Algorithms

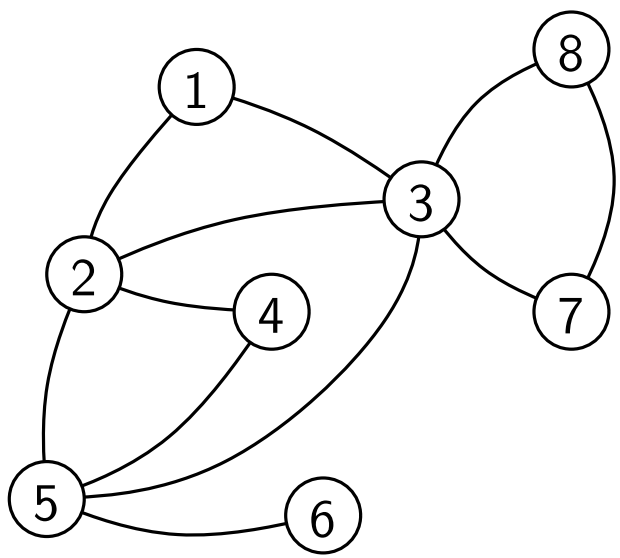
Graph Algorithms

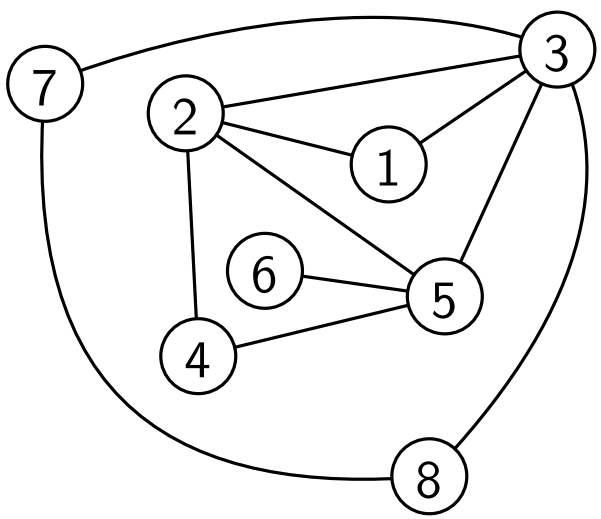
Hee-Kap Ahn
Graduate School of Artificial Intelligence
Dept. Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)

Textbooks: Algorithms by Jeff Erickson /Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.

Undirected Graphs

An undirected graph is denoted by $G = (V, E)$, where V denotes the set of **nodes** and E denotes the set of **edges** between pairs of nodes.
Undirected graphs capture pairwise relationship between objects.



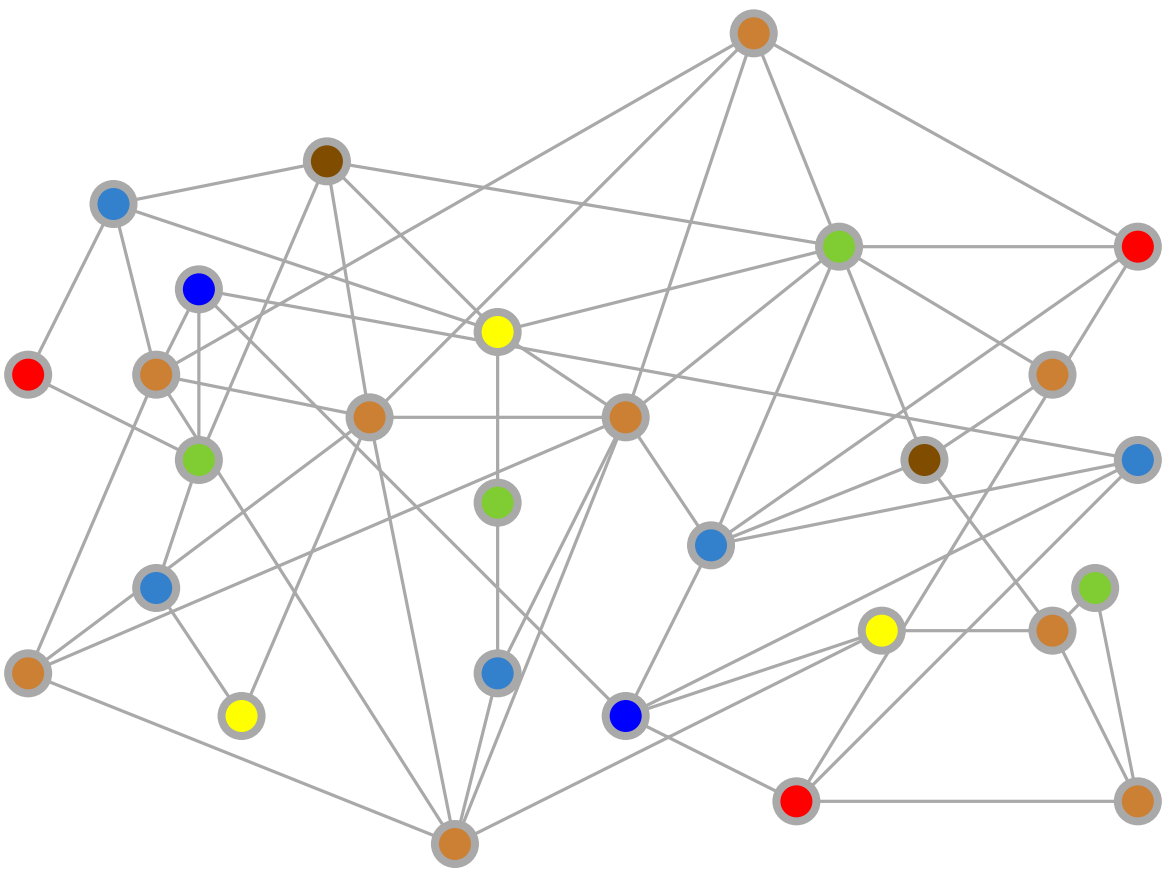


$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$E = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 7), (3, 8), (4, 5), (5, 6), (7, 8)\}$

Graph Applications

Graphs of

- World Wide Web: V - web pages, E - hyperlinks.
- Social networks (Instagram/Twitter/Facebook): V - people, E - relationship between two people.
- circuits: V - gates, E - wires.



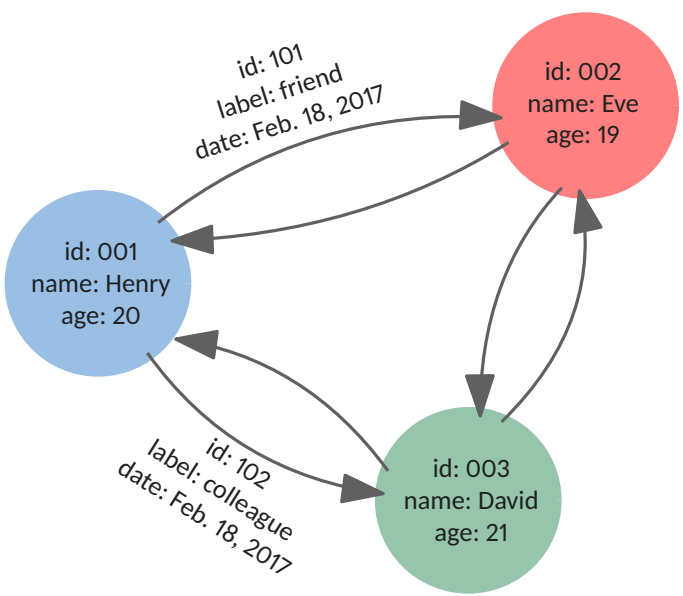
Graph Applications

Graph Database

- Uses graph structures for semantic queries with nodes, edges and properties to represent and store data.
- Represents/stores relationships of data items directly using links.
- Allows simple and fast retrieval of complex hierarchical structures.

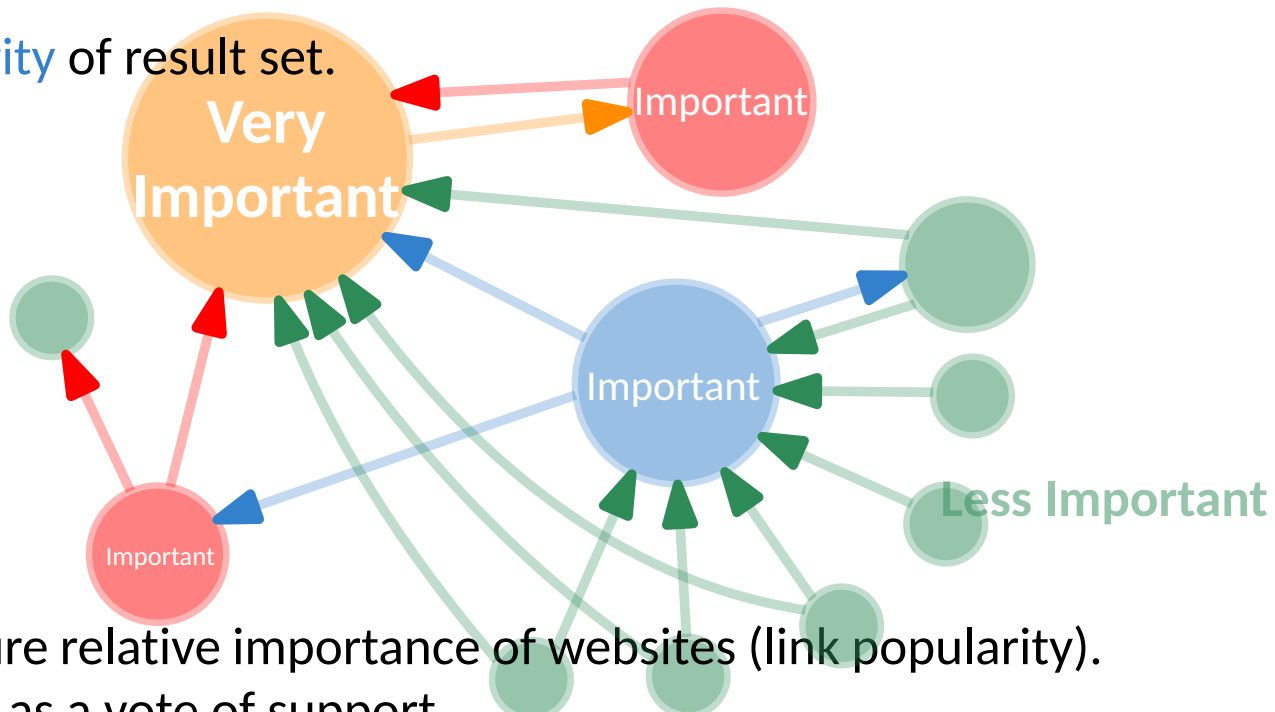
As the complexity of a query grows, a simple and fast graph retrieval is more demanding. For example,

> Play the Korean song on a sad love story sung by a female singer. Bo-gum Park is the main actor of the music video.



Graph Applications

- Web Search Engines on WWW, from 1993.
  Lycos (1994), AltaVista, Daum (1995), Google, MSN (1998), Naver (1999), Baidu (2000), Bing (2009).
- Web crawling, Indexing, Searching in near real time.
- Usefulness of search engine:
  relevance/popularity/authority of result set.

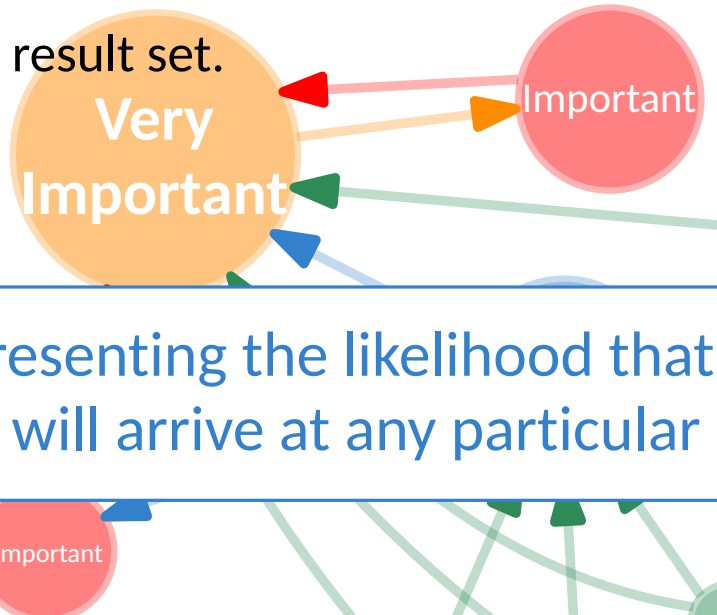


Google Search wanted to measure relative importance of websites (link popularity).

