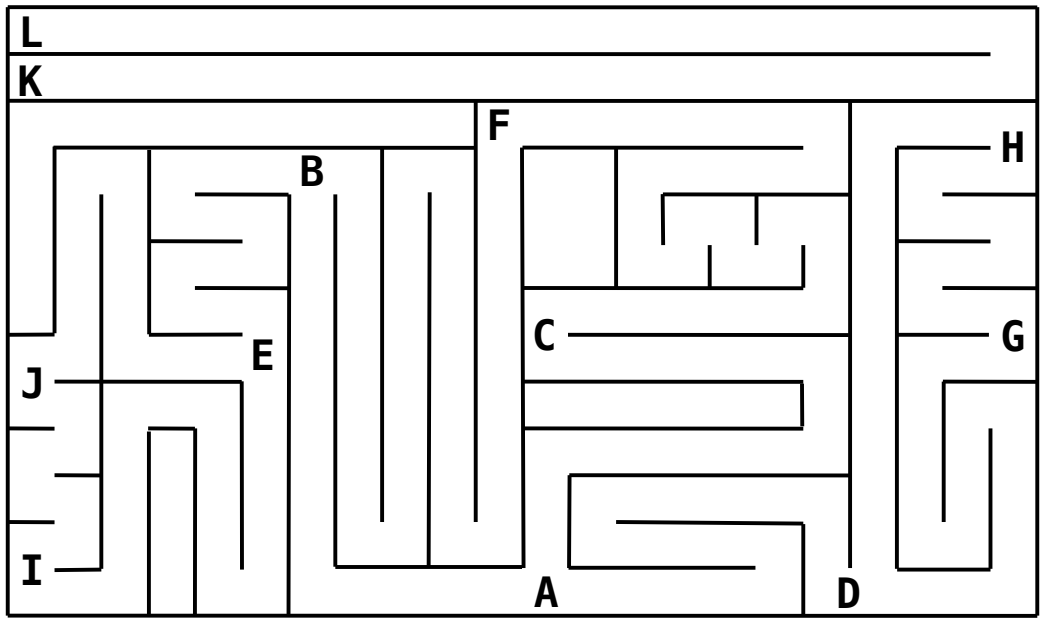




Which vertices of the graph are reachable from a given vertex?

Source: UPC - J. Cortadella & J. Petit

Reachability: exploring a maze



To explore a labyrinth we need a ball of string and a piece of chalk:

- The chalk prevents looping, by marking the visited junctions.
- The string allows you to go back to the starting place and visit routes that were not previously explored.

Source: UPC - J. Cortadella & J. Petit

Graph Traversal

What makes it different from tree traversals:

- you can visit the same node more than once
- you can get in a cycle

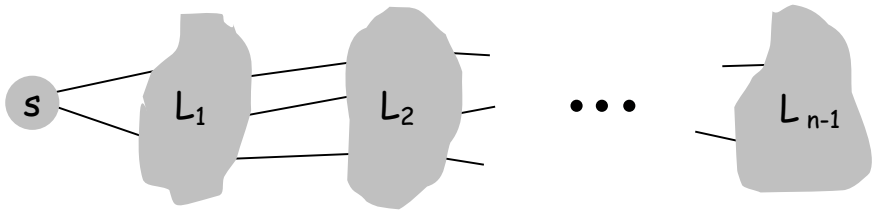
What to do about it:

- **mark** the nodes
  - -White: unvisited
  - -Grey: (still being considered) on the frontier: not all adjacent nodes have been visited yet
  - -Black: off the frontier: all adjacent nodes visited (not considered anymore)

Source: ColoState - Wim Bohm

Breadth First Search

BFS intuition. Explore outward from s, adding nodes one "layer" at a time.



BFS algorithm.

- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

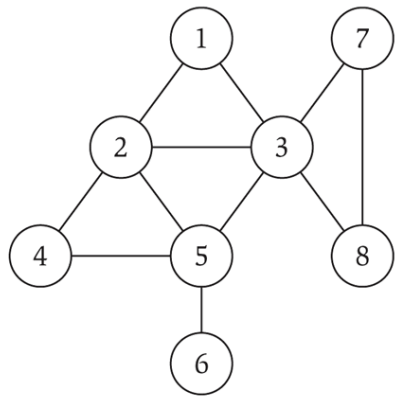
For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.

Source: ColoState - Wim Bohm

Breadth First Tree

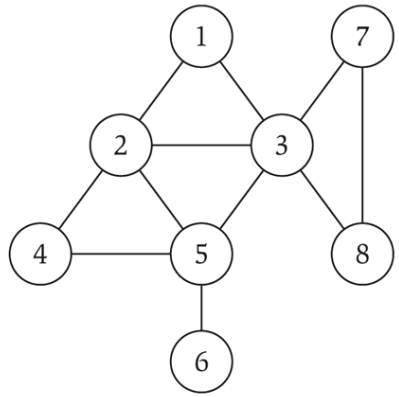
BFS produces a **Breadth First (spanning) Tree** rooted at s: when a node v in L_{i+1} is discovered as a neighbor of node u in L_i we add edge (u,v) to the BF tree

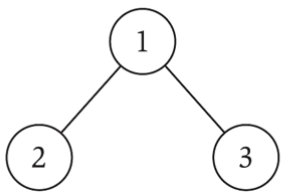
Property. Let T be a BFS tree of G, and let (x, y) be an edge of G. Then the level of x and y differ by at most 1. **WHY?**



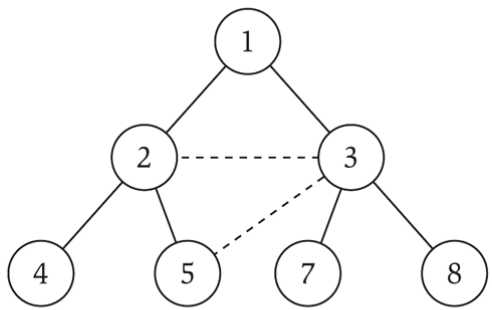
Source: ColoState - Wim Bohm

Breadth First Search

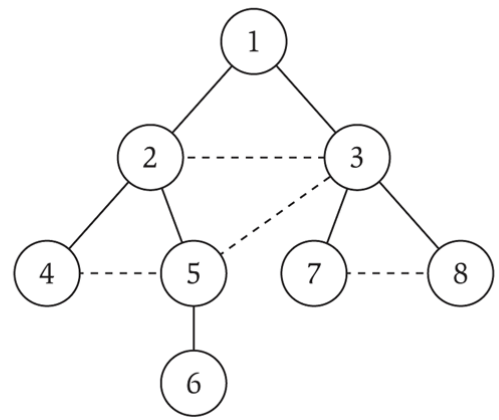




(a)



(b)



(c)

L_0

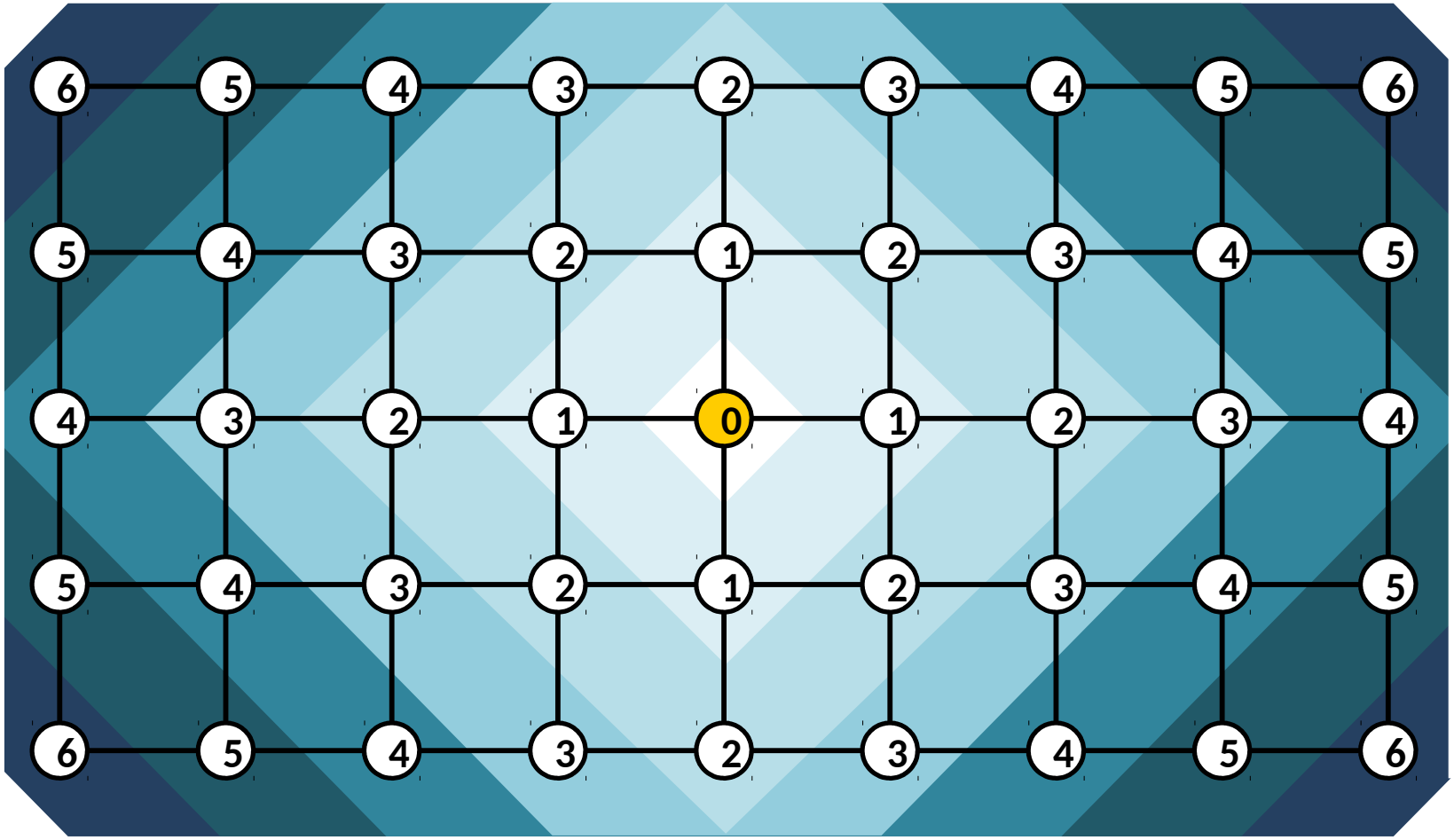
L_1

L_2

L_3

Source: ColoState - Wim Bohm

Breadth-first search



Source: UPC - J. Cortadella & J. Petit

BFS algorithm

- BFS visits vertices layer by layer: 0,1,2,...,*d* .
- Once the vertices at layer *d* have been visited, start visiting vertices at layer *d*+1 .
- Algorithm with two active layers:
  - Vertices at layer *d* (currently being visited).
  - Vertices at layer *d*+1 (to be visited next).
- Central data structure: a queue.

```
BFS(G,s)
  #d: distance,  c: color,  p: parent in tree
  forall v in V-s {c[v]=white; d[v]=∞,p[v]=nil}
  c[s]=grey; d[s]=0; p[s]=nil;
  Q=empty;
  enque(Q,s);
  while (Q != empty)
     u = deque(Q);
      forall  v in adj(u)
         if (c[v]==white)
            c[v]=grey; d[v]=d[u]+1; p[v]=u;
            enque(Q,v)
     c[u]=black;
   # don't really need grey here, why?
```

Source: ColoState - Wim Bohm

Complexity BFS

Each node is painted white once, and is enqueued and dequeued at most once.

Enque and deque take constant time. The adjacency list of each node is scanned only once, when it is dequeued.
Therefore time complexity for BFS is

$$O(|V|+|E|) \text{ or } O(n+m)$$

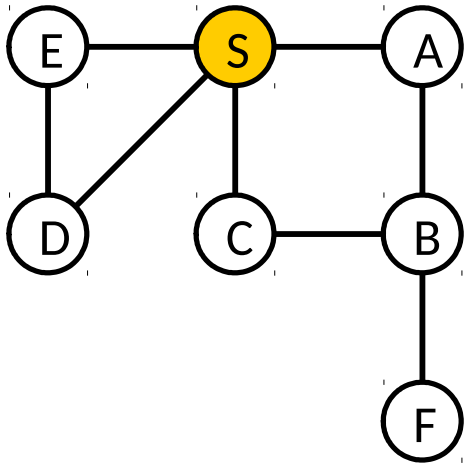
Source: ColoState - Wim Bohm

BFS algorithm

```
def BFS(G, s):      # Pseudo-Python!
    # Input: Graph G(V, E), source vertex s.
    # Output: For each vertex u, dist[u] is
    #         the distance from s to u.
    for all u € V: dist[u]=∞
    dist[s] = 0
    # Queue containing just s
    q = Queue(); q.enqueue(s)
    while not q.empty():
        u = q.first(); q.dequeue()
        for all (u, v) € E:
            if dist[v] == ∞ :
                dist[v] = dist[u] + 1
                q.enqueue(v)
    return dist
```

Runtime $\mathcal{O}(|V| + |E|)$: Each vertex is visited once, each edge is visited once (for directed graphs) or twice (for undirected graphs).

BFS algorithm



		S	A	B	C	D	E	F
	S_0	0	∞	∞	∞	∞	∞	∞
S_0	$A_1\ C_1\ D_1\ E_1$	0	1	∞	1	1	1	∞
A_1	$C_1\ D_1\ E_1\ B_2$	0	1	2	1	1	1	∞
C_1	$D_1\ E_1\ B_2$	0	1	2	1	1	1	∞
D_1	$E_1\ B_2$	0	1	2	1	1	1	∞
E_1	B_2	0	1	2	1	1	1	∞
B_2	F_3	0	1	2	1	1	1	3
F_3		0	1	2	1	1	1	3

Source: UPC - J. Cortadella & J. Petit

DFS: Depth First Search

Explores edges from the most recently discovered node; backtracks when reaching a dead-end. The book does not used white, grey, black, but uses explored (and implicitly unexplored). Recursive code:

```
DFS(u):
  mark u as Explored and add u to R
  for each edge (u,v) :
    if v is not marked Explored :
       DFS(v)
```

Source: ColoState - Wim Bohm

Recursive / node coloring version

```
DFS(u):
    #c: color,  p: parent
    c[u]=grey
    forall  v in Adj(u):
      if c[v]==white:
        p[v]=u
        DFS(v)
    c[u]=black
```

The above implementation of DFS runs in $O(m + n)$ time if the graph is given by its adjacency list representation.

Proof:

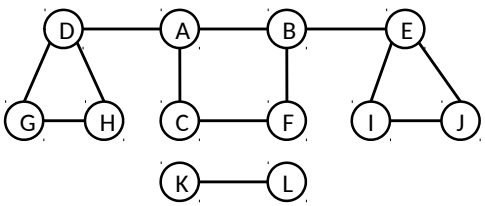
Same as in BFS ·

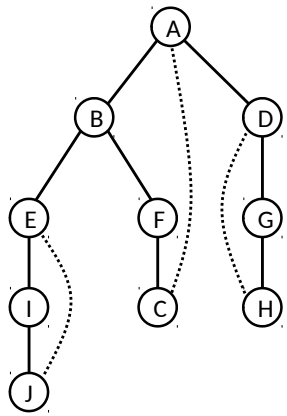
Source: ColoState - Wim Bohm

DFS: Python-like code and complexity

Finding the nodes reachable from another node

```
def explore(G, v):
    visited(v) = true
    for each edge (v,u) ∈ E:
        if not visited(u):
            explore(G, u)
```





Dotted edges are ignored (*back edges*): they lead to previously visited vertices.
The solid edges (*tree edges*) form a tree.

Depth-first search

```
def DFS(G):
    for all v ∈ V:
        visited(v) = false
    for all v ∈ V:
        if not visited(v):
            explore(G,v)
```

DFS traverses the entire graph.

Complexity:

- Each vertex is visited only once (thanks to the chalk marks)
- For each vertex:
  - A fixed amount of work (pre/postvisit)
  - All adjacent edges are scanned

Running time is $\boldsymbol{O}(|\mathrm{V}| + |\mathrm{E}|)$.
Difficult to improve: reading a graph already takes $\boldsymbol{O}(|\mathrm{V}| + |\mathrm{E}|)$.

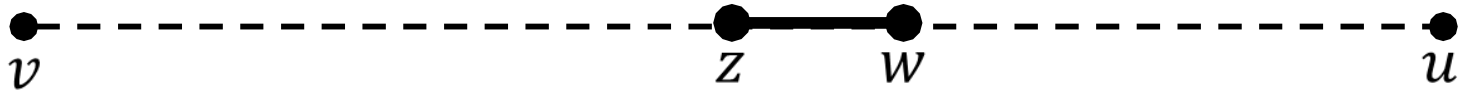
Source: UPC - J. Cortadella & J. Petit

Finding the nodes reachable from another node

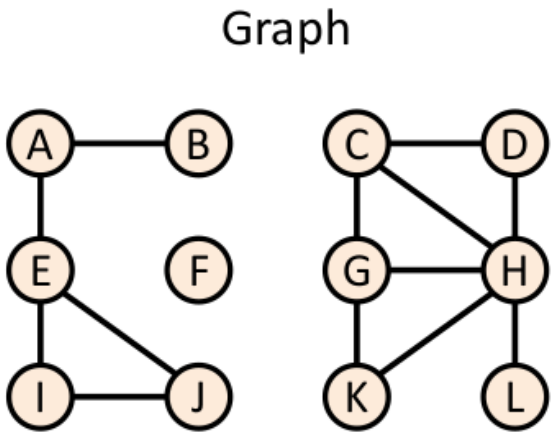
```
def explore(G, v):
    visited(v) = true
    for each edge (v,u) ∈ E:
        if not visited(u):
            explore(G, u)
```

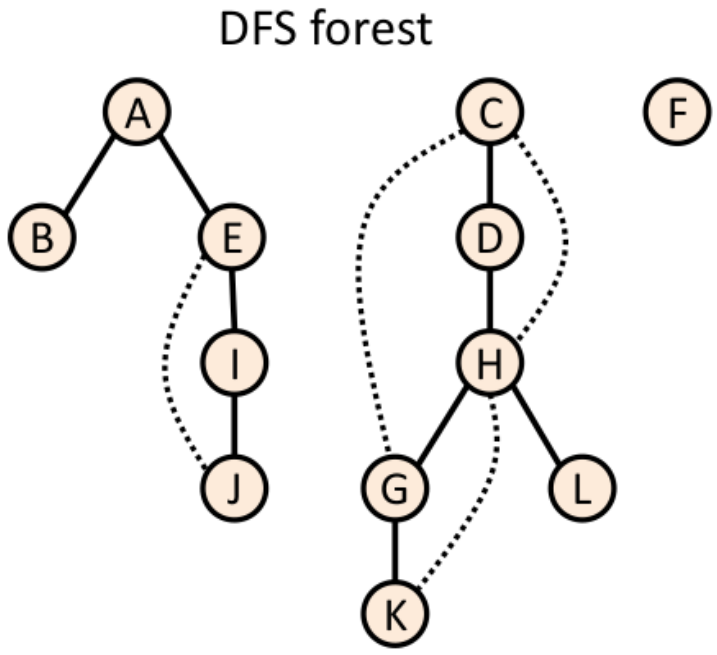
- All visited nodes are reachable because the algorithm only moves to neighbors and cannot jump to an unreachable region.

- Does it miss any reachable vertex? No. Proof by contradiction.
  - Assume that a vertex u is missed.
  - Take any path from v to u and identify the last vertex that was visited on that path (z). Let w be the following node on the same path. Contradiction: w should have also been visited.



DFS example





- The outer loop of DFS calls ***explore*** three times (for A, C and F)
- Three trees are generated. They constitute a *forest*.

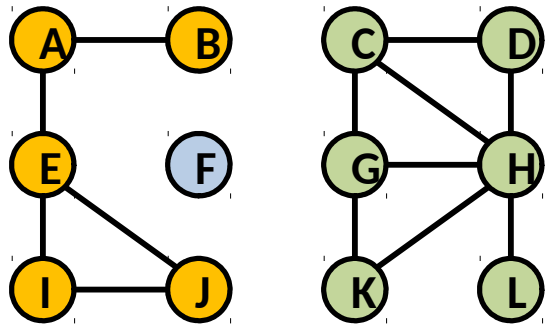
Source: UPC - J. Cortadella & J. Petit

Connectivity

- An undirected graph is connected if there is a path between any pair of vertices.

- A disconnected graph has disjoint *connected components*.

- Example: this graph has 3 connected components:



{A, B, E, I, J} {C, D, G, H, K, L} {F}

Connected Components

```
def explore(G, v, cc):
    # Input: G=(V,E) is a graph, cc is a CC number
    # Output:ccnum[u] = cc for each vertex u in the same CC as v
    ccnum[v] = cc
    for each edge (v,u) ∈ E:
        if ccnum[u] ≠ cc: explore(G, u, cc)

def ConnComp(G):
    # Input: G=(V,E) is a graph
    # Output: Every vertex v has a CC number in ccnum[v]
    for all v ∈ V: ccnum[v] = 0 # Clean cc numbers
    cc = 1   # Identifier of the first CC
    for all v ∈ V:
        if ccnum[v] == 0:   # A new CC starts
            explore(G, v, cc); cc += 1
```

- Performs a DFS traversal assigning a CC number to each vertex.
- The outer loop of **ConnComp** determines the number of CC's.
- The variable ccnum[v] also plays the role of visited[v].

Revisiting the explore function

```
def explore(G, v):
    visited(v) = True
    previsit(v)
    for each edge (v,u) ∈ E:
        if not visited(u):
            explore(G, u)
    postvisit(v)
```

Let us consider a **global variable clock** that can determine the occurrence times of previsit and postvisit.

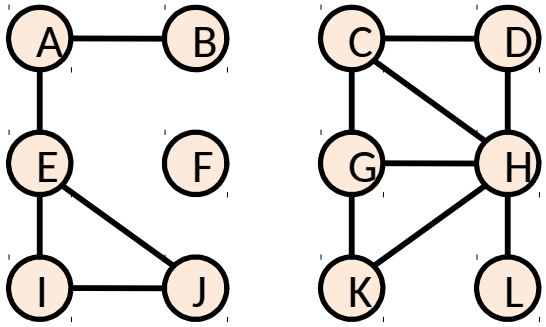
```
def previsit(v):
    pre[v] = clock
    clock += 1

def postvisit(v):
    post[v] = clock
    clock += 1
```

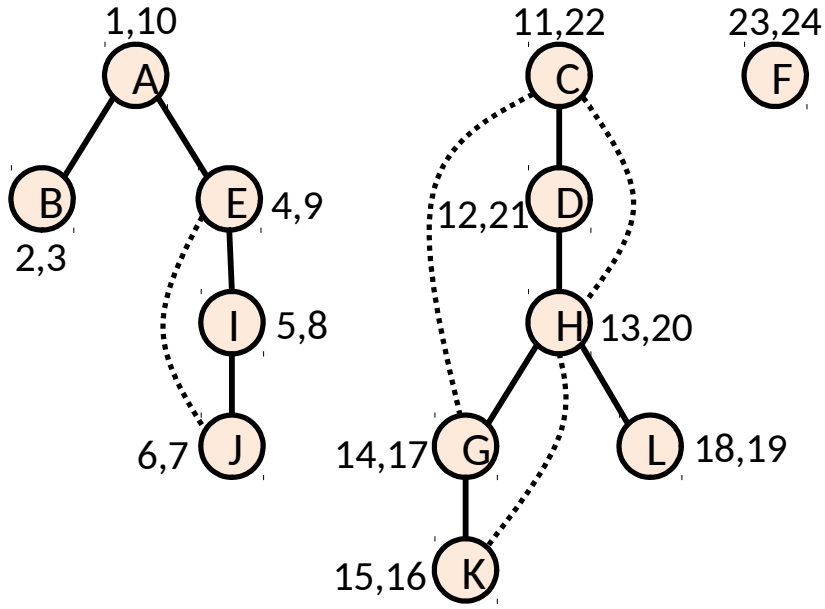
Every node v will have an interval (pre[v], post[v]) that will indicate the time the node was first visited (pre) and the time of departure from the exploration (post).

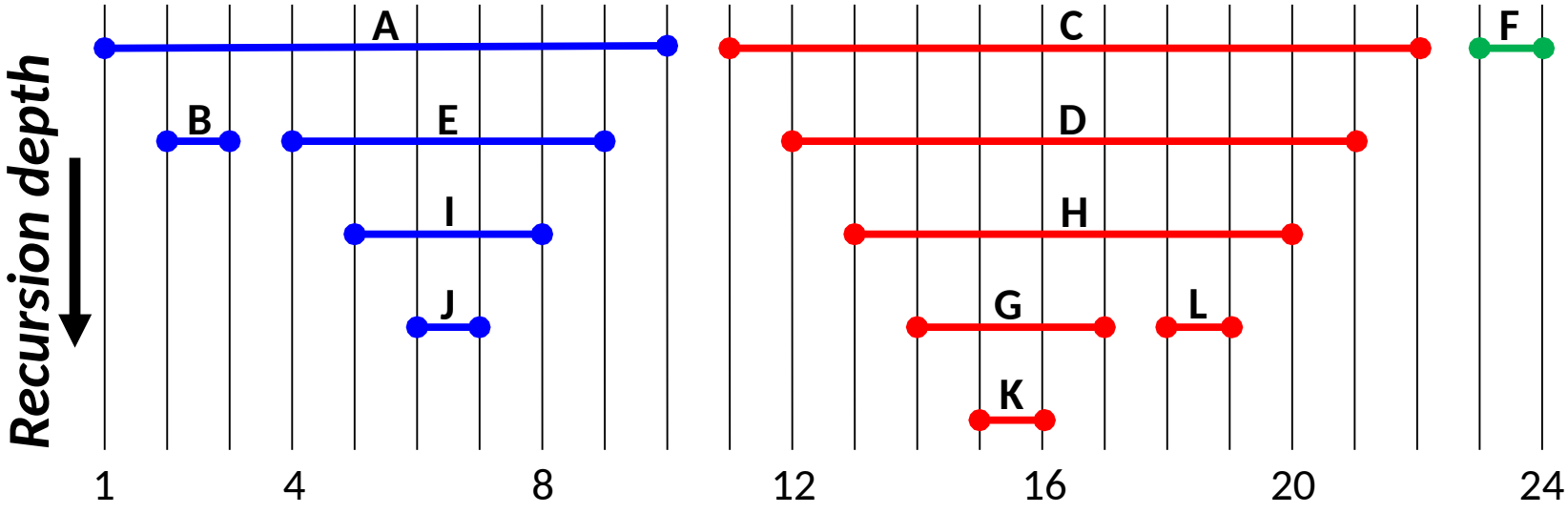
Property: Given two nodes u and v, the intervals (pre[u], post[u]) and (pre[v], post[v]) are either disjoint or one is contained within the other.

The pre/post interval of u is the lifetime of explore(u) in the stack (LIFO).