- A hyperlink to a page counts as a vote of support.
- PageRank of a page is defined recursively on the number and PageRank metric of all pages that link to it ("incoming links").
- A page linked to by many pages with high PageRank receives a high rank.

Graph Applications

- Web Search Engines on WWW, from 1993.
  Lycos (1994), AltaVista, Daum (1995), Google, MSN (1998), Naver (1999), Baidu (2000), Bing (2009).
- Web crawling, Indexing, Searching in near real time.
- Usefulness of search engine:
  relevance/popularity/authority of result set.



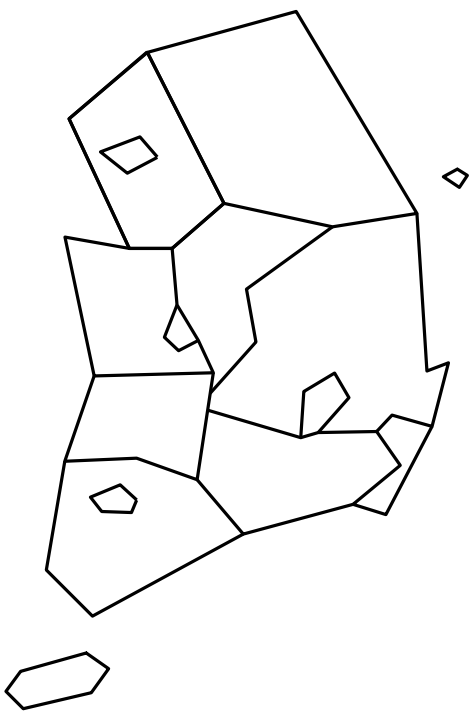
A probability distribution representing the likelihood that a person randomly clicking on links will arrive at any particular page.

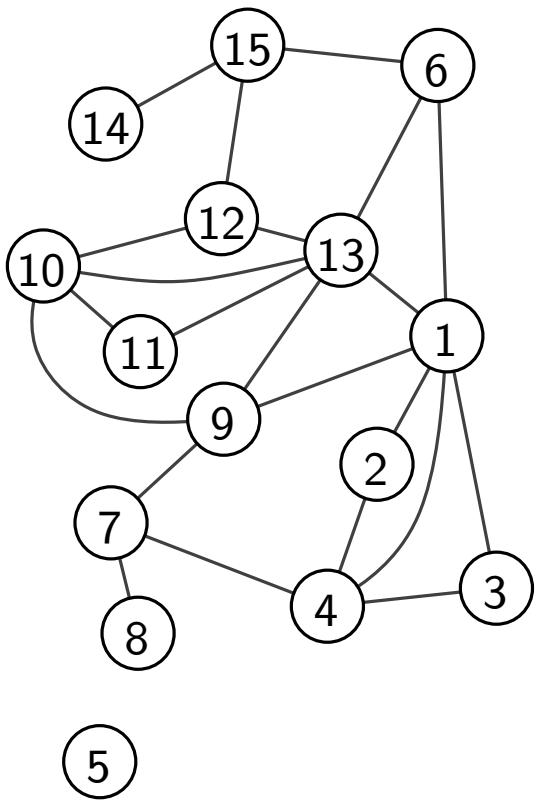
Google Search wanted to measure relative importance of websites (link popularity).

- A hyperlink to a page counts as a vote of support.
- PageRank of a page is defined recursively on the number and PageRank metric of all pages that link to it ("incoming links").
- A page linked to by many pages with high PageRank receives a high rank.

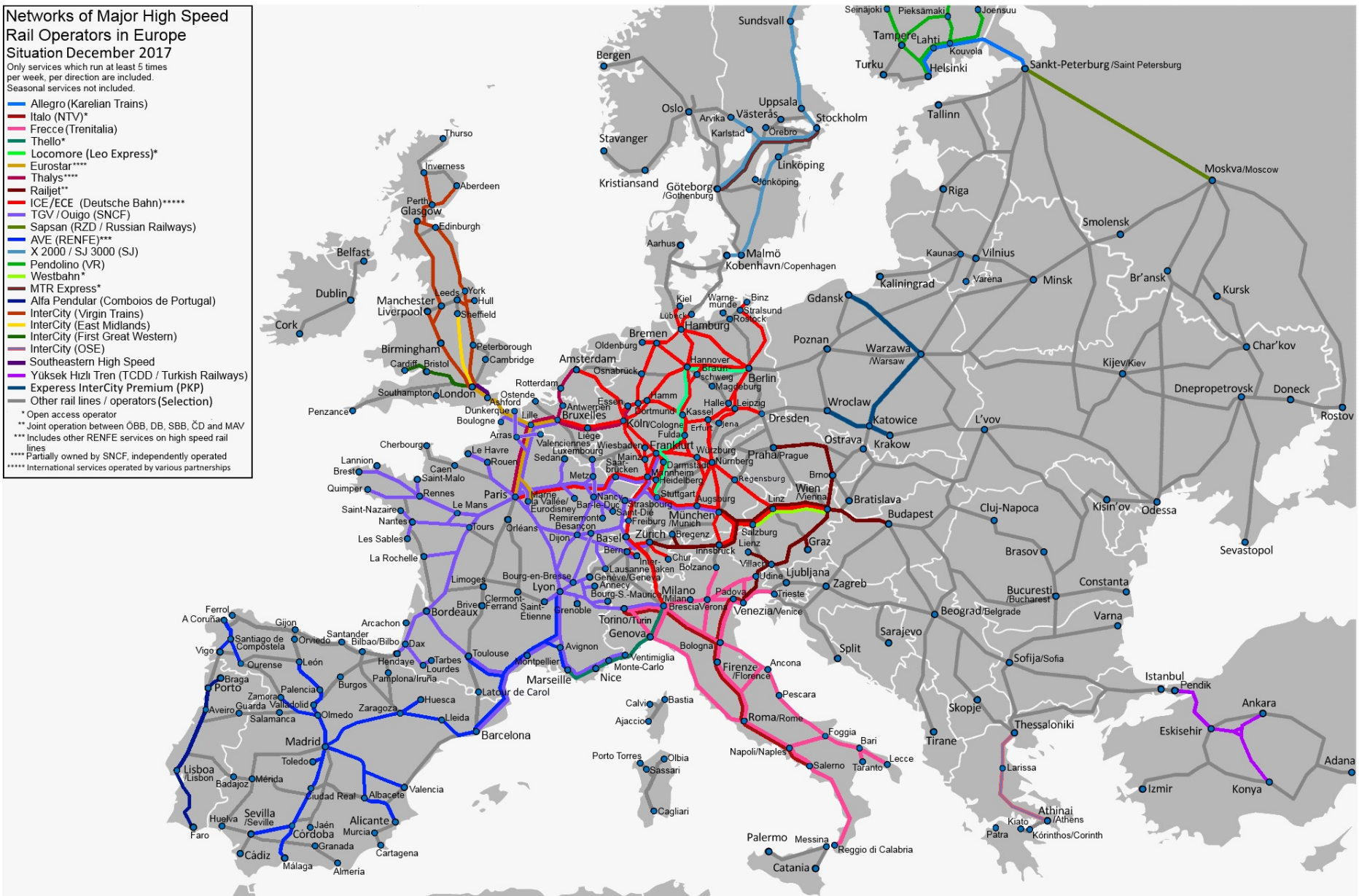
Graph Applications

Regions in maps



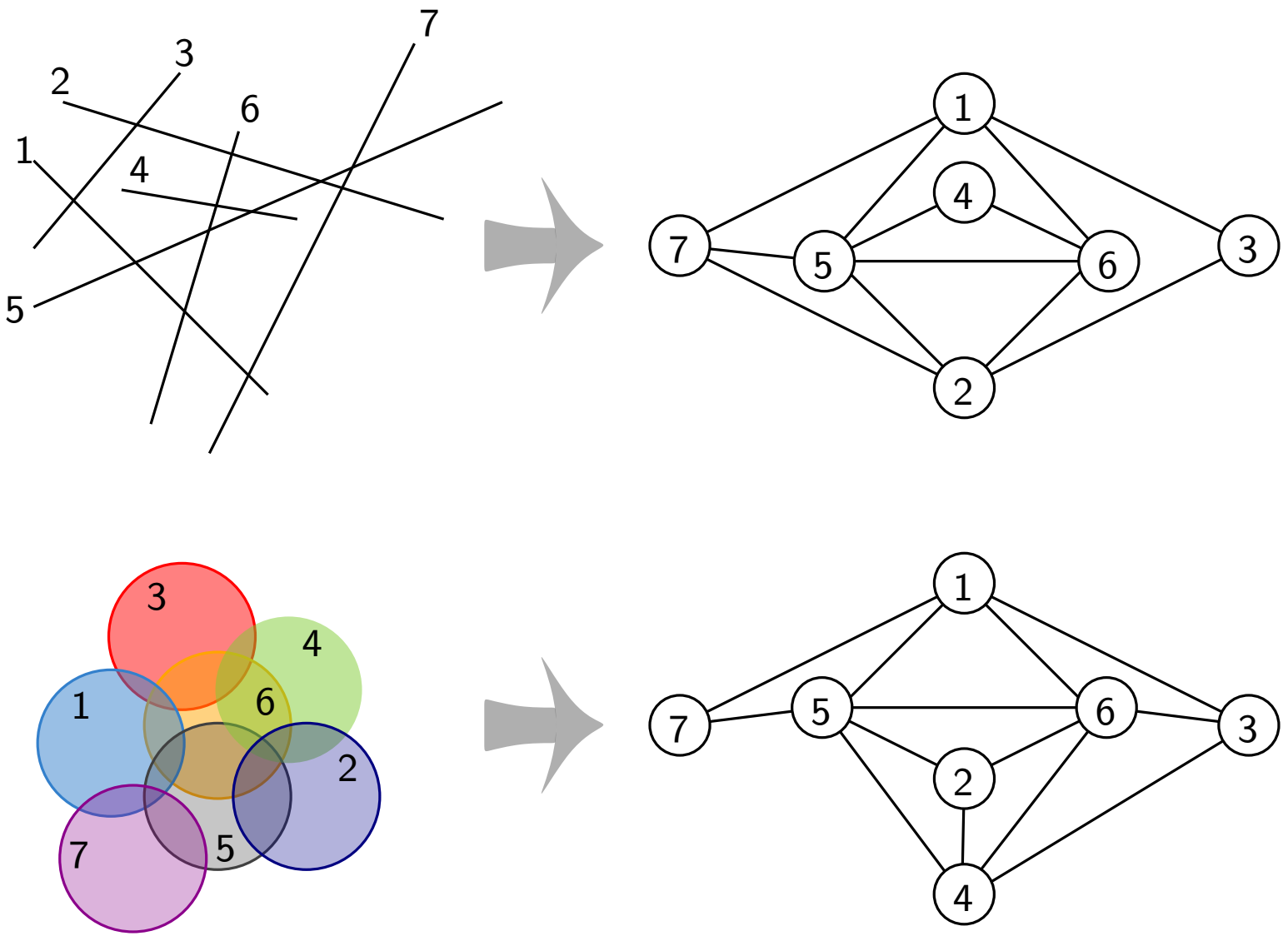


Graph Applications



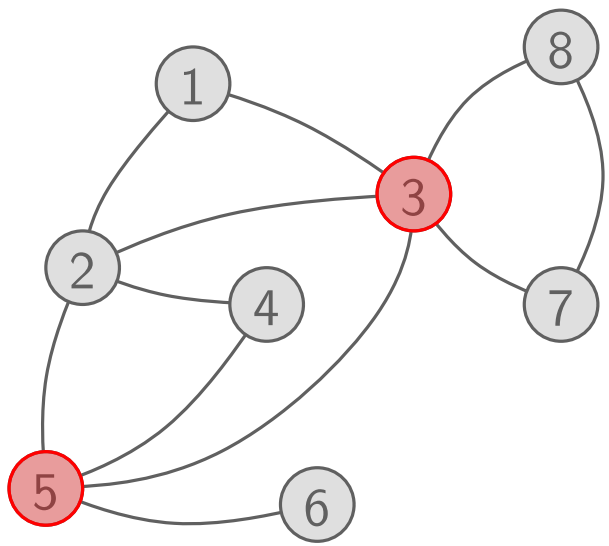
from Wikimedia Commons. (Bernese media)

Intersection Graphs



Graph Representation

Adjacency matrix. For a graph with $n = |V|$ nodes, this is an $n \times n$ array whose (i, j) entry $a_{i,j} = 1$ if (i, j) is an edge.