Example of pre/postvisit orderings

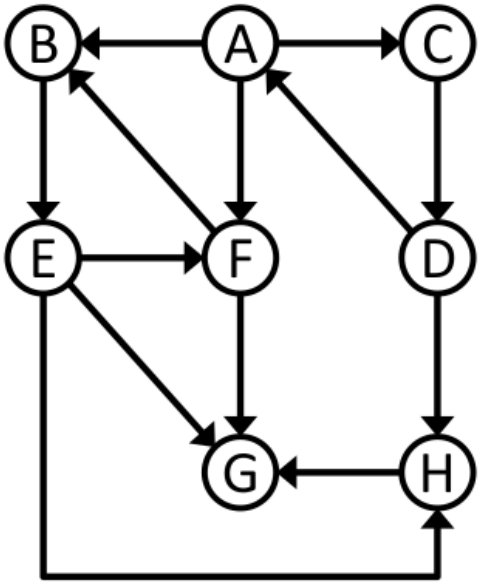


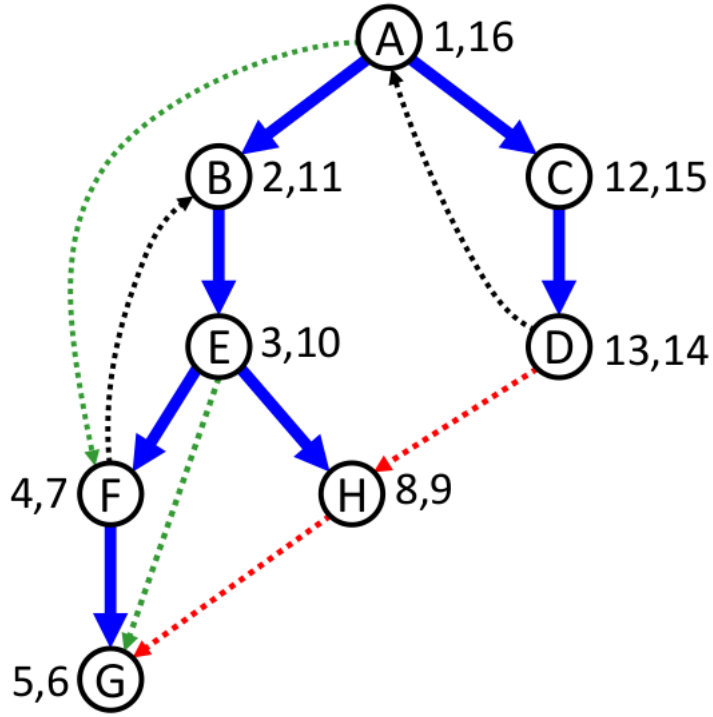


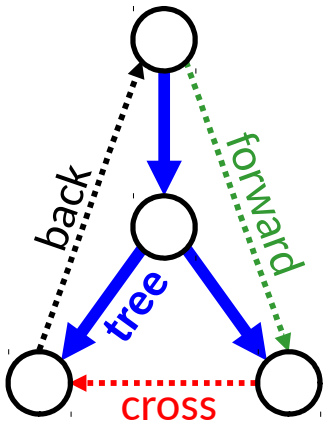


Source: UPC - J. Cortadella & J. Petit

DFS in directed graphs: types of edges





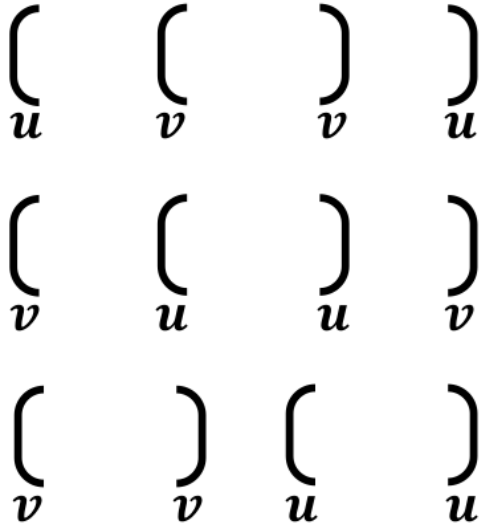


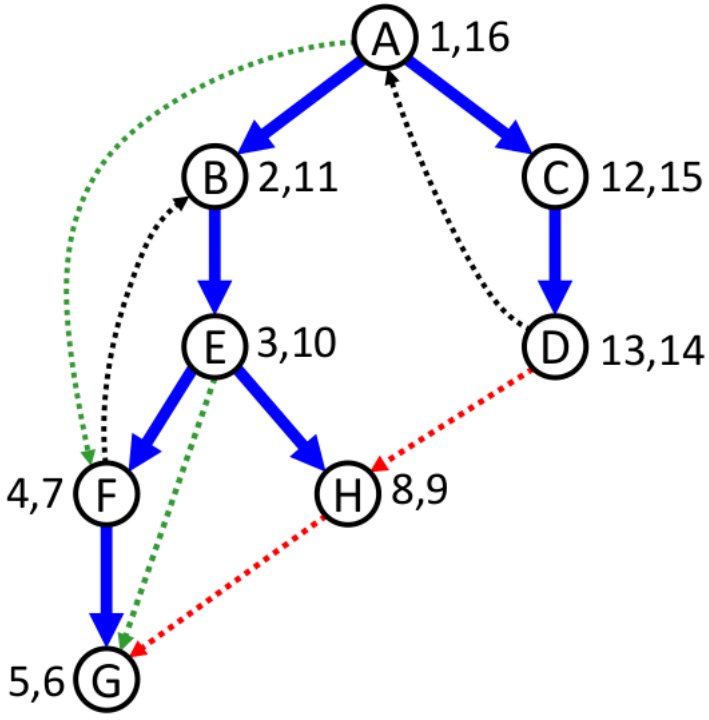
- **Tree edges:** those in the DFS forest.
- **Forward edges:** lead to a nonchild descendant in the DFS tree.
- **Back edges:** lead to an ancestor in the DFS tree.
- **Cross edges:** lead to neither descendant nor ancestor.

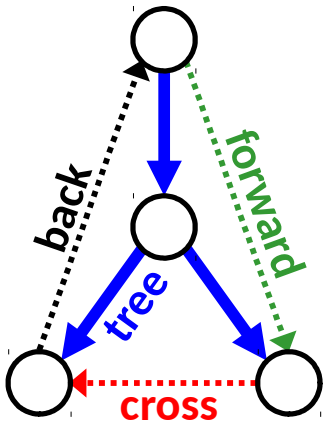
Source: UPC - J. Cortadella & J. Petit

DFS in directed graphs: types of edges

pre/post ordering for (u, v)



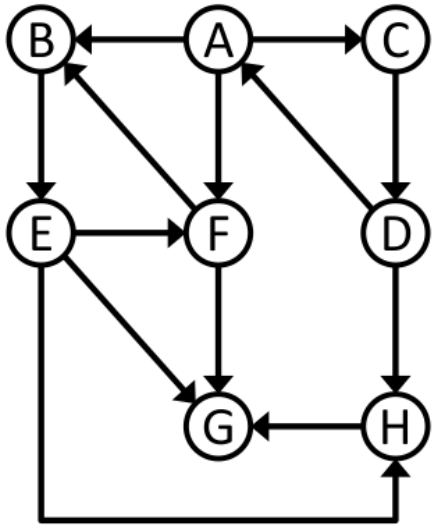




- **Tree edges:** those in the DFS forest.
- **Forward edges:** lead to a nonchild descendant in the DFS tree.
- **Back edges:** lead to an ancestor in the DFS tree.
- **Cross edges:** lead to neither descendant nor ancestor.

Source: UPC - J. Cortadella & J. Petit

Cycles in graphs



A ***cycle*** is a circular path:

$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$

Example:

$C \rightarrow D \rightarrow A \rightarrow C$

Property: A directed graph has a cycle iff its DFS reveals a back edge.

Proof:

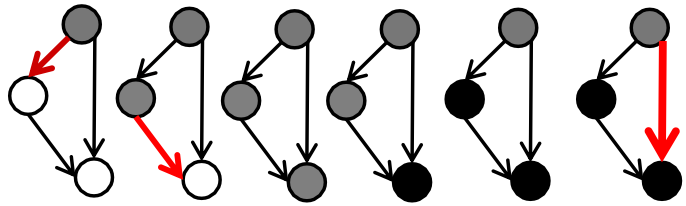
$\Leftarrow$ If (u,v) is a back edge, there is a cycle with (u,v) and the path from v to u in the search tree.

$\Rightarrow$ Let us consider a cycle $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$. Let us assume that v_i is the first discovered vertex (lowest pre number). All the other v_j on the cycle are reachable from v_i and will be its descendants in the DFS tree. The edge $v_{i-1} \rightarrow v_i$ leads from a vertex to its ancestor and is thus a back edge.

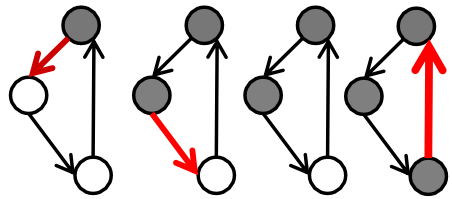
DFS and cyclic graphs

There are two ways DFS can **revisit** a node:

1. DFS has already fully explored the node. **What color does it have then? Is there a cycle then?**
No, the node is revisited from outside.



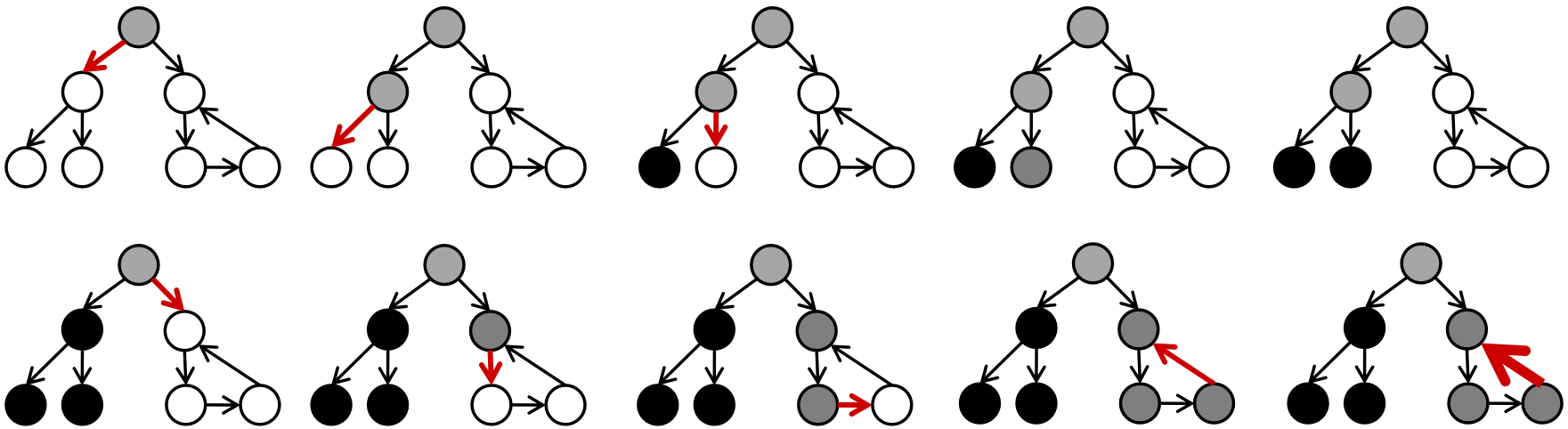
2. DFS is still exploring this node. **What color does it have in this case? Is there a cycle then?**
Yes, the node is revisited on a path containing the node itself.



So DFS with the white, grey, black coloring scheme detects a cycle when a GREY node is visited

Source: ColoState - Wim Bohm

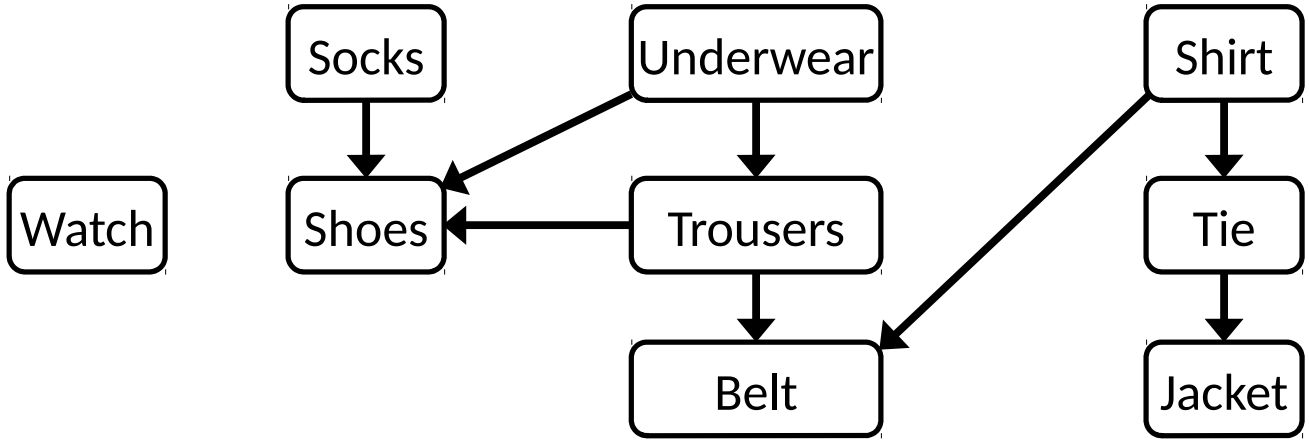
Cycle detection: DFS + coloring



When a grey (frontier) node is visited, a cycle is detected.

Source: ColoState - Wim Bohm

Getting dressed: DAG representation



A list of tasks that must be executed in a certain order (cannot be executed if it has cycles).