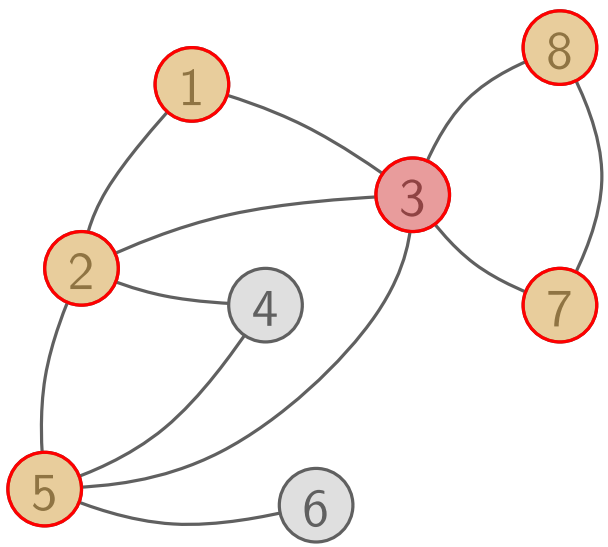


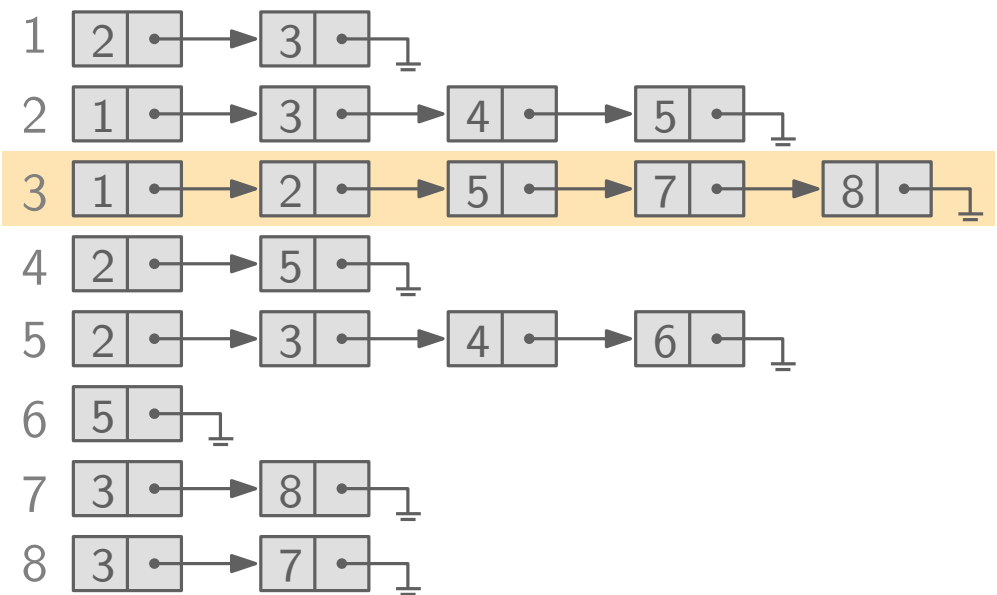
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

- contains two representations of each edge.
- requires $O(n^2)$ space.
- checks in $O(1)$ time whether $(u, v) \in E$ or not.
- identifies all edges in $\Theta(n^2)$ time.

Graph Representation

Adjacency list. It consists of $n = |V|$ linked lists, one per node. The linked list for node u holds the names of nodes to which u has an edge.





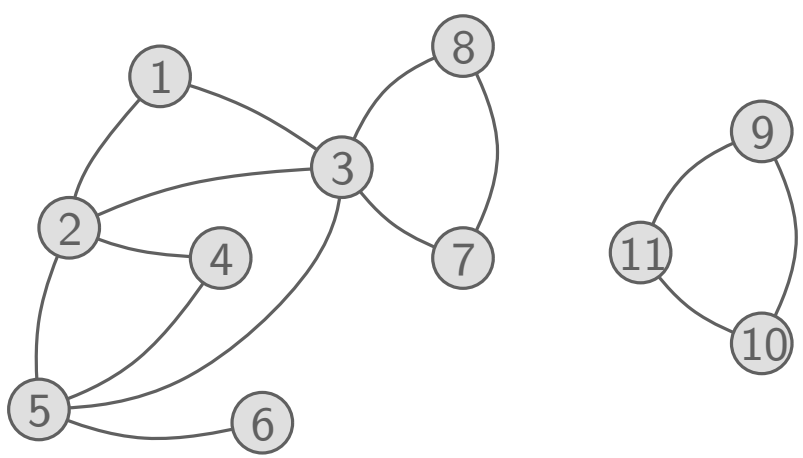
- contains two representations of each edge.
- requires $O(n + m)$ space. ($m = |E|$)
- checks in $O(\min\{\deg(u), \deg(v)\})$ time whether $(u, v) \in E$.
- identifies all edges in $\Theta(m + n)$ time.

Paths, Cycles, Connectivity

A **path** of a graph $G = (V, E)$ is a sequence of nodes, $\pi = v_1 v_2 \cdots v_{k-1} v_k$ with the property that each consecutive pair $v_i v_{i+1}$ is joined by an edge of G.
A path is **simple** if all nodes are distinct.

A **cycle** is a path $\pi = v_1 v_2 \cdots v_{k-1} v_k$ of length at least three in which $v_1 = v_k$ and the first $k - 1$ nodes are all distinct.

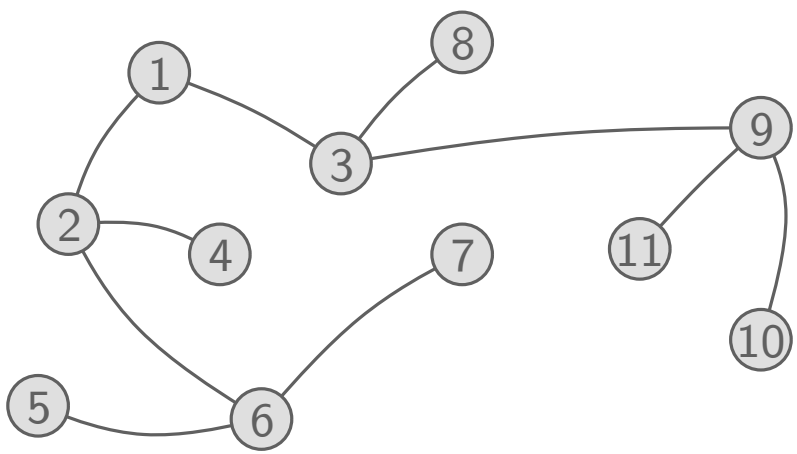
An undirected graph is **connected** if for every pair of nodes u and v, there is a path between u and v.



Trees

A **tree** is an undirected graph that is connected and acyclic.

Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.
(a) G is connected.
(b) G does not contain a cycle.
(c) G has $n - 1$ edges.



Property. An undirected graph is a tree if and only if there is a unique (simple) path between any pair of nodes.

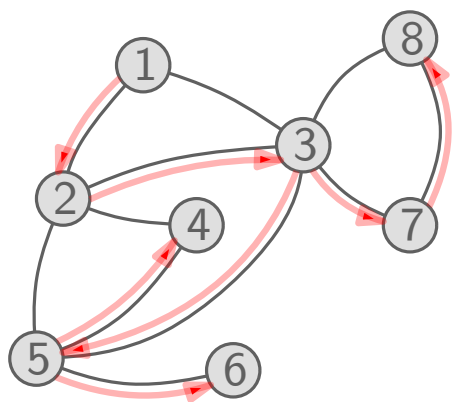
Proof. In a tree, any two nodes can have only one path between them. If there were more than one path, their union contains a cycle.
On the other hand, if a graph has a path between any pair of nodes, the graph is connected. Since the paths are unique, the graph is acyclic.

Depth-First Search

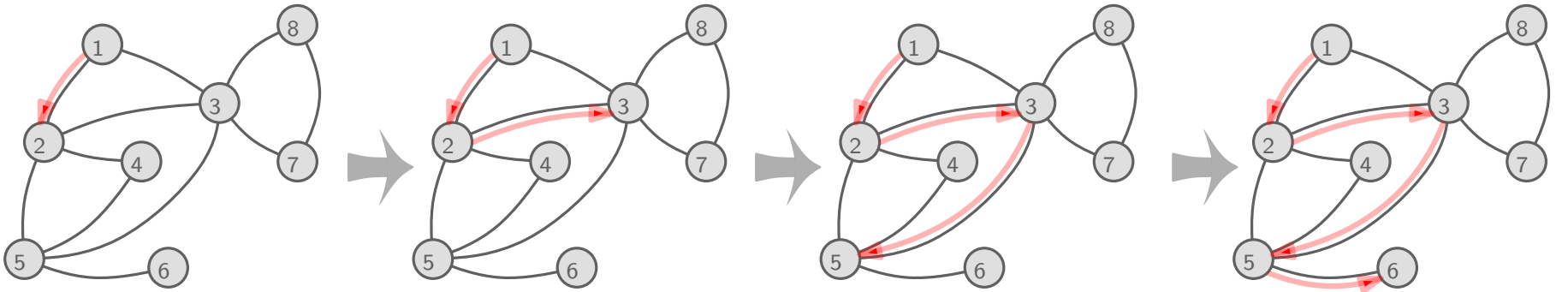
Q: What parts of the graph are reachable from a given vertex s?

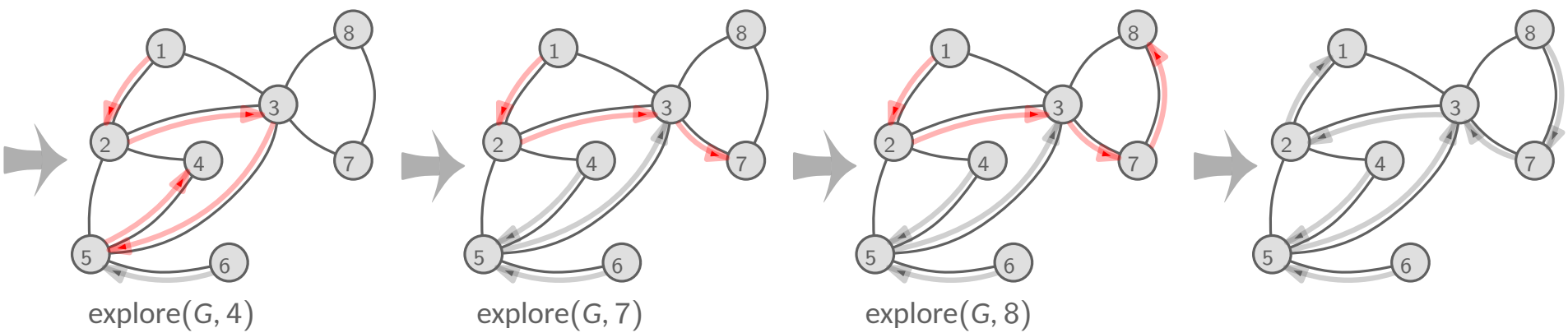
Explore outward from s in an unexplored direction until visiting any *terminal* or *explored node*. Retract to the previous node and continue exploring.

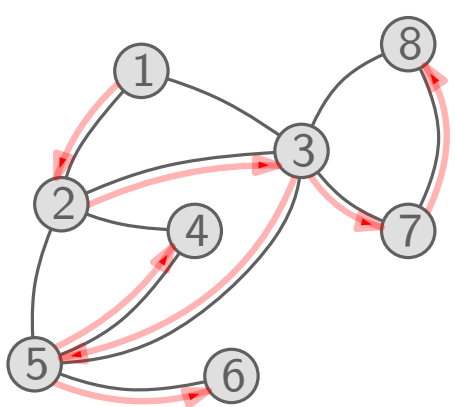
```
procedure explore(G, v)
visited(v) = true
for each edge (v, u) ∈ E do
    if not visited(u) then
        explore(G, u)
```



Depth-First Search







Depth-First Search

We need to confirm that the procedure **explore** always works correctly. It never jumps to a region that is not reachable from v, because it only moves from nodes to their neighbors.

Q: Does it find all vertices reachable from v?

Assume u is reachable from v but **explore** *misses* u.

- Let π be any path from v to u, and let $z \in \pi$ be the last vertex visited by **explore**.
- Let w be the node immediately after z on π.
- When z was visited, the procedure would have noticed w and moved on to it.

Over the course of the entire DFS, each edge $(u, v) \in E$ is examined *twice*,

- once during **explore**(u) and
- once during **explore**(v).

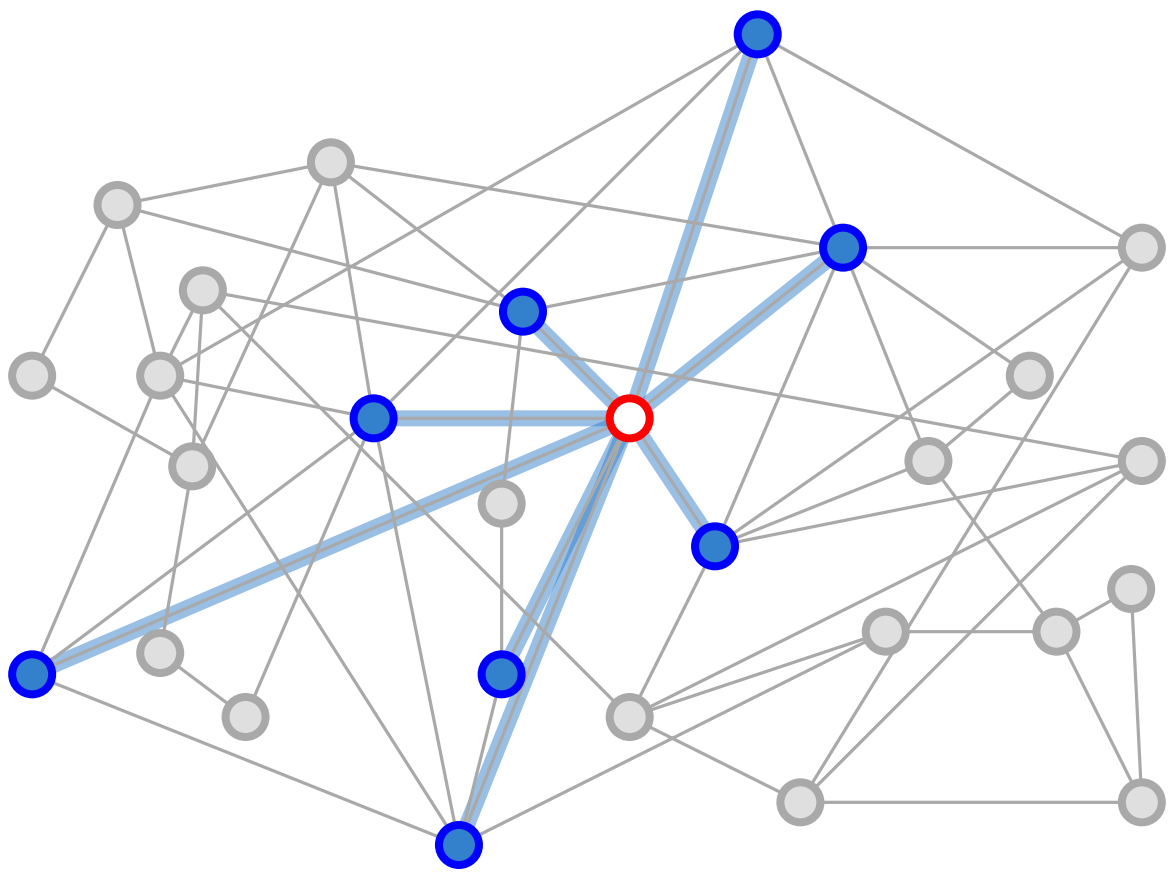
Thus, the total running time is $O(|V| + |E|)$.

```
procedure dfs(G)
for all v ∈ V do
    visited(v) =false
for all v ∈ V do
    if not visited(v) then
        explore(v)
```

```
procedure explore(G, v)
visited(v) =true
for each (v, u) ∈ E do
    if not visited(u) then
        explore(G, u)
```

Breadth-First Search

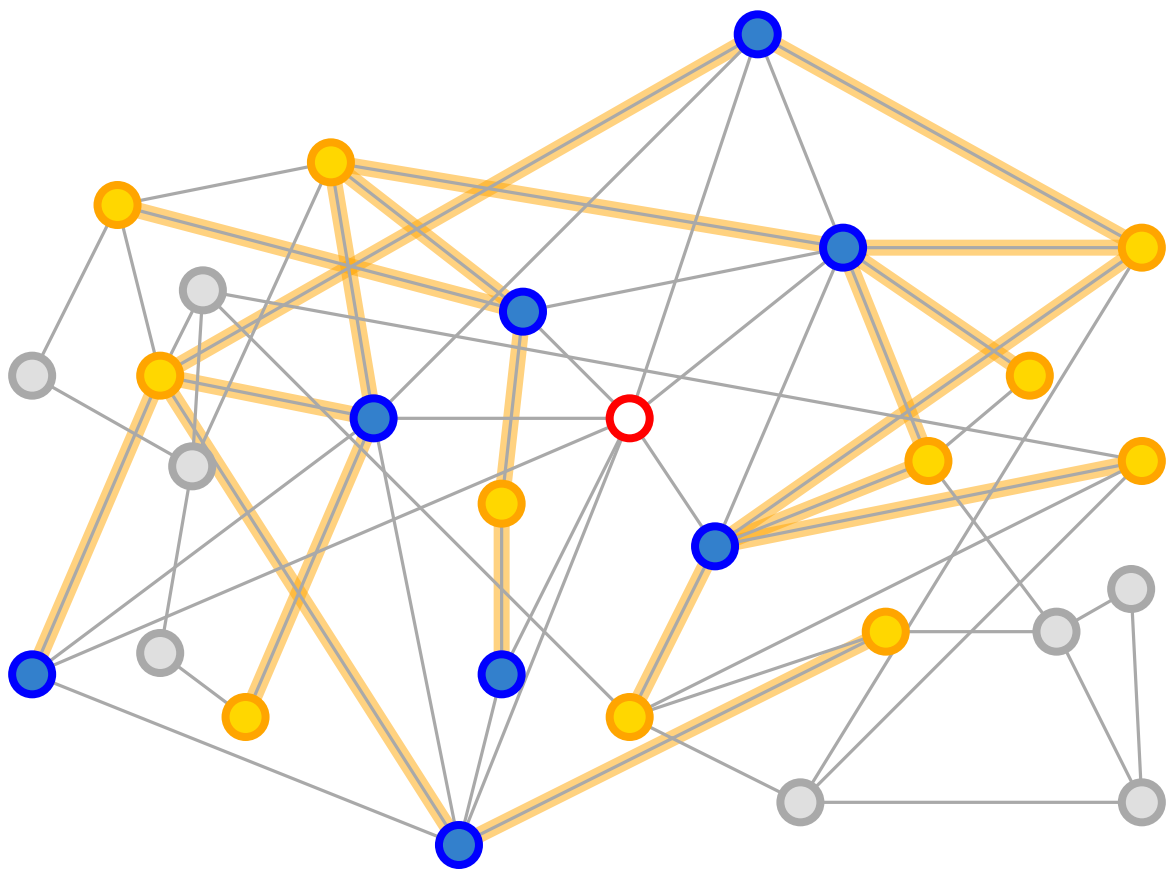
Explore outward from s in all possible directions layer by layer, and compute distances from s to the other vertices. Once the nodes at distance $0, 1, \ldots, d$ are chosen, the ones at $d + 1$ are the remaining nodes adjacent to the layer at d.



Layer 0 Layer 1

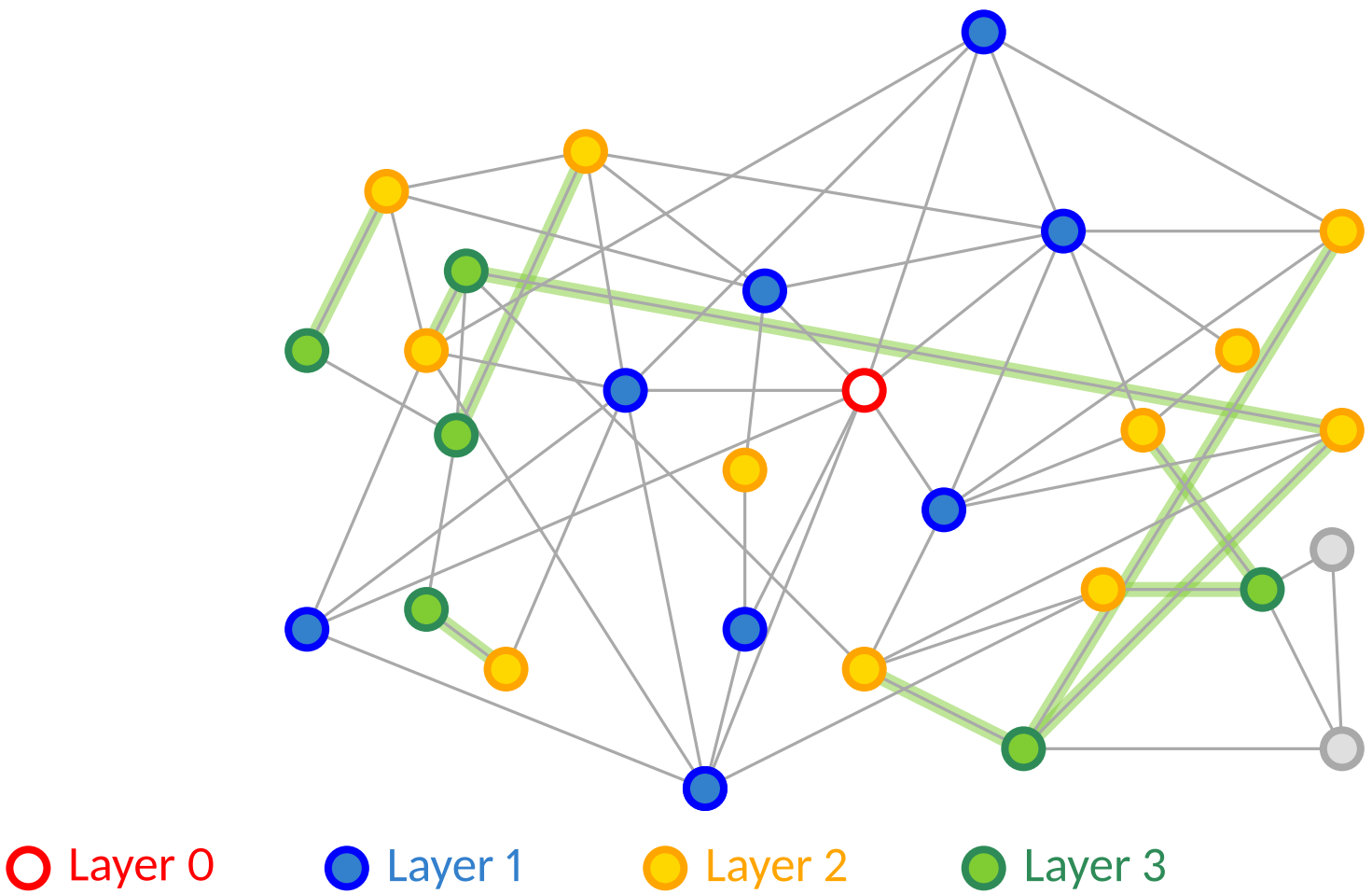
Breadth-First Search

Explore outward from s in all possible directions layer by layer, and compute distances from s to the other vertices. Once the nodes at distance $0, 1, \ldots, d$ are chosen, the ones at $d + 1$ are the remaining nodes adjacent to the layer at d.