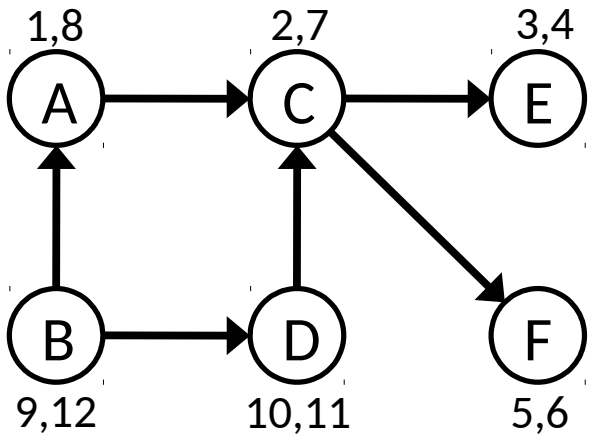
Legal task *linearizations* (or *topological sorts*):

Underwear	Socks	Trousers	Shoes	Watch	Shirt	Belt	Tie	Jacket

Watch	Socks	Shirt	Tie	Jacket	Underwear	Trousers	Belt	Shoes

Source: UPC - J. Cortadella & J. Petit

Directed Acyclic Graphs (DAGs)



A **DAG** is a directed graph without cycles.

DAGs are often used to represent causalities or temporal dependencies, e.g., task A must be completed before task C.

- Cyclic graphs cannot be linearized.

- All DAGs can be linearized. How?
  - Decreasing order of the post numbers.
  - The only edges (u,v) with post[u] < post[v] are back edges (do not exist in DAGs).

- **Property:** In a DAG, every edge leads to a vertex with a lower post number.

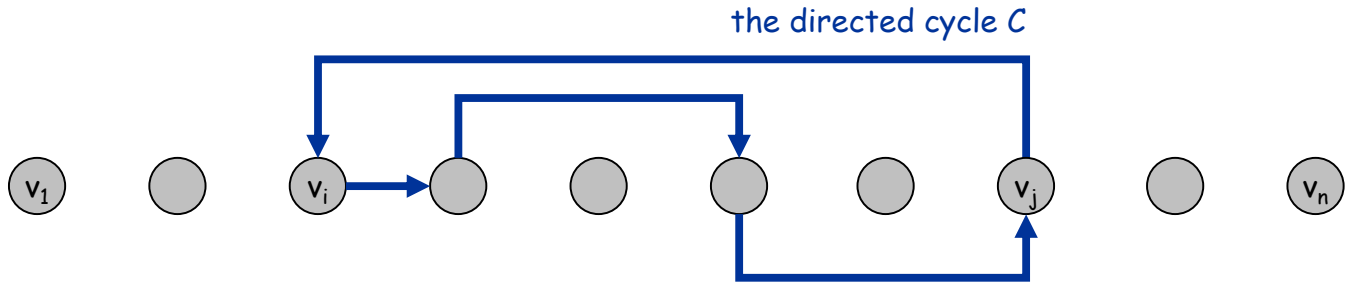
- **Property:** Every DAG has at least one source and at least one sink.
  (source: highest post number, sink: lowest post number).

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Proof. (by contradiction)

- Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C.
- Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
- By our choice of i, we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$
- **contradiction.** ▪



the supposed topological order: $v_1, \ldots, v_n$

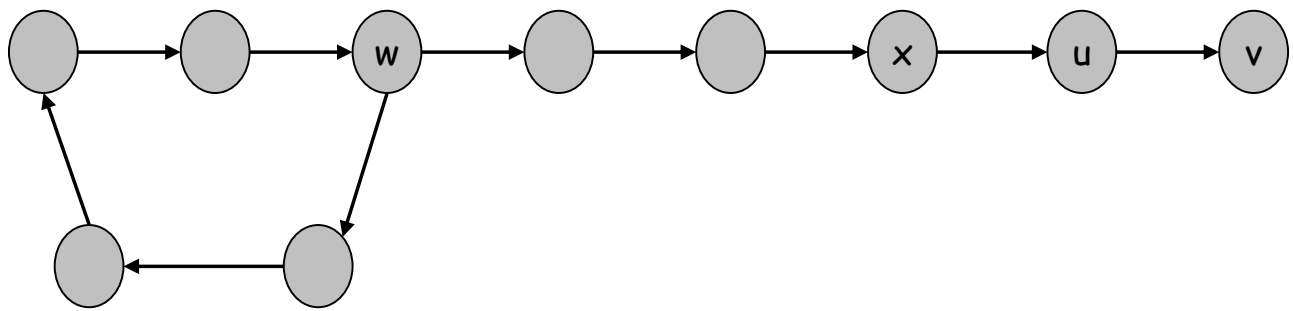
Source: ColoState - Wim Bohm

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Proof. (by contradiction)

- Suppose that G is a DAG and every node has at least one incoming edge. Then it has a cycle and thus is not a DAG:
  - Pick any node v, and begin following edges backward from v.
  - Repeat. After n + 1 steps we have visited a node, say w, twice. The sequence of nodes encountered between successive visits is a cycle.
- **Contradiction**.



Source: ColoState - Wim Bohm

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.
Proof. (by induction on n)

- Base: true if n = 1.
- Step: Given a DAG with n > 1 nodes, find a node v with no incoming edges. G - { v } is a DAG, since deleting v cannot create cycles. By induction hypothesis, G - { v } has a topological ordering. ▪

To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of $G-\{v\}$
  and append this order after v

Source: ColoState - Wim Bohm

Topological Sort: Algorithm Running Time

Theorem. Algorithm can run in $O(m + n)$ time.

Proof.

- Maintain the following information:
  - `count[w]` = remaining number of incoming edges
  - S = set of nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: pick a node v in S
  - remove v from S
  - for each edge (v, w) : decrement `count[w]` and add w to S if `count[w]` hits 0
  - this is $O(1)$ per edge ▪

Source: ColoState - Wim Bohm

Topological sort

```
def explore(G, v):
    visited(v) = true
    previsit(v)
    for each edge (v,u) ∈ E:
        if not visited(u):
            explore(G, u)
    postvisit(v)
```

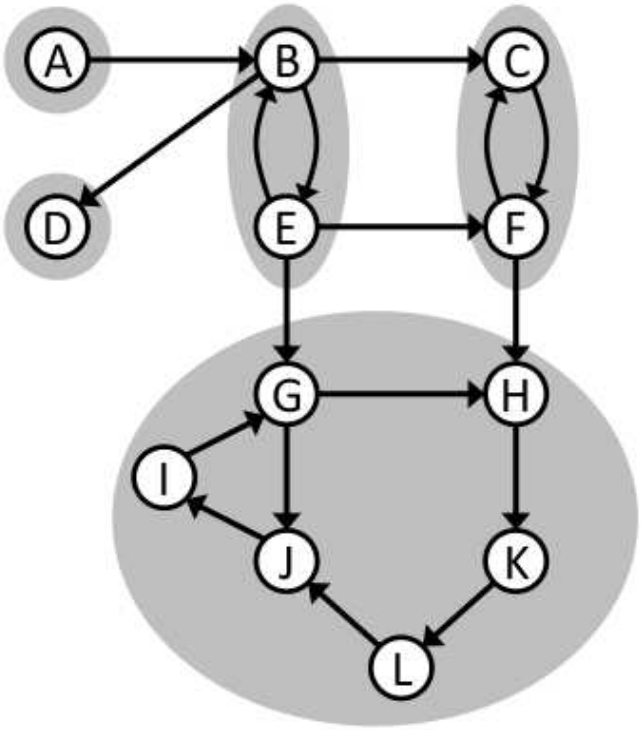
```
Initially: TSort = []

def postvisit(v):
    TSort.insert(0,v)

# After DFS, TSort
# contains a topological
# sort
```

Another algorithm:

- Find a source vertex, write it, and delete it (mark) from the graph.
- Repeat until the graph is empty.

It can be executed in linear time. How?

Strongly Connected Components



The graph is not *strongly connected*.

Two nodes u and v of a directed graph are connected if there is a path from u to v and a path from v to u.

The *connected* relation is an equivalence relation and partitions V into disjoint sets of *strongly connected components*.

This graph is connected (undirected view), but there is no path between any pair of nodes.

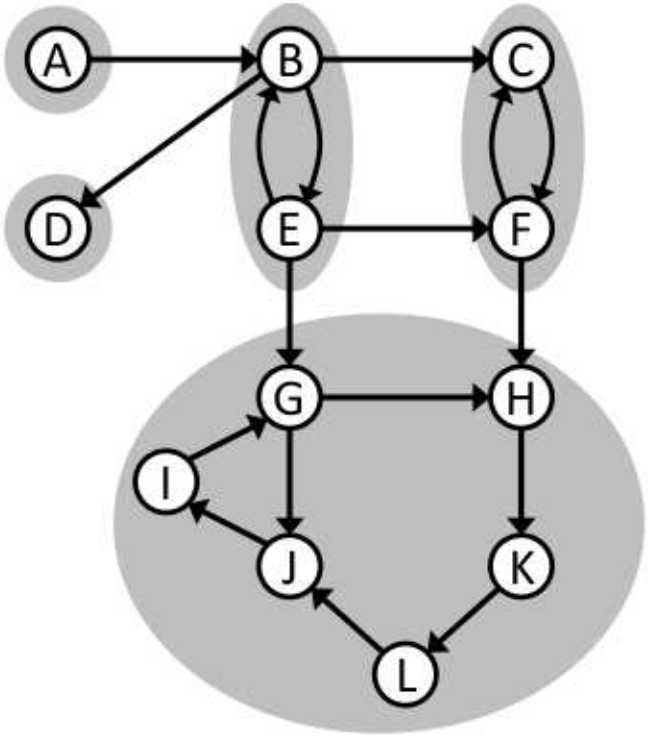
For example, there is no path $K \rightarrow \cdots \rightarrow C$ or $E \rightarrow \cdots \rightarrow A$.

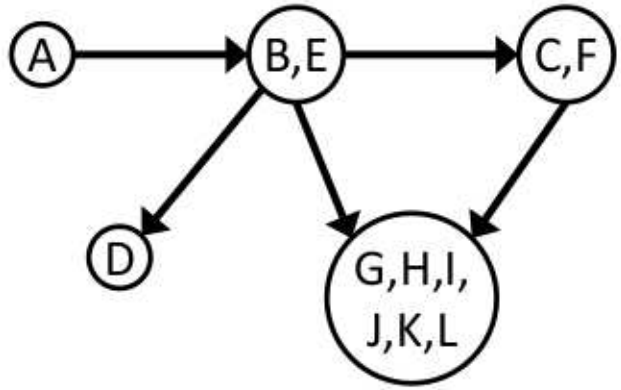
Strongly Connected Components

$\{A\}$
$\{B, E\}$
$\{C, F\}$
$\{D\}$
$\{G, H, I, J, K, L\}$

Source: UPC - J. Cortadella & J. Petit

Strongly Connected Components





Property: every directed graph is a DAG of its strongly connected components.

A directed graph can be seen as a 2-level structure. At the top we have a DAG of SCCs. At the bottom we have the details of the SCCs.

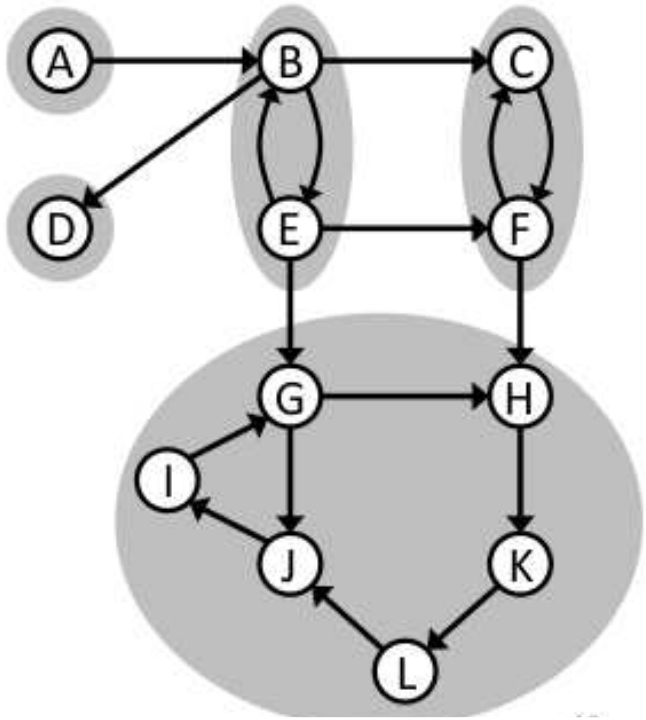
Every directed graph can be represented by a ***meta-graph***, where each meta-node represents a strongly connected component.

Source: UPC - J. Cortadella & J. Petit

Properties of DFS and SCCs

- **Property:** If the *explore* function starts at u, it will terminate when all vertices reachable from u have been visited.
  - If we start from a vertex in a sink SCC, it will retrieve exactly that component.
  - If we start from a non-sink SCC, it will retrieve the vertices of several components.

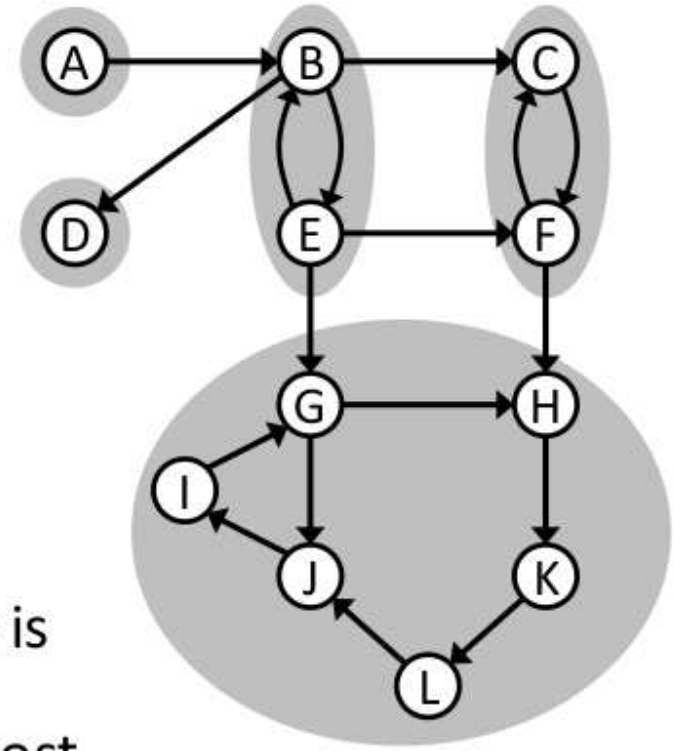
- **Examples:**
  - If we start at K it will retrieve the component $\{G, H, I, J, K, L\}$.
  - If we start at B it will retrieve all vertices except A.