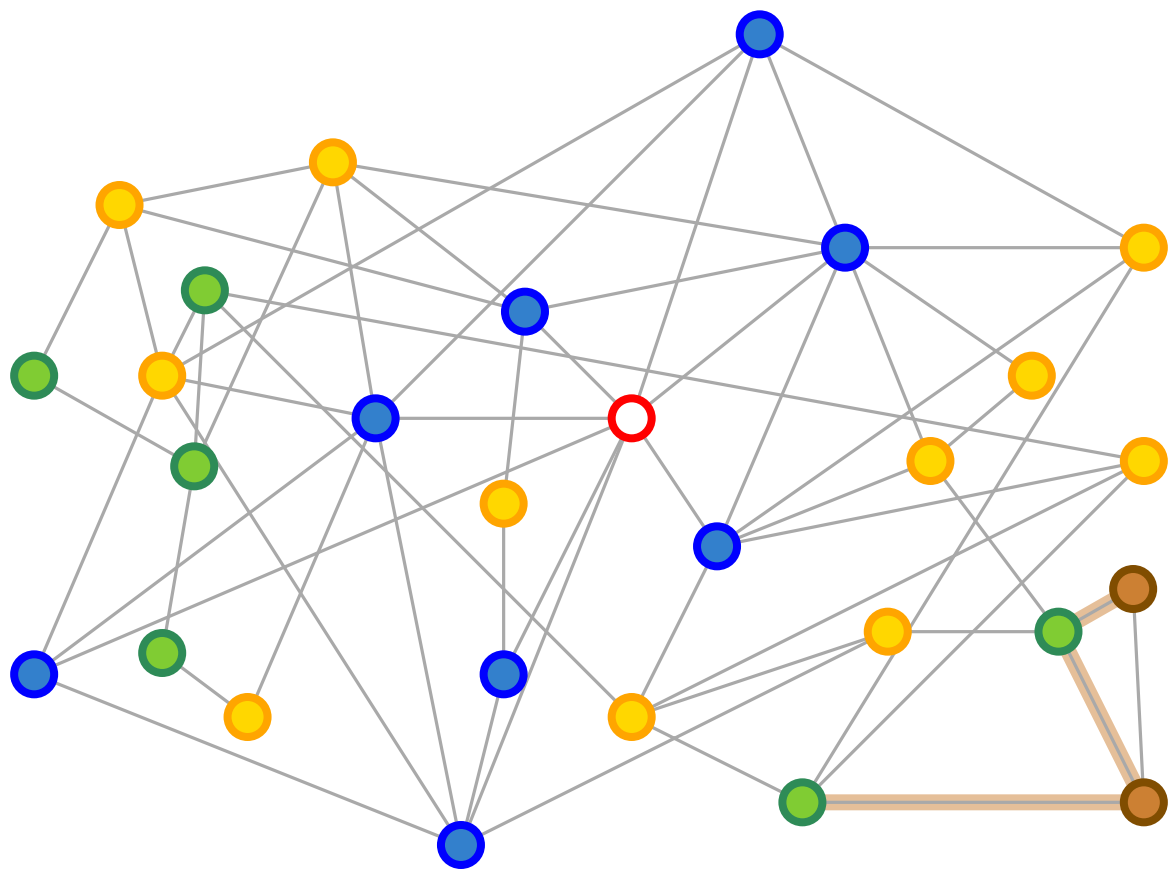
Breadth-First Search

Explore outward from s in all possible directions layer by layer, and compute distances from s to the other vertices. Once the nodes at distance $0, 1, \ldots, d$ are chosen, the ones at $d + 1$ are the remaining nodes adjacent to the layer at d.



Breadth-First Search

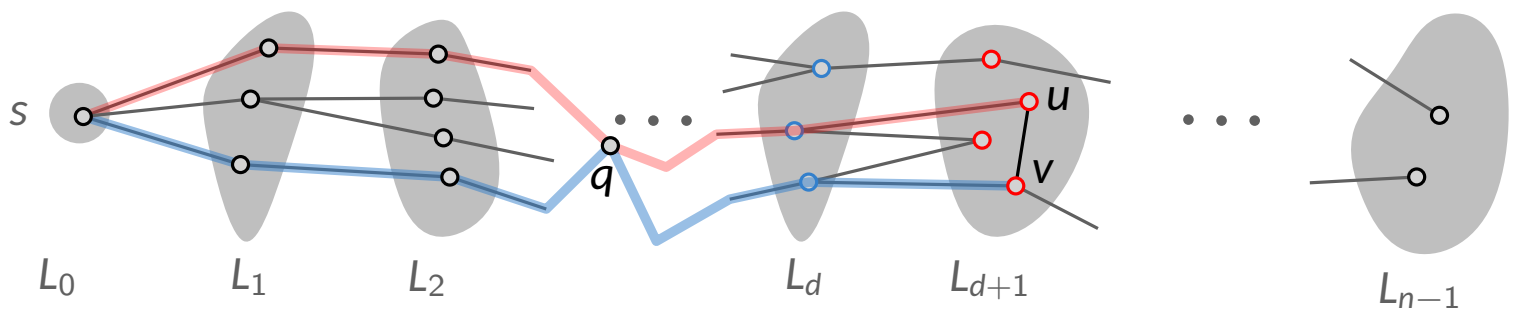
Explore outward from s in all possible directions layer by layer, and compute distances from s to the other vertices. Once the nodes at distance $0, 1, \ldots, d$ are chosen, the ones at $d + 1$ are the remaining nodes adjacent to the layer at d.



Layer 0 Layer 1 Layer 2 Layer 3 Layer 4

Breadth-First Search

Explore outward from s in all possible directions layer by layer, and compute distances from s to the other vertices. Once the nodes at distance $0, 1, \ldots, d$ are chosen, the ones at $d+1$ are the remaining nodes adjacent to the layer at d.



$L_0 = \{s\}$.

$L_1 =$ all neighbors of L_0.

$L_2 =$ all nodes not belonging to $L_0 \cup L_1$, but having an edge to a node in L_1.

$L_{i+1} =$ all nodes not belonging to any earlier layer, but having an edge to a node in L_i.

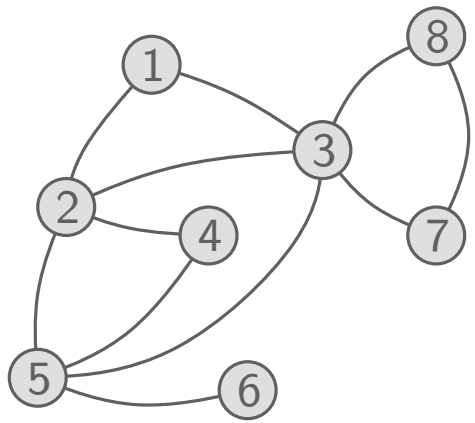
Observation. Each edge connects two nodes either in the same layer or in two consecutive layers.

Observation. If there is an edge connecting two nodes in the same layer, there is an odd-length cycle.

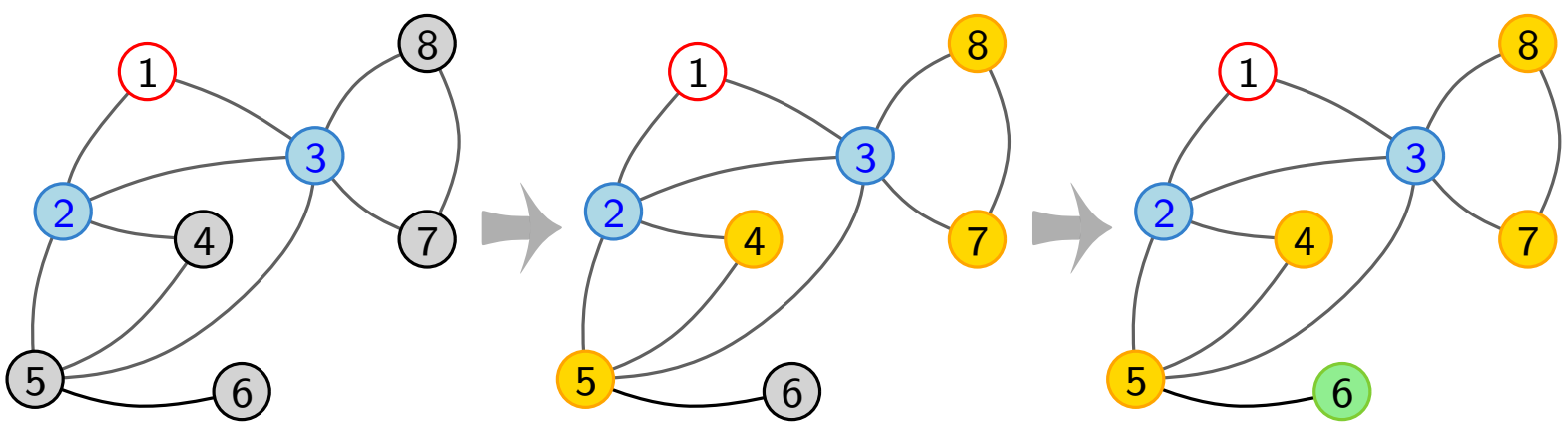
> Let (u, v) be an edge for u, v in the same layer. There is a cycle of odd length, consisting of (u, v) and two paths one from u to q and one from v to q, each consisting of i edges connecting nodes in consecutive layers, for some node q in an earlier layer.

Breadth-First Search

procedure **bfs**(G, s)
for all $u \in V$ **do**
  $\text{dist}(u) = \infty$
$\text{dist}(s) = 0$
$Q = [s]$ (queue containing just s)
while Q is not empty **do**
  $u = \text{eject}(Q)$
  for all edges $(u, v) \in E$ **do**
    if $\text{dist}(v) = \infty$ **then**
      $\text{inject}(Q, v)$
      $\text{dist}(v) = \text{dist}(u) + 1$



visit order	Queue
	[1]
1	[2 3]
2	[3 4 5]
3	[4 5 7 8]
4	[5 7 8]
5	[7 8 6]
7	[8 6]
8	[6]
6	[]



Running time is linear, $O(|V| + |E|)$.

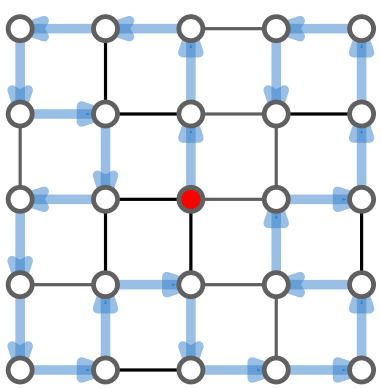
DFS vs. BFS

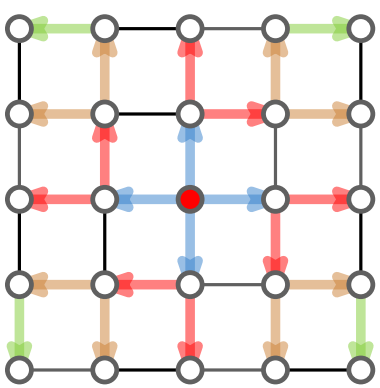
Depth-first search (DFS)

- makes deep incursions into a graph.
- uses stack (recursive calls).
- restarts the search in other connected components.

Breadth-first search (BFS)

- makes broader, shallower search (visiting vertices in increasing order of their distance from *s*).
- uses queue.
- ignores nodes not reachable from *s*.

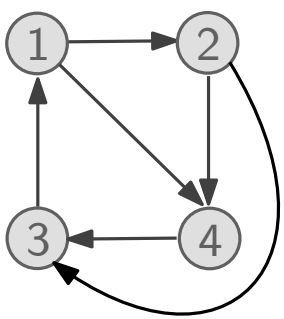


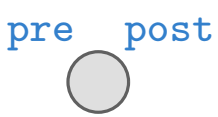


DFS on Directed graphs

Our depth-first search algorithm can be run on directed graphs, taking care to traverse edges only in their prescribed directions.

Terminology for important relations between nodes in the tree: **root, descendant, ancestor, parent, child**
Terminology for edges: **tree edge, forward edge, back edge, cross edge**

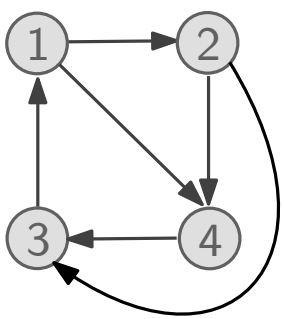


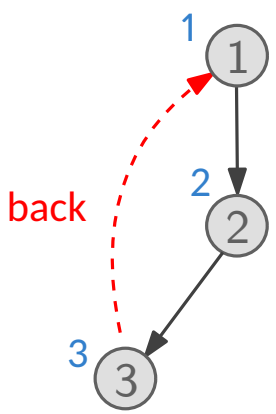


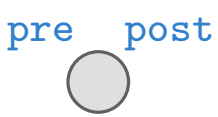
DFS on Directed graphs

Our depth-first search algorithm can be run on directed graphs, taking care to traverse edges only in their prescribed directions.

Terminology for important relations between nodes in the tree: **root, descendant, ancestor, parent, child**
Terminology for edges: **tree edge, forward edge, back edge, cross edge**



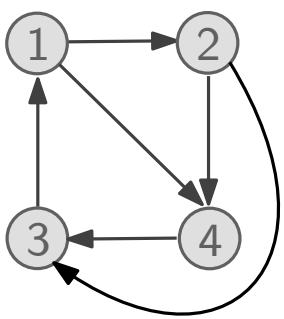


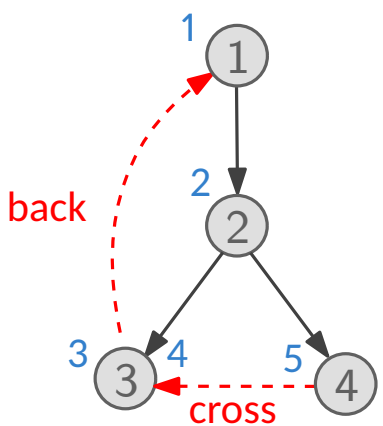


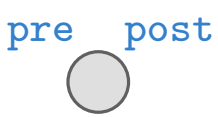
DFS on Directed graphs

Our depth-first search algorithm can be run on directed graphs, taking care to traverse edges only in their prescribed directions.

Terminology for important relations between nodes in the tree: **root, descendant, ancestor, parent, child**
Terminology for edges: **tree edge, forward edge, back edge, cross edge**



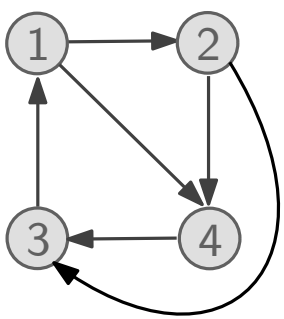


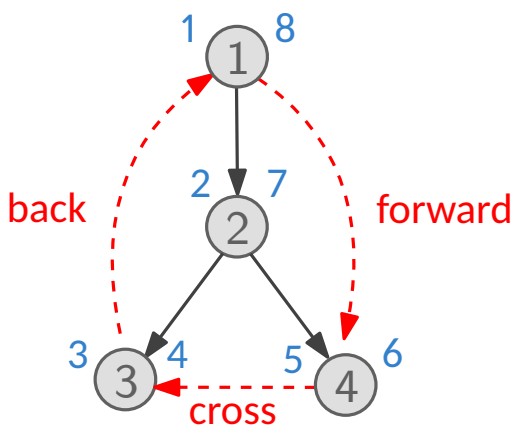


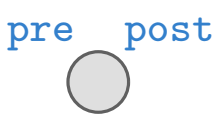
DFS on Directed graphs

Our depth-first search algorithm can be run on directed graphs, taking care to traverse edges only in their prescribed directions.

Terminology for important relations between nodes in the tree: **root, descendant, ancestor, parent, child**
Terminology for edges: **tree edge, forward edge, back edge, cross edge**



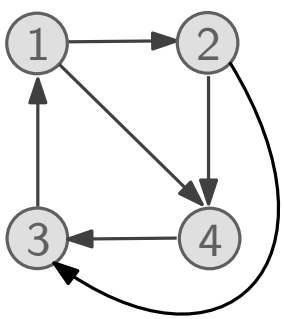


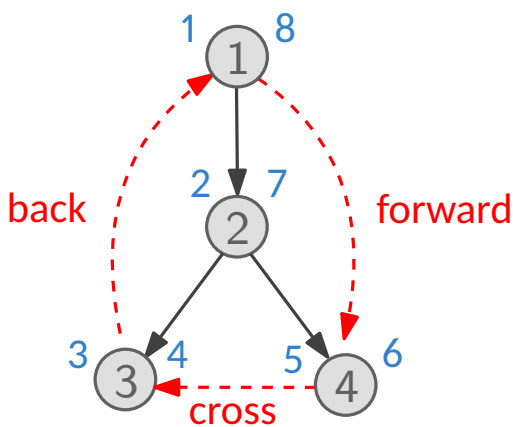


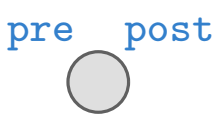
DFS on Directed graphs

Our depth-first search algorithm can be run on directed graphs, taking care to traverse edges only in their prescribed directions.

Terminology for important relations between nodes in the tree: **root, descendant, ancestor, parent, child**
Terminology for edges: **tree edge, forward edge, back edge, cross edge**







visit order for an edge $\overrightarrow{xy}$	edge
$x_{pre} < y_{pre} < y_{post} < x_{post}$	tree/forward
$y_{pre} < x_{pre} < x_{post} < y_{post}$	back
$y_{pre} < y_{post} < x_{pre} < x_{post}$	cross

DFS on Directed graphs

A cycle in a directed graph is a circular path

$$v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0.$$

A graph without cycles is acyclic, which can be tested in linear time, with a single depth-first search.

Property. A directed graph has a cycle if and only if its depth-first search reveals a back edge.

Directed acyclic graphs (dag) are directed graphs without cycles.
Dags can be linearized! - perform depth-first search in *decreasing* order of their `post` numbers.

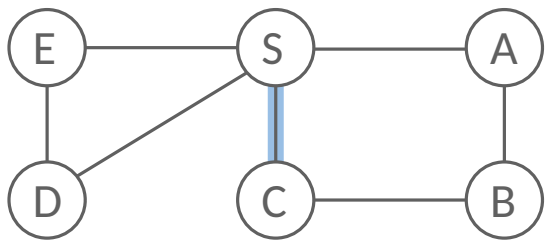
Property. In a dag, every edge leads to a vertex with a lower `post` number.

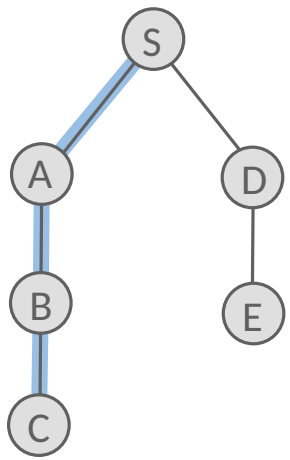
Property. Every dag has at least one source and at least one sink.

Shortest paths in graphs

The **distance** between two nodes is the **length of the shortest path** (the sum of lengths of edges in the path) between them.

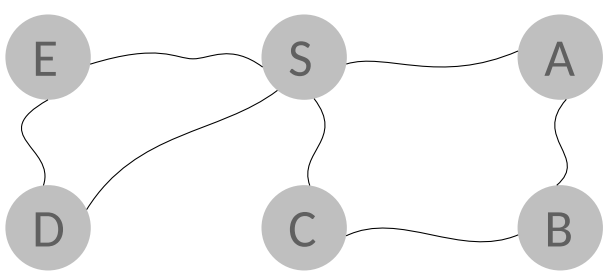
A simple graph and its depth-first search tree rooted at S.

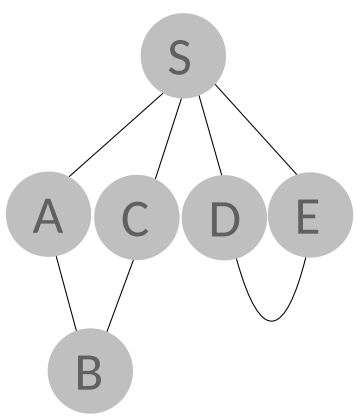




DFS search tree does not reflect the distances from *S* to other nodes correctly - the distance between S and C.

A physical model of a graph reflects the distances.

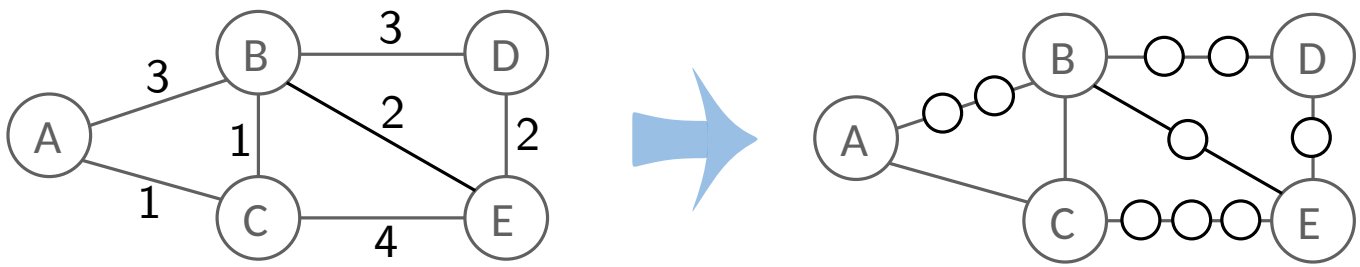




Dijkstra's algorithm

BFS finds shortest paths in any graph whose edges have unit length.
Can we adapt it to a more general graph whose edge lengths are positive integers?

1st trial:



BFS works! But when G has very long edges, BFS is inefficient as it spends most of its time computing distances to these dummy nodes.

2nd trial:

Use an alarm clock algorithm!

- Set $\text{alarm}(s) = 0$ and $\text{alarm}(v) = \infty$ for all the other nodes v.
- Repeat until there are no more alarms:

  Say the next alarm rings at time T, for node u. Then:
  - The distance from s to u is T.
  - For each neighbor v of u, $\text{alarm}(v) = \min\{\text{alarm}(v), T + \ell(u, v)\}$.

Dijkstra's algorithm

Implement the system of alarms using a priority queue (via a heap):

- `insert`(H, v). Add a new element v to the set.
- `decreasekey`(H, v). Accommodate the decease in key value of v.
- `deletemin`(H). Return the element with the smallest key, and remove it from H.
- `makequeue`(V). Build a priority queue out of the elements in V with key values.

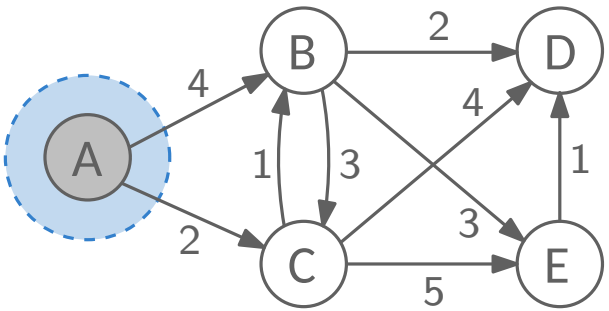
procedure **dijkstra**(G, ℓ, s)

for all $u \in V$ **do**
  $\texttt{dist}(u) = \infty$
  $\texttt{prev}(u) = \texttt{nil}$
$\texttt{dist}(s) = 0$
$H = \texttt{makequeue}(V)$
while H is not empty **do**
  $u = \texttt{deletemin}(H)$
  for all edges $(u, v) \in E$ **do**
    if $\texttt{dist}(v) > \texttt{dist}(u) + \ell(u, v)$ **then**
      $\texttt{dist}(v) = \texttt{dist}(u) + \ell(u, v)$
      $\texttt{prev}(v) = u$
      $\texttt{decreasekey}(H, v)$

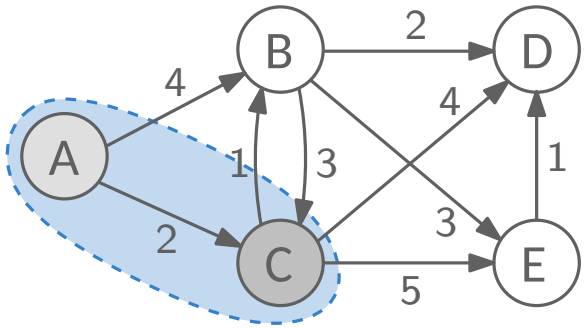
Input: $G = (V, E)$ is a (un)directed graph; positive edge lengths;
$s \in V$.

Output: $\forall$ vertices u reachable from s, $\text{dist}(u)$ =distance from s to u.