Source: UPC - J. Cortadella & J. Petit

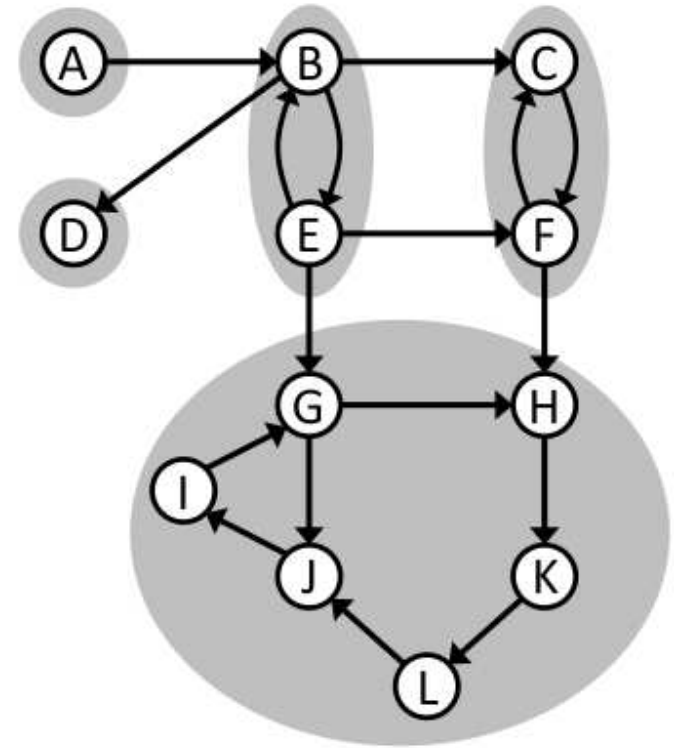
Properties of DFS and SCCs

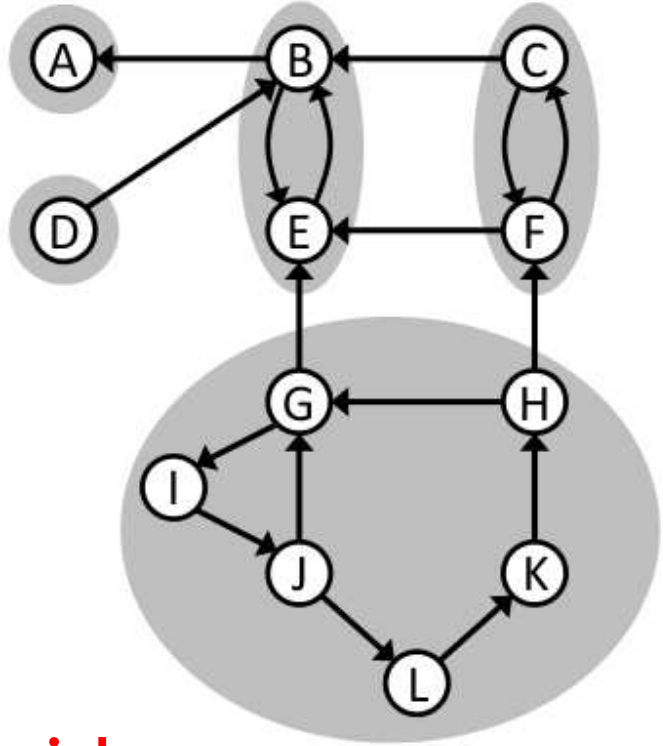
- **Intuition for the algorithm:**
  - Find a vertex located in a sink SCC
  - Extract the SCC
- **To be solved:**
  - How to find a vertex in a sink SCC?
  - What to do after extracting the SCC?
- **Property:** If C and C' are SCCs and there is an edge $C \rightarrow C'$, then the highest post number in C is bigger than the highest post number in C'.
- **Property:** The vertex with the highest DFS post number lies in a source SCC.

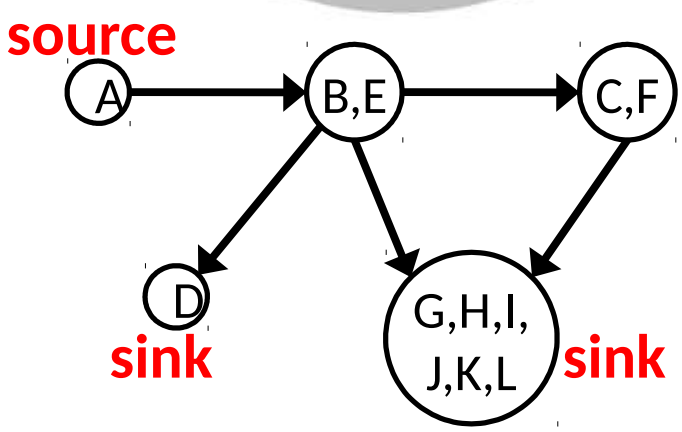


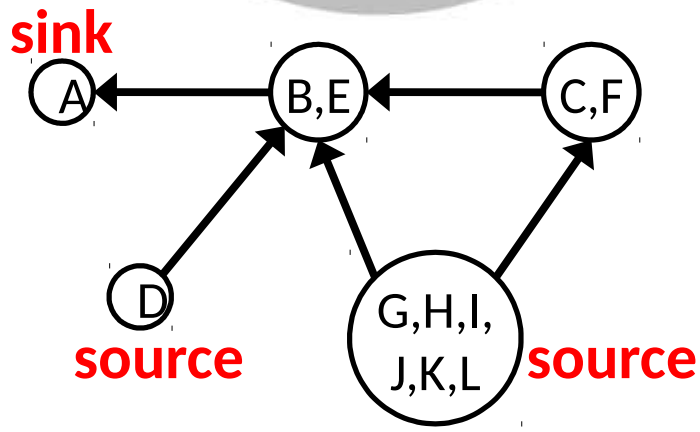
Source: UPC - J. Cortadella & J. Petit

Reverse graph (G^R)









Source: UPC - J. Cortadella & J. Petit

SCC algorithm

```
def SCC(G):
    # Input: G=(V,E) a directed graph
    # Output: each vertex v has an SCC number in ccnum[v]
    G^R = reverse(G)
    DFS(G^R)  # calculates post numbers
    sort V    # decreasing order of post number
    ConnComp(G)
```

Runtime complexity:

- DFS and ConnComp run in linear time $O(|V|+|E|)$.
- Can we reverse G in linear time?
- Can we sort V by post number in linear time?

Reversing G in linear time

```
def SCC(G):
    # Input: G=(V,E) a directed graph
    # Output: each vertex v has an SCC number in ccnum[v]
    G^R = reverse(G)
    DFS(G^R)  # calculates post numbers
    sort V    # decreasing order of post number
    ConnComp(G)
```

```
def reverse(G):
    # Input: G(V,E) graph represented by an adjacency list
    #        edges[v] for each vertex v.
    # Output: G(V,E^R) the reversed graph of G, with the
    #         adjacency list edgesR[v].
    for each u ∈ V:
        for each v ∈ edges[u]
            edgesR[v].insert(u)
    return (V, edgesR)
```

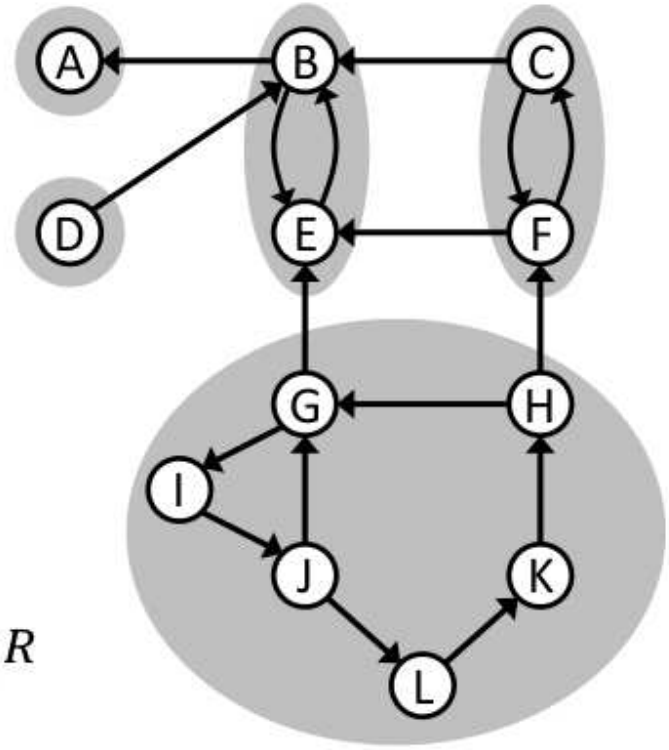
Sorting V in linear time

```
def SCC(G):
    # Input: G=(V,E) a directed graph
    # Output: each vertex v has an SCC number in ccnum[v]
    G^R = reverse(G)
    DFS(G^R)  # calculates post numbers
    sort V    # decreasing order of post number
    ConnComp(G)
```

Use the explore function for topological sort:

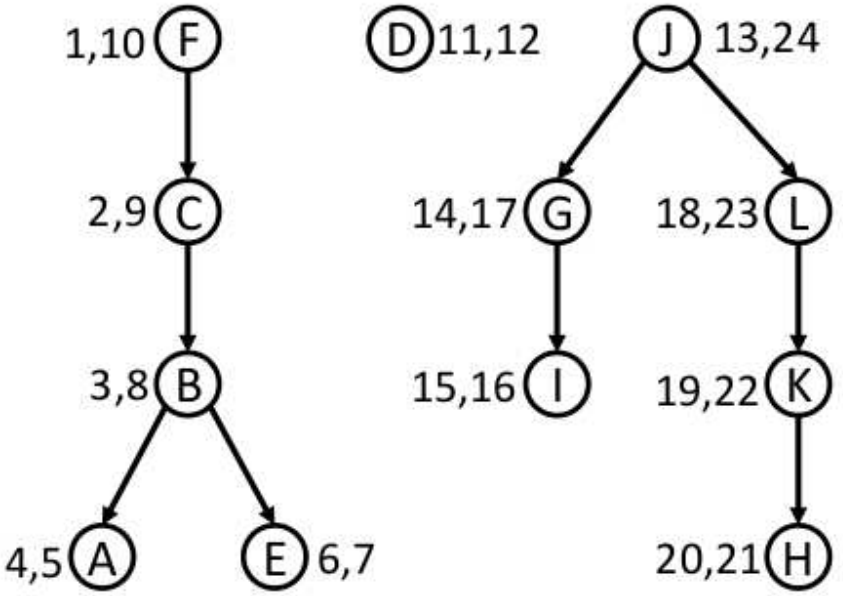
- Each time a vertex is post-visited, it is inserted at the top of the list.
- The list is ordered by decreasing order of post number.
- It is executed in linear time

Sorting V in linear time



G^R

DFS tree



Assume the initial order:

$F, A, B, C, D, E, J, G, H, I, K, L$

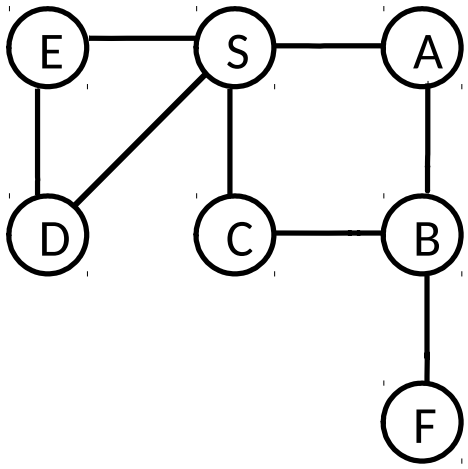
Vertex:	J	L	K	H	G	I	D	F	C	B	E	A
Post:	24	23	22	21	17	16	12	10	9	8	7	5

Source: UPC - J. Cortadella & J. Petit

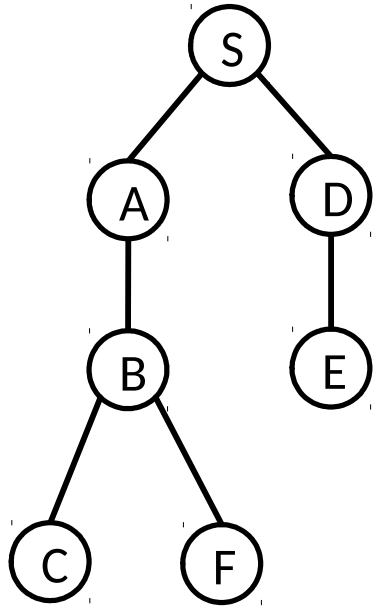
Distance in a graph

Depth-first search finds vertices reachable from another given vertex. The paths are not the shortest ones.