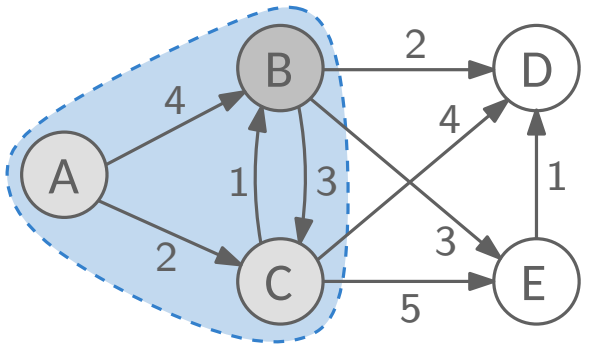
Dijkstra's algorithm



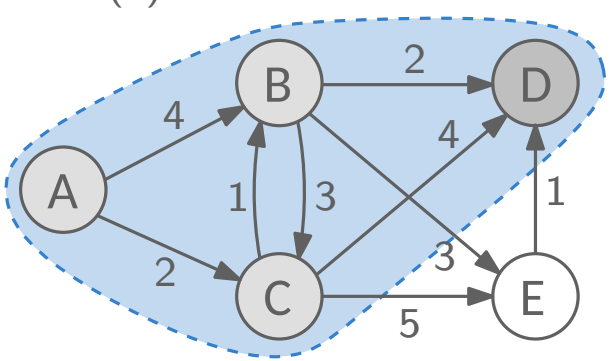
$\text{dist}(A) = 0$
$\text{dist}(B) = 4$
$\text{dist}(C) = 2$
$\text{dist}(D) = \infty$
$\text{dist}(E) = \infty$



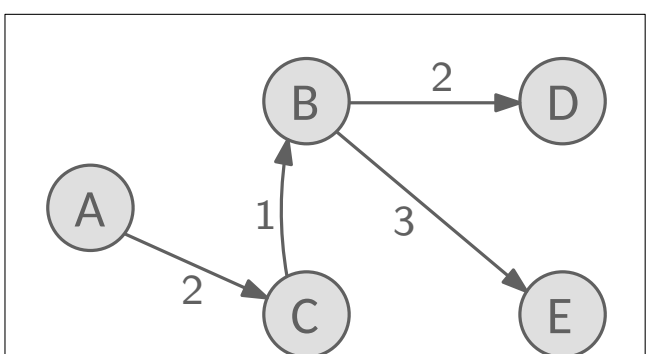
$\text{dist}(A) = 0$
$\text{dist}(B) = 3$
$\text{dist}(C) = 2$
$\text{dist}(D) = 6$
$\text{dist}(E) = 7$



$\text{dist}(A) = 0$
$\text{dist}(B) = 3$
$\text{dist}(C) = 2$
$\text{dist}(D) = 5$
$\text{dist}(E) = 6$

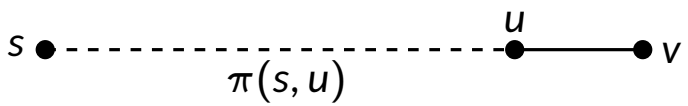


$\text{dist}(A) = 0$
$\text{dist}(B) = 3$
$\text{dist}(C) = 2$
$\text{dist}(D) = 5$
$\text{dist}(E) = 6$

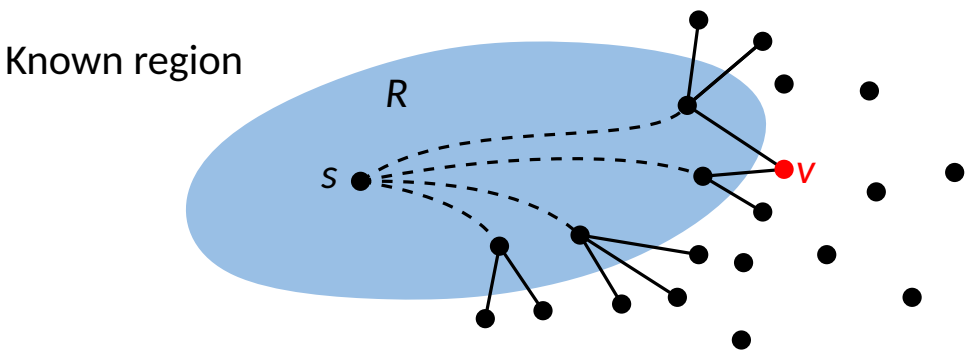


Shortest path tree

Dijkstra's algorithm

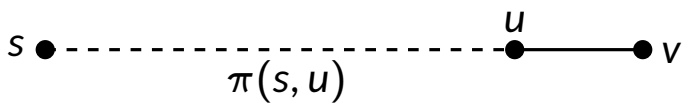


The shortest path from s to v is a known shortest path $\pi(s,u)$ extended by a single edge (u,v).

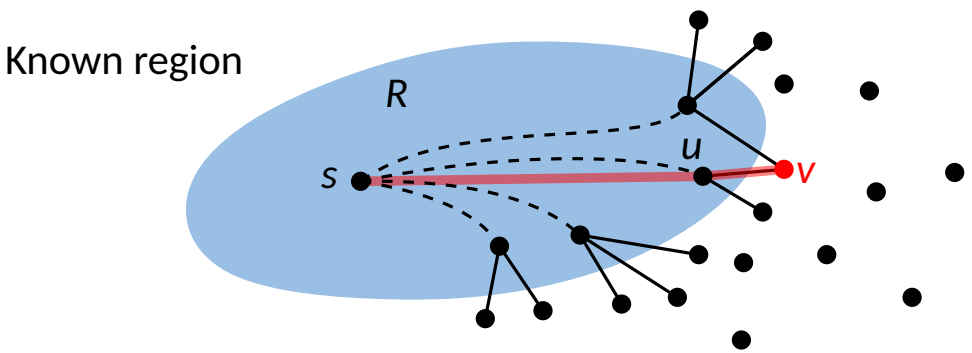


Among many single-edge extensions of the nodes in R to nodes outside R, the shortest (its endpoint) of these extended paths is the next node of R.

Dijkstra's algorithm

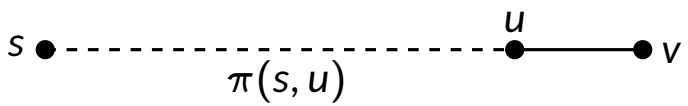


The shortest path from s to v is a known shortest path $\pi(s,u)$ extended by a single edge (u,v).

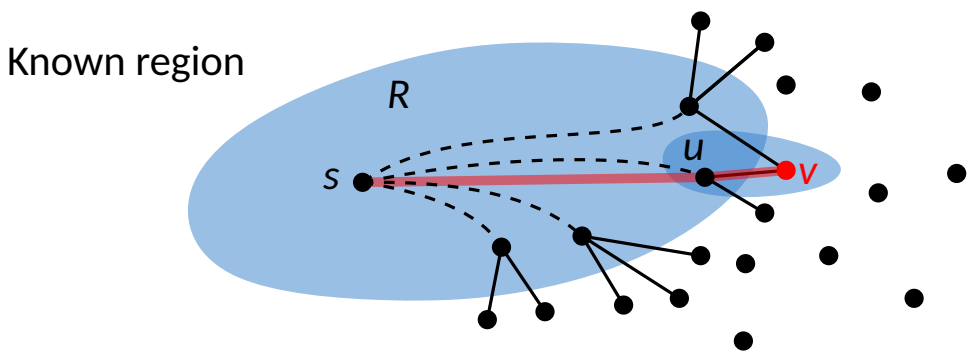


Among many single-edge extensions of the nodes in R to nodes outside R, the shortest (its endpoint) of these extended paths is the next node of R.

Dijkstra's algorithm



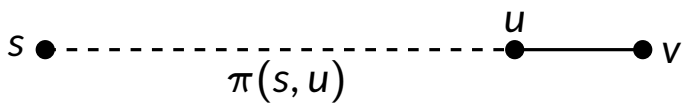
The shortest path from s to v is a known shortest path $\pi(s,u)$ extended by a single edge (u,v).



Among many single-edge extensions of the nodes in R to nodes outside R, the shortest (its endpoint) of these extended paths is the next node of R.

Among all extended paths (by an edge from nodes in R) from s to a node outside R, the shortest path from s to u extended by uv is the shortest. Therefore, this path is the shortest path from s to v. Do you see why there is no other path shorter than this path to v?

Dijkstra's algorithm



The shortest path from s to v is a known shortest path $\pi(s,u)$ extended by a single edge (u,v).

procedure **dijkstra**(G,ℓ,s)
for all $u \in V$ **do**
  $\texttt{dist}(u) = \infty$
$\texttt{dist}(s) = 0$
$R = \{\}$ (the "known region")
while $R \neq V$ **do**
  Pick the node $v \notin R$ with smallest $\texttt{dist}(\cdot)$
  Add v to R
  for all edges $(v,z) \in E$ **do**
    if $\texttt{dist}(z) > \texttt{dist}(v) + \ell(v,z)$ **then**
      $\texttt{dist}(z) = \texttt{dist}(v) + \ell(v,z)$

Dijkstra's algorithm

Dijkstra's algorithm is slower than BFS because the priority queue primitives are computationally more demanding.

- $|V|$ `insert` operations for `makequeue`.
- $|V|$ `deletemin` and $|V| + |E|$ `insert/decreasekey` operations.

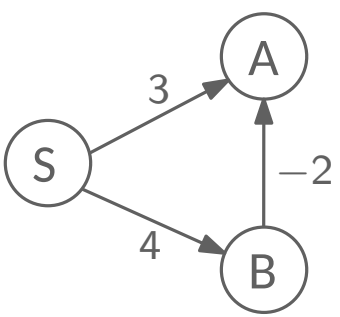
- **array**: $O(|V|)$ time for `deletemin`, $O(1)$ for `insert/decreasekey`. $O(|V|^2)$ time in total.
- **binary heap**: $O(\log |V|)$ time for `deletemin`, $O(\log |V|)$ for `insert/decreasekey`. $O((|V| + |E|) \log |V|)$ time in total.
- ***d*-ary heap**: efficient for both sparse and dense graphs with $d \approx |E|/|V|$. $O(d \log |V| / \log d)$ time for `deletemin`, $O(\log |V| / \log d)$ time for insert/decreasekey. $O((|V| \cdot d + |E|) \frac{\log |V|}{\log d})$ time in total.
- **Fibonacci heap**: $O(\log |V|)$ time for `deletemin`, $O(1)$ *amortized* time for `insert/decreasekey`. $O(|V| \log |V| + |E|)$ time in total. However, it requires considerably more work in implementation.

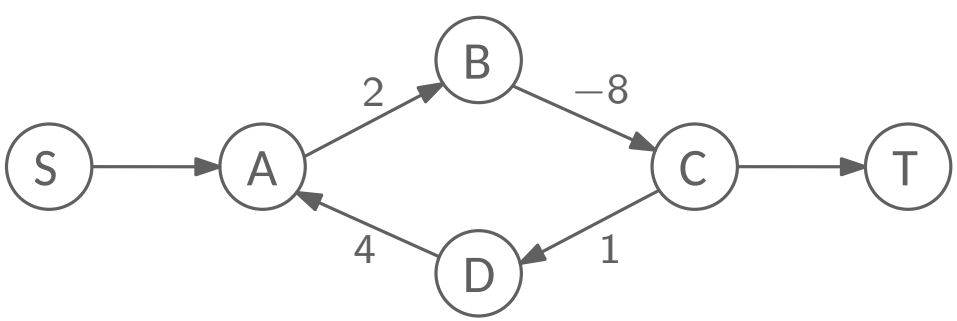
Negative Edges

One invariant of Dijkstra's algorithm is

> *The shortest path from the starting point s to any node v must pass exclusively through nodes that are closer than v.*

This doesn't work in the presence of negative edges.