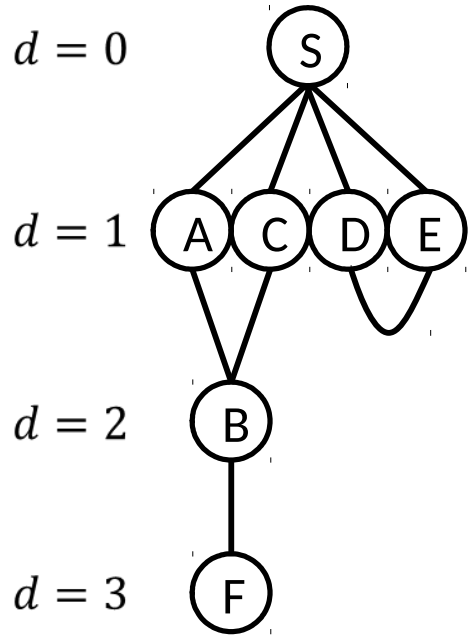
Graph



DFS tree

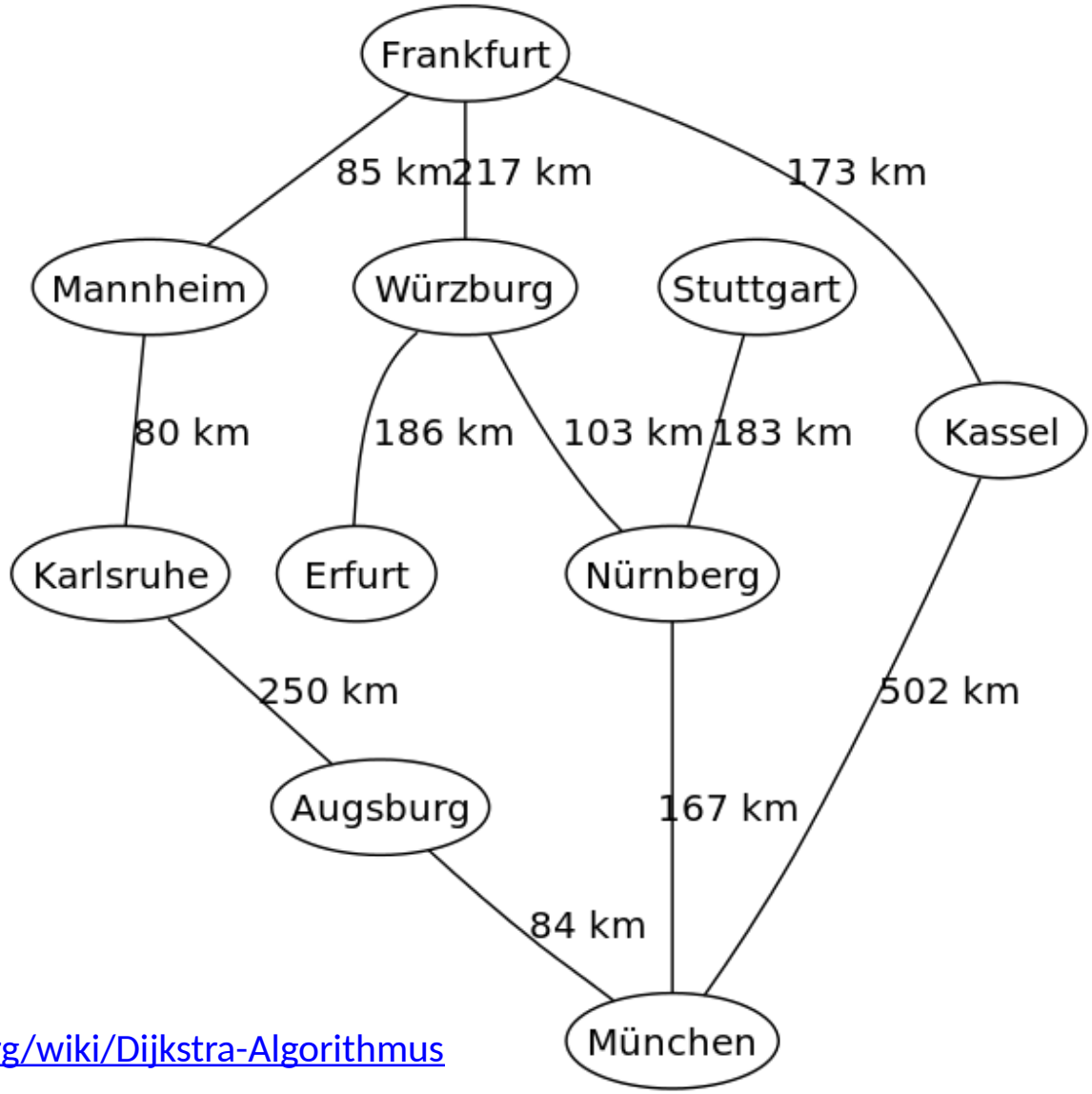


Shortest distance



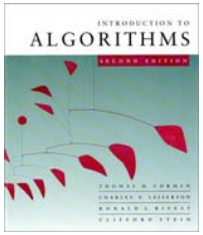
Distance between two nodes: length of the shortest path between them

Distances on edges



https://de.wikipedia.org/wiki/Dijkstra-Algorithmus

Source: UPC - J. Cortadella & J. Petit

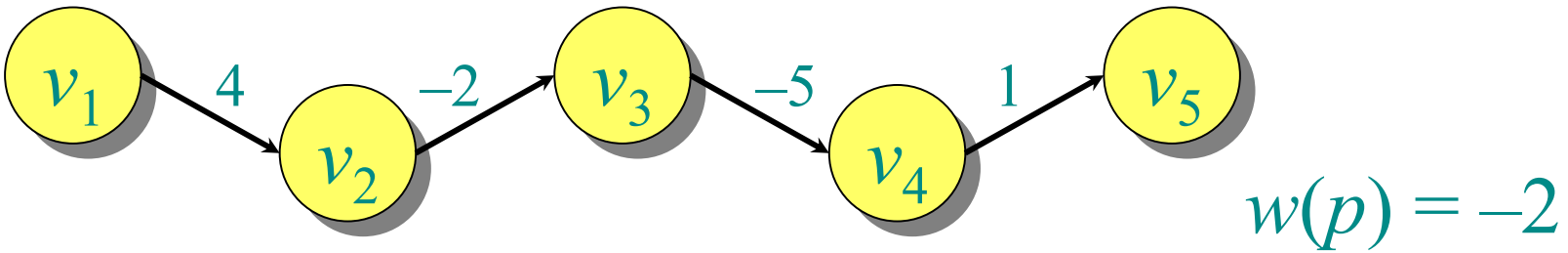


Paths in graphs

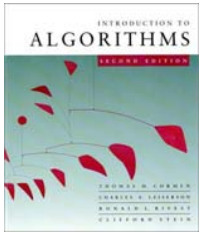
Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to \mathbb{R}$. The ***weight*** of path $p = v_1 \to v_2 \to \cdots \to v_k$ is defined to be

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}) .$$

Example:



$w(p) = -2$

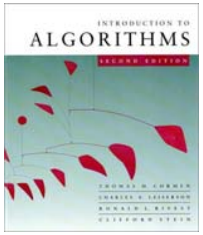


Shortest paths

A ***shortest path*** from u to v is a path of minimum weight from u to v. The ***shortest-path weight*** from u to v is defined as

$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.$$

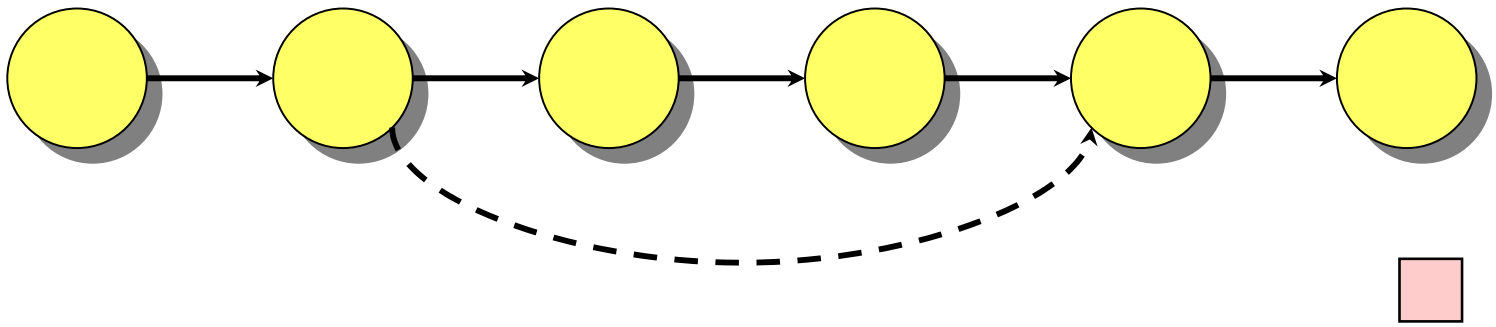
Note: $\delta(u, v) = \infty$ if no path from u to v exists.

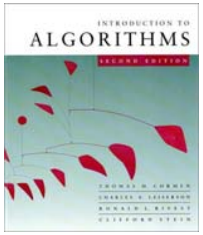


Optimal substructure

Theorem. A subpath of a shortest path is a shortest path.

Proof. Cut and paste:



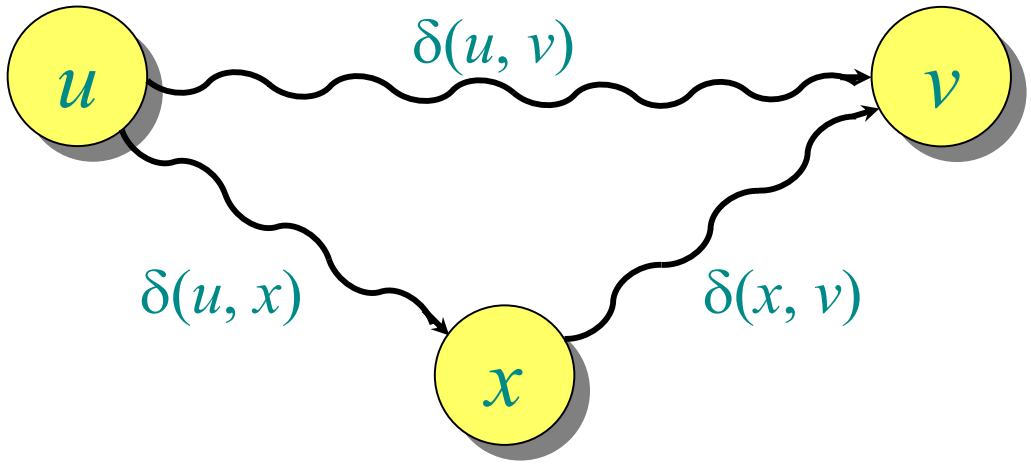


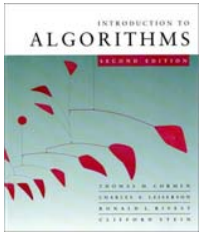
Triangle inequality

Theorem. For all $u, v, x \in V$, we have

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v).$$

Proof.

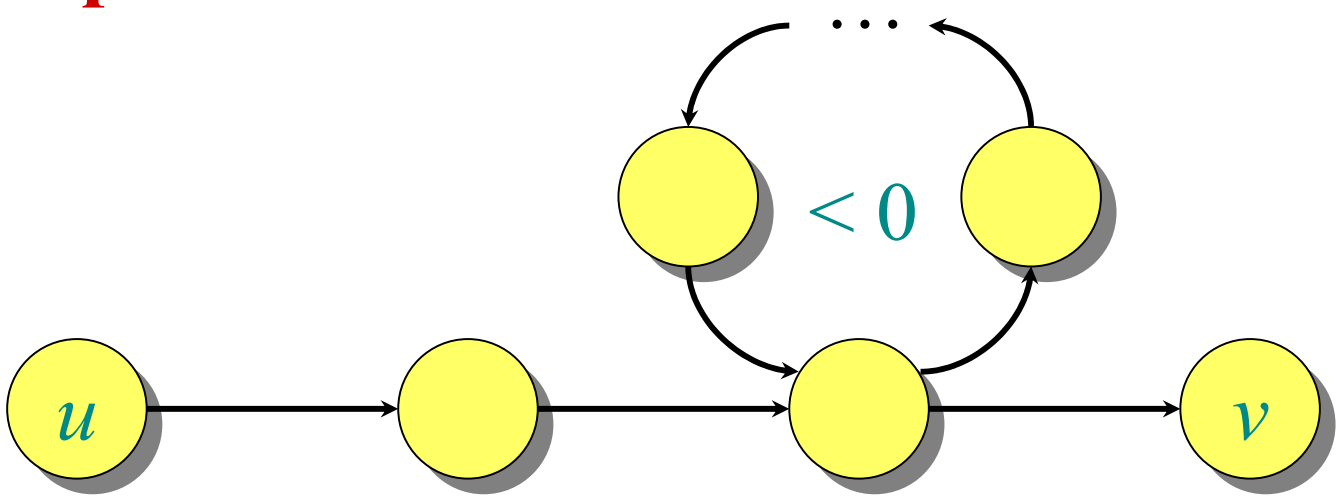


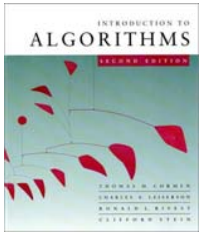


Well-definedness of shortest paths

If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Example:

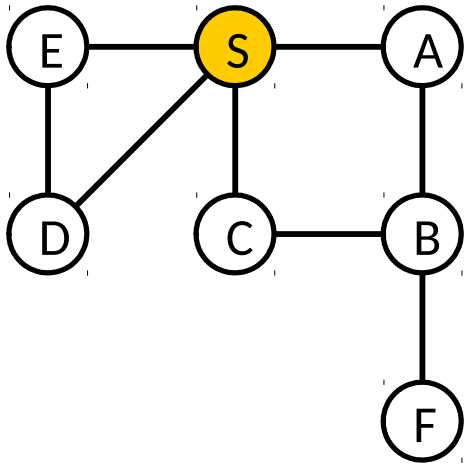




Hallmark for "greedy" algorithms

Greedy-choice property
A locally optimal choice is globally optimal.

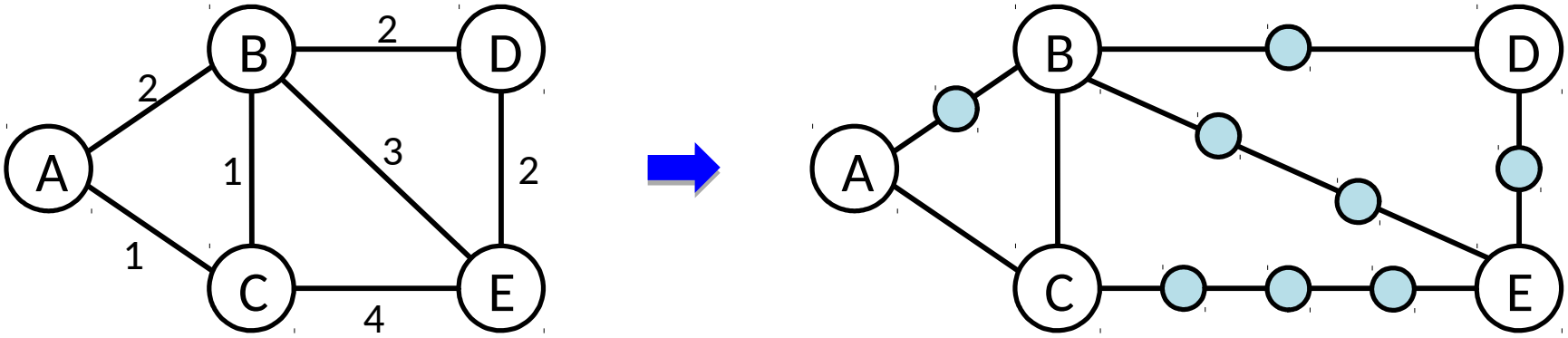
BFS algorithm

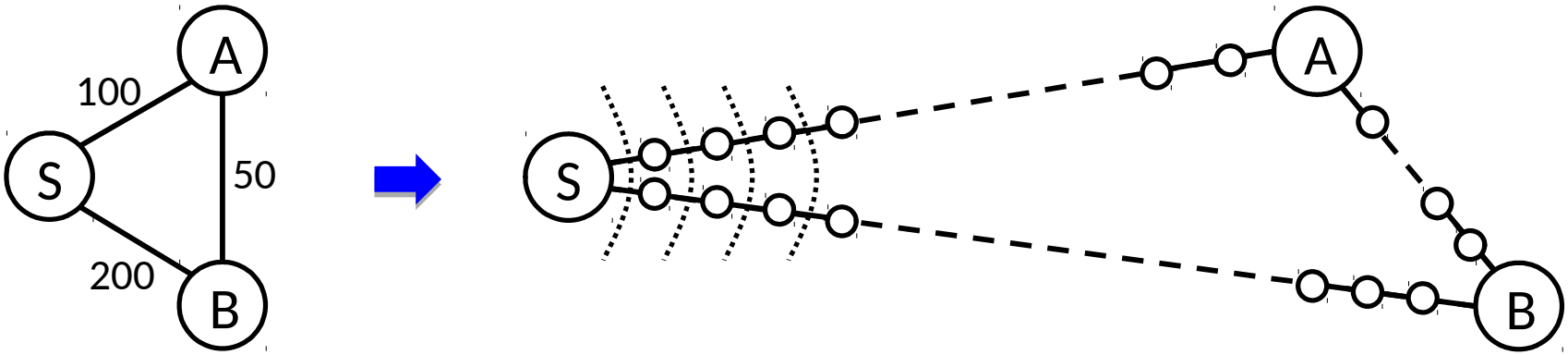


		S	A	B	C	D	E	F
	S_0	0	∞	∞	∞	∞	∞	∞
S_0	$A_1\ C_1\ D_1\ E_1$	0	1	∞	1	1	1	∞
A_1	$C_1\ D_1\ E_1\ B_2$	0	1	2	1	1	1	∞
C_1	$D_1\ E_1\ B_2$	0	1	2	1	1	1	∞
D_1	$E_1\ B_2$	0	1	2	1	1	1	∞
E_1	B_2	0	1	2	1	1	1	∞
B_2	F_3	0	1	2	1	1	1	3
F_3		0	1	2	1	1	1	3

Source: UPC - J. Cortadella & J. Petit

Reusing BFS

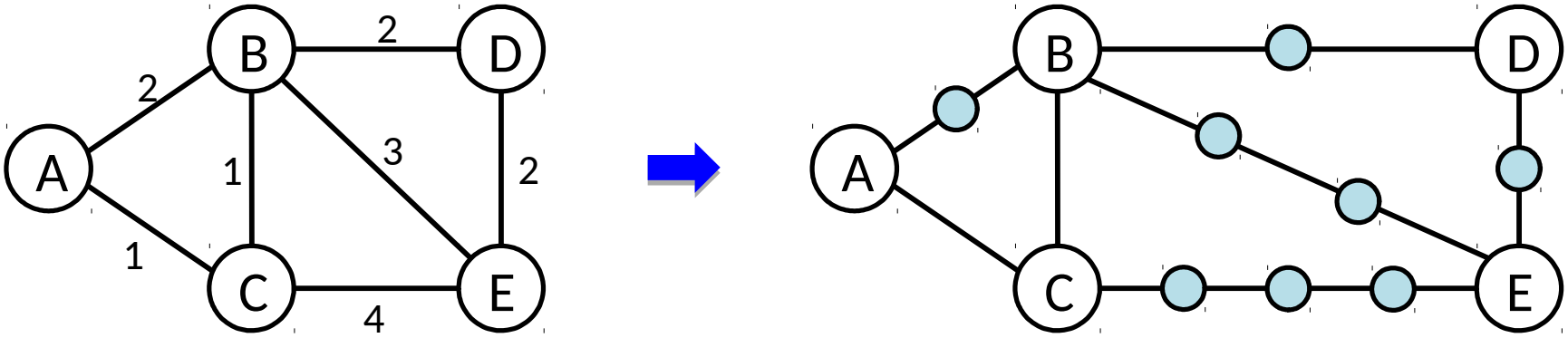


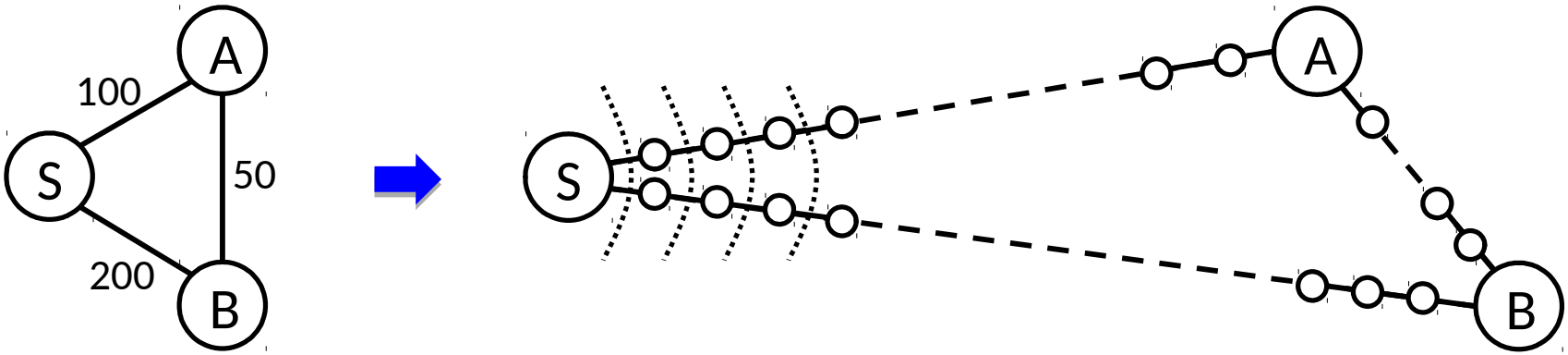


Inefficient: many cycles without any interesting progress.

Source: UPC - J. Cortadella & J. Petit

Reusing BFS



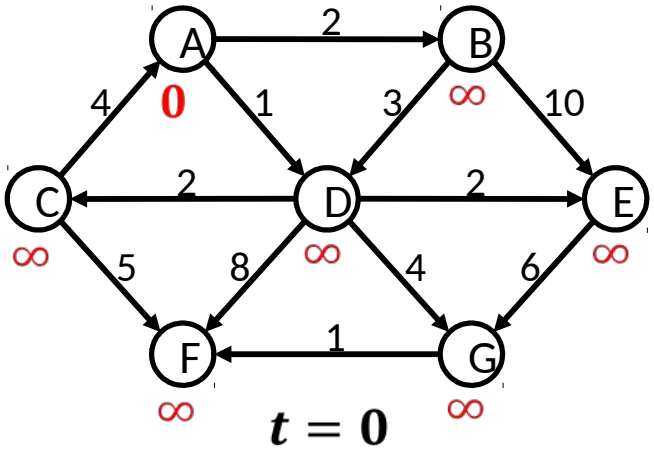


Inefficient: many cycles without any interesting progress.

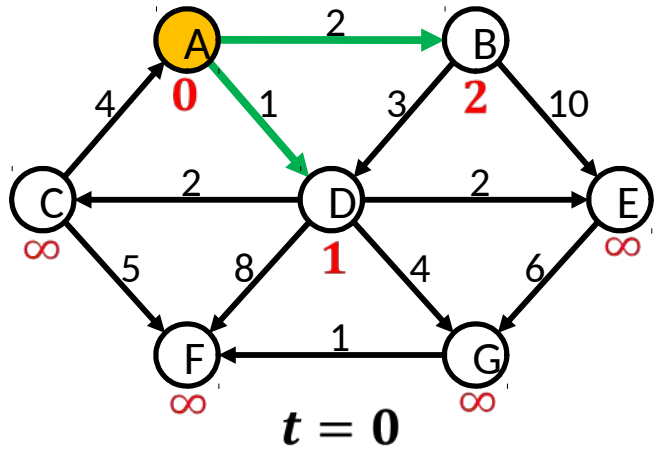
"Use visit times"

Source: UPC - J. Cortadella & J. Petit

Example



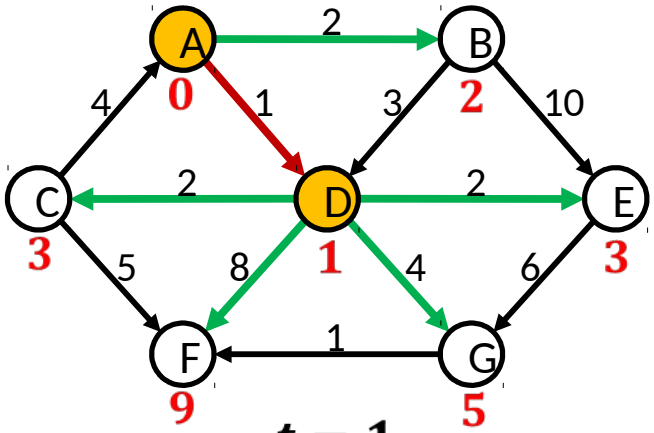
Done	Queue
	A:0
	B:∞
	E:∞
	D:∞
	C:∞
	F:∞
	G:∞



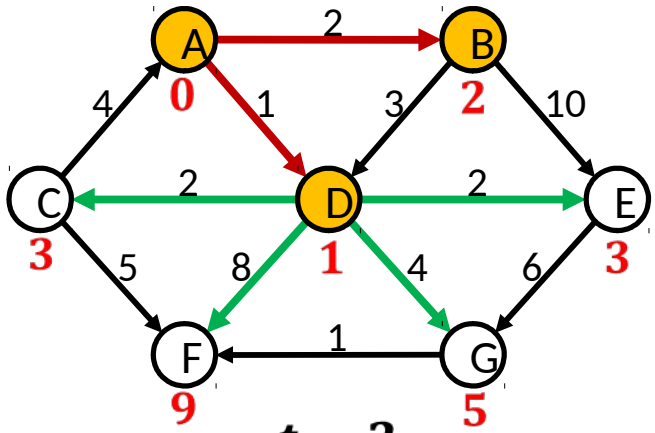
Done	Queue
A:0	D:1
	B:2
	E:∞
	C:∞
	F:∞
	G:∞

Source: UPC - J. Cortadella & J. Petit

Example



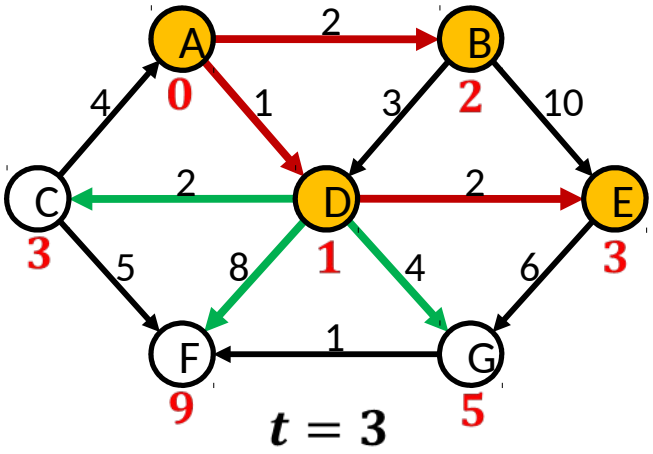
Done	Queue
A:0	B:2
D:1	E:3
	C:3
	G:5
	F:9



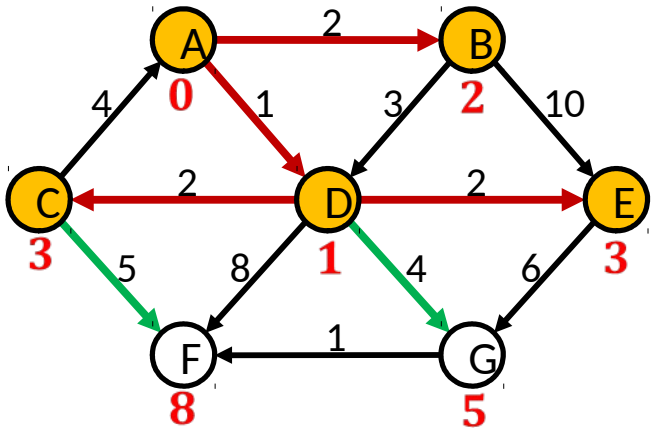
Done	Queue
A:0	E:3
D:1	C:3
B:2	G:5
	F:9

Source: UPC - J. Cortadella & J. Petit

Example



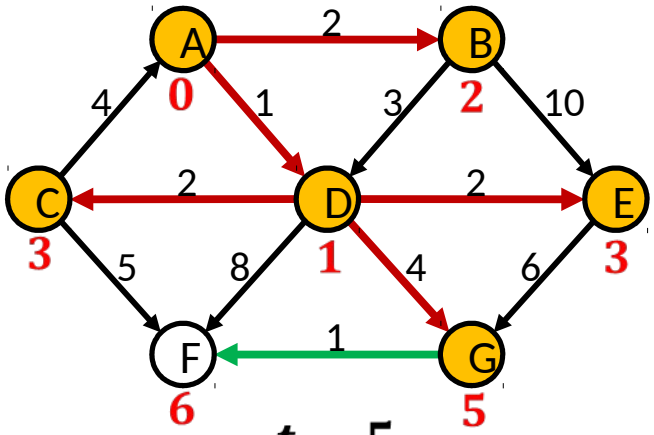
Done	Queue
A:0	C:3
D:1	G:5
B:2	F:9
E:3	



Done	Queue
A:0	G:5
D:1	F:8
B:2	
E:3	
C:3	

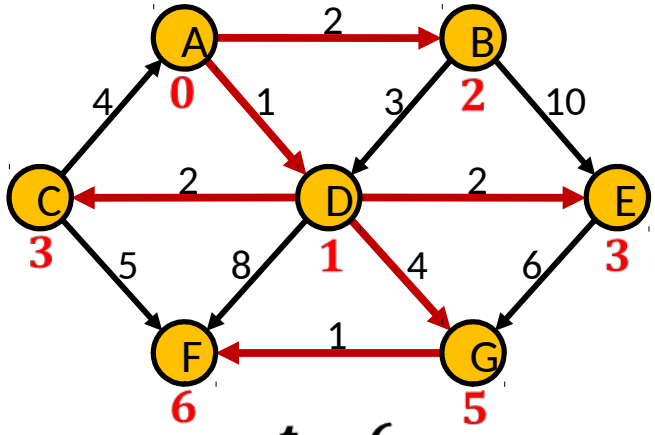
Source: UPC - J. Cortadella & J. Petit

Example



$t = 5$

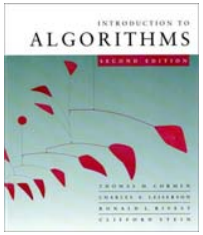
Done	Queue
A:0	F:6
D:1	
B:2	
E:3	
C:3	
G:5	



$t = 6$

Done	Queue
A:0	
D:1	
B:2	
E:3	
C:3	
G:5	
F:6	

Source: UPC - J. Cortadella & J. Petit



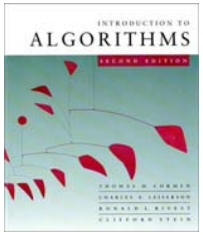
Single-source shortest paths

Problem. From a given source vertex $s \in V$, find the shortest-path weights $\delta(s, v)$ for all $v \in V$.

If all edge weights $w(u, v)$ are *nonnegative*, all shortest-path weights must exist.

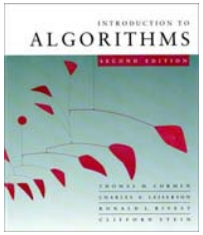
IDEA: Greedy.

1. Maintain a set S of vertices whose shortest-path distances from s are known.
2. At each step add to S the vertex $v \in V - S$ whose distance estimate from s is minimal.
3. Update the distance estimates of vertices adjacent to v.



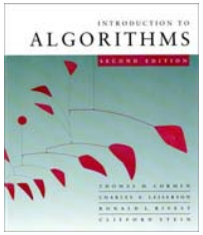
Dijkstra's algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$ ▷ Q is a priority queue maintaining $V - S$



Dijkstra's algorithm

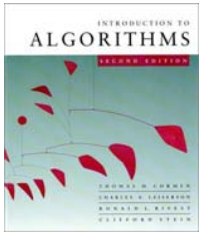
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
  do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$ ▷ Q is a priority queue maintaining $V - S$
while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$



Dynamic-programming hallmark #1

Optimal substructure

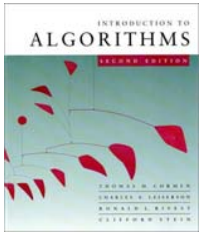
An optimal solution to a problem (instance) contains optimal solutions to subproblems.



Dynamic-programming hallmark #2

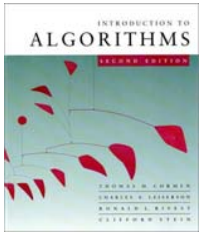
Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.



Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.



Dijkstra's algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
  do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$ ▷ Q is a priority queue maintaining $V - S$
while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$

relaxation step

Implicit DECREASE-KEY

Dijkstra's algorithm for shortest paths

```
def ShortestPaths(G, l, s):          # Pseudo-Python!
    # Input: Graph G, source vertex s,
    #        positive edge lengths l for all e in E
    # Output: dist[u] has the distance from s,
    #         prev[u] has the predecessor in the tree

    for all u in V:
        dist[u] = ∞
        prev[u] = nil

    dist[s] = 0
    Q = makequeue(V)    # using dist as keys

    while not Q.empty():
        u = Q.deletemin()
        for all(u,v) in E:
            if dist[v] > dist[u] + l(u,v):
                dist[v] = dist[u] + l(u,v)
                prev[v] = u
                Q.decreasekey(v)
```

Dijkstra's algorithm: complexity

```
Q = makequeue(V)
while not Q.empty():
  u = Q.deletemin()
  for all (u,v) in E:
    if dist[v] > dist[u] + l(u,v):
      dist[v] = dist[u] + l(u,v)
      prev[v] = u
      Q.decreasekey(v)
```

$|V|$ times

$|E|$ times

- The skeleton of Dijkstra's algorithm is based on BFS, which is $O(|V| + |E|)$
- We need to account for the cost of:
  - **makequeue**: insert $|V|$ vertices to a list.
  - **deletemin**: find the vertex with min dist in the list ($|V|$ times)
  - **decreasekey**: update dist for a vertex ($|E|$ times)

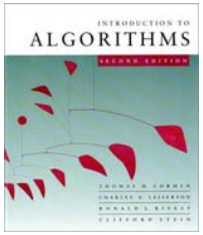
- Let us consider two implementations for the list: **vector** and **binary heap**

Dijkstra's algorithm: complexity

Implementation	deletemin	insert/ decreasekey	Dijkstra's complexity
Vector	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$

Binary heap:

- The elements are stored in a complete (balanced) binary tree.
- **Insertion:** place element at the bottom and let it *bubble up* swapping the location with the parent (at most $\log_2 |V|$ levels).
- **Deletemin:** Remove element from the root, take the last node in the tree, place it at the root and let it *sift down* (at most $\log_2 |V|$ levels).
- **Decreasekey:** decrease the key in the tree and let it bubble up (same as insertion). A data structure might be required to known the location of each vertex in the heap (table of pointers).



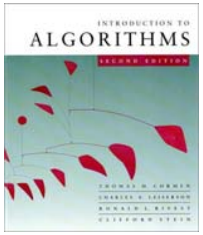
Correctness — Part I

Lemma. Initializing $d[s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all $v \in V - \{s\}$ establishes $d[v] \geq \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps.

Proof. Suppose not. Let v be the first vertex for which $d[v] < \delta(s, v)$, and let u be the vertex that caused $d[v]$ to change: $d[v] = d[u] + w(u, v)$. Then,

$$
\begin{aligned}
d[v] &< \delta(s, v) && \text{supposition} \\
&\leq \delta(s, u) + \delta(u, v) && \text{triangle inequality} \\
&\leq \delta(s,u) + w(u, v) && \text{sh. path} \leq \text{specific path} \\
&\leq d[u] + w(u, v) && v \text{ is first violation}
\end{aligned}
$$

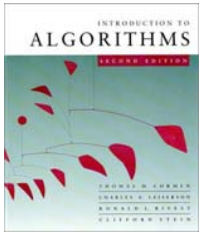
Contradiction. □



Correctness — Part II

Lemma. Let u be v's predecessor on a shortest path from s to v. Then, if $d[u] = \delta(s, u)$ and edge (u, v) is relaxed, we have $d[v] = \delta(s, v)$ after the relaxation.

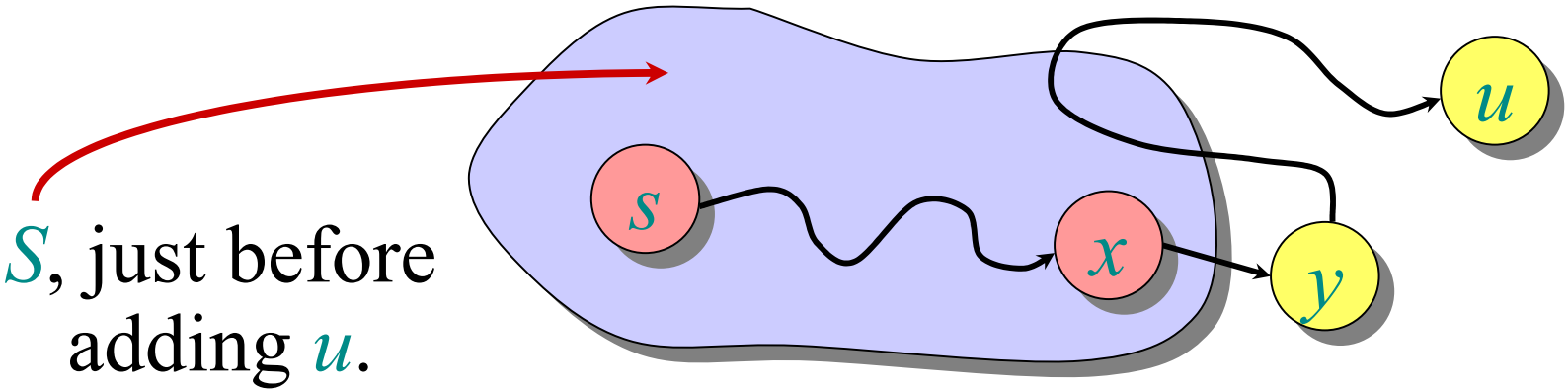
Proof. Observe that $\delta(s, v) = \delta(s, u) + w(u, v)$. Suppose that $d[v] > \delta(s, v)$ before the relaxation. (Otherwise, we're done.) Then, the test $d[v] > d[u] + w(u, v)$ succeeds, because $d[v] > \delta(s, v) = \delta(s, u) + w(u, v) = d[u] + w(u, v)$, and the algorithm sets $d[v] = d[u] + w(u, v) = \delta(s, v)$. □

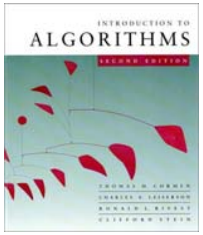


Correctness — Part III

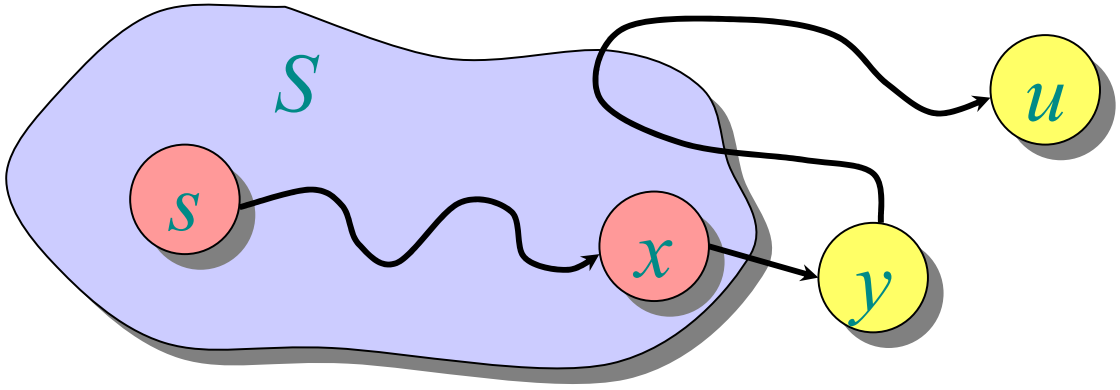
Theorem. Dijkstra's algorithm terminates with $d[v] = \delta(s, v)$ for all $v \in V$.

Proof. It suffices to show that $d[v] = \delta(s, v)$ for every $v \in V$ when v is added to S. Suppose u is the first vertex added to S for which $d[u] > \delta(s, u)$. Let y be the first vertex in $V - S$ along a shortest path from s to u, and let x be its predecessor:





Correctness — Part III (continued)



Since u is the first vertex violating the claimed invariant, we have $d[x] = \delta(s, x)$. When x was added to S, the edge (x, y) was relaxed, which implies that $d[y] = \delta(s, y) \leq \delta(s, u) < d[u]$. But, $d[u] \leq d[y]$ by our choice of u. Contradiction. □