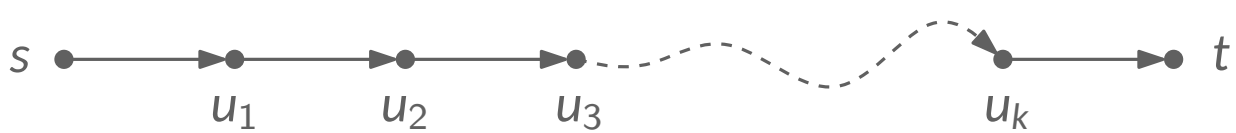
Negative Edges

But the **dist** values are always either overestimates or exactly correct.

procedure **update**$((u, v) \in E)$
if $\text{dist}(v) > \text{dist}(u) + \ell(u, v)$:
$\text{dist}(v) = \text{dist}(u) + \ell(u, v)$

It gives the correct (and final) $\text{dist}(v)$ value when

- u is the second last node of the shortest path to v and
- $\text{dist}(u)$ is correctly set.



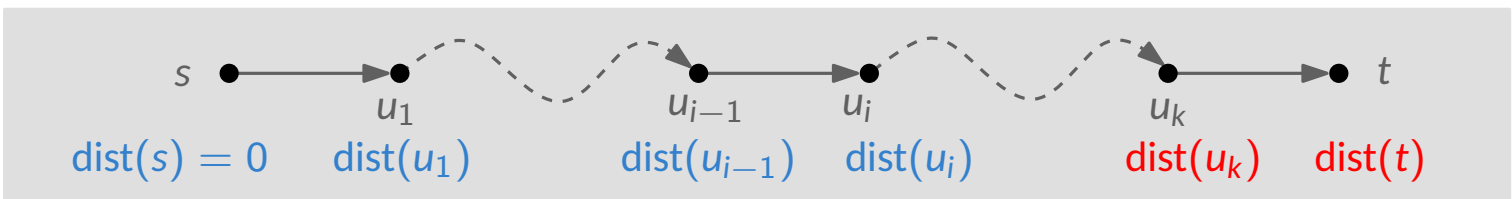
Consider the shortest path $\pi = su_1u_2u_3 \cdots u_kt$ from s to t. Then every subpath $\pi_i = su_1 \cdots u_i$ is

- the shortest path from s to u_i, otherwise we can decrease $\text{dist}(t)$.
- simple, otherwise there is a cycle and we can still decrease $\text{dist}(t)$.

Negative Edges

Consider the shortest path $\pi = su_1u_2u_3 \cdots u_kt$ from s to t. Then every subpath $\pi_i = su_1 \cdots u_i$ is

- the shortest path from s to u_i, otherwise we can decrease dist(t).
- simple, otherwise there is a cycle and we can still decrease dist(t).



- π_i consists of exactly i edges.
- dist(u_1) is set to the final value after updating all the edges once.
  s always has its final dist(s) value. Thus, all the nodes whose shortest paths from s consist of only one edge have their final dist values.
- dist(u_i) is set to the final value after updating all the edges i times.
  u_{i-1} has its final dist(u_{i-1}) value. Thus, all the nodes whose shortest paths from s consist of exactly i edges have their final dist values.

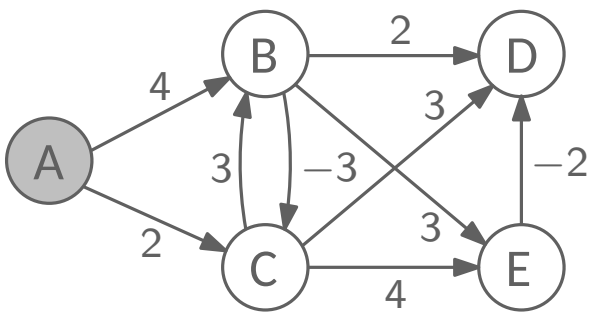
Thus, by simply update all the edges $|V| - 1$ times, we can compute the final dist values of all the nodes.

Negative Edges

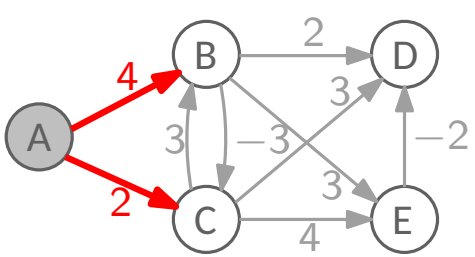
function **shortest-path**(G, ℓ, s)

for all $u \in V$ **do**
  dist(u) = ∞
  prev(u) = nil
dist(s) = 0
for $i = 1$ to $i = |V| - 1$ **do**
  for all $e \in E$ **do**
    update(e)

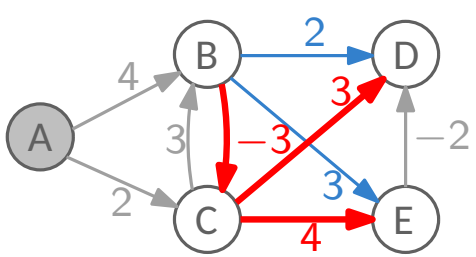
procedure **update**$((u, v))$
if dist(v) $>$ dist(u) + $\ell(u, v)$:
  dist(v) = dist(u) + $\ell(u, v)$
  prev(v) = u



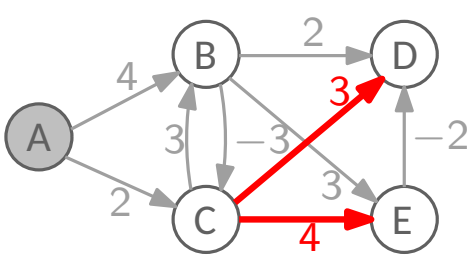
A : 0 / nil
B : ∞ / nil
C : ∞ / nil
D : ∞ / nil
E : ∞ / nil



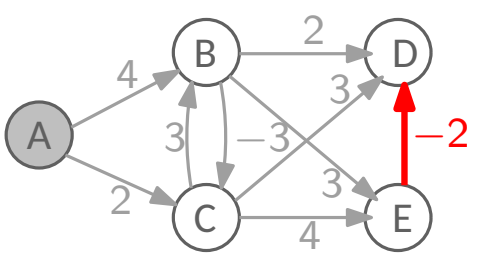
A : 0 / nil
B : 4 / A
C : 2 / A
D : ∞ / nil
E : ∞ / nil



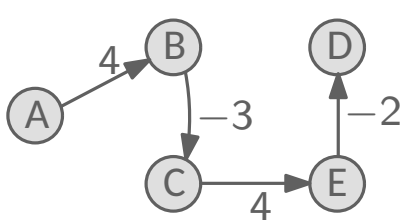
A : 0 / nil
B : 4 / A
C : 1 / B
D : 5 / C
E : 6 / C



A : 0 / nil
B : 4 / A
C : 1 / B
D : 4 / C
E : 5 / C



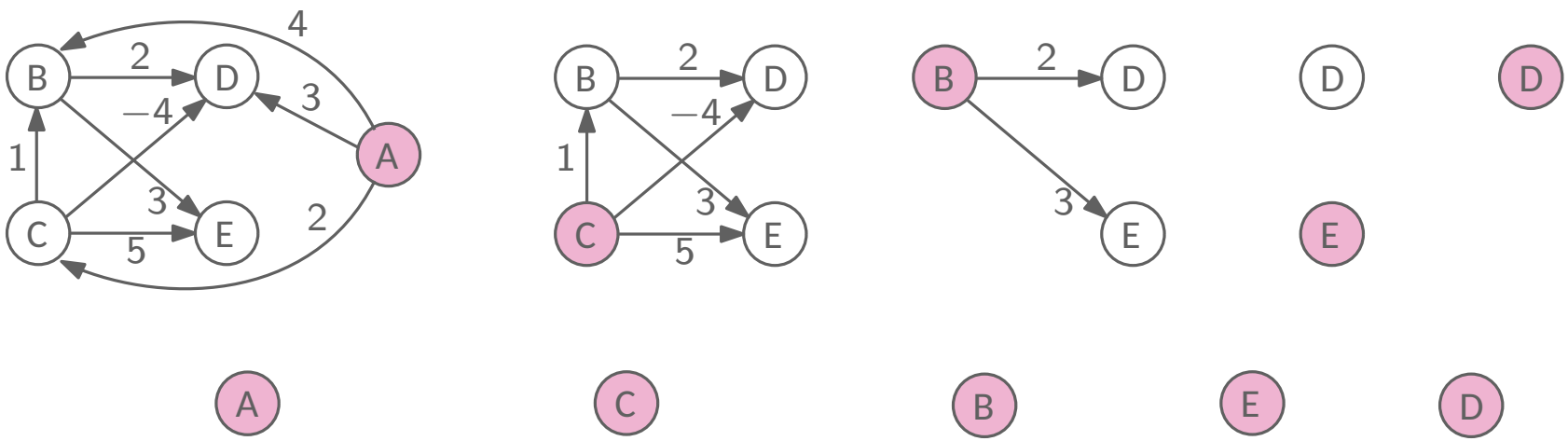
A : 0 / nil
B : 4 / A
C : 1 / B
D : 3 / E
E : 5 / C



Shortest path tree

Directed Acyclic Graphs

A directed acyclic graph (DAG) is a directed graph with no directed cycle.

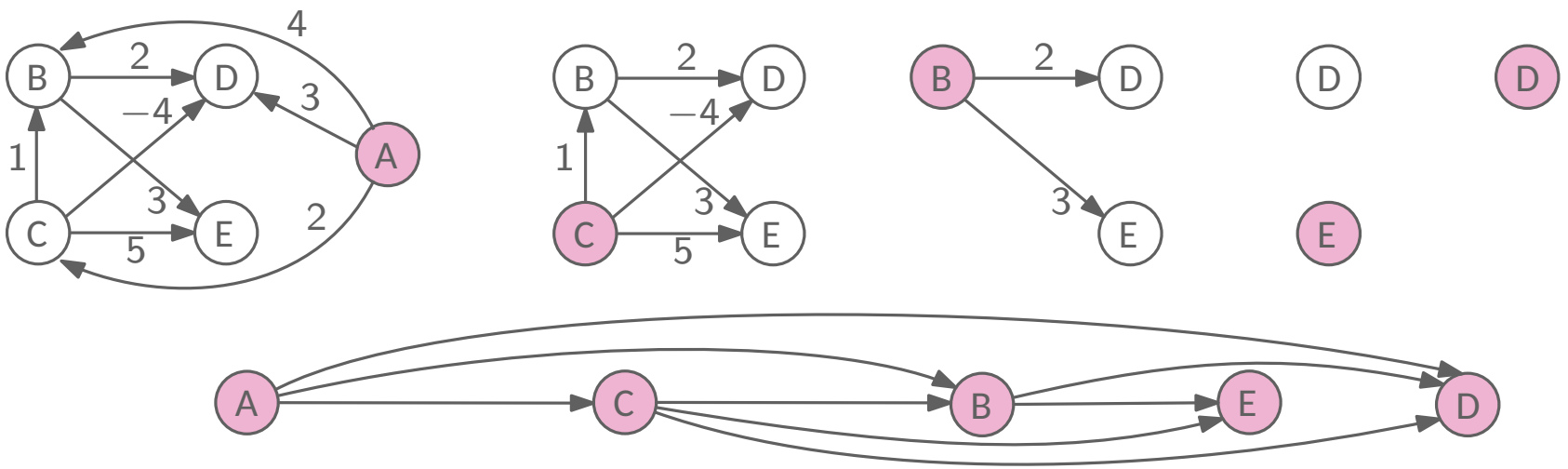


Topological ordering of the vertices: for every directed edge $e = (u, v)$, u occurs earlier than v in the ordering.

Every DAG has at least one vertex with no incoming edge. Can you see why?
$\rightarrow$ A topological ordering can be obtained by removing a vertex (with its outgoing edges) with no incoming edge repeatedly.

Directed Acyclic Graphs

A directed acyclic graph (DAG) is a directed graph with no directed cycle.



Topological ordering of the vertices: for every directed edge $e = (u, v)$, u occurs earlier than v in the ordering.

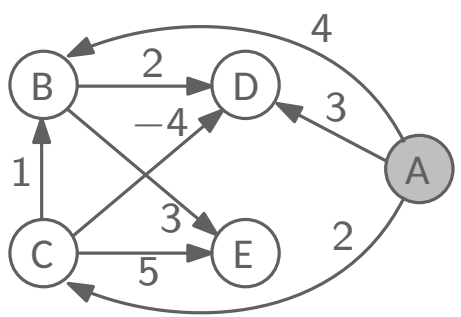
Every DAG has at least one vertex with no incoming edge. Can you see why?
$\rightarrow$ A topological ordering can be obtained by removing a vertex (with its outgoing edges) with no incoming edge repeatedly.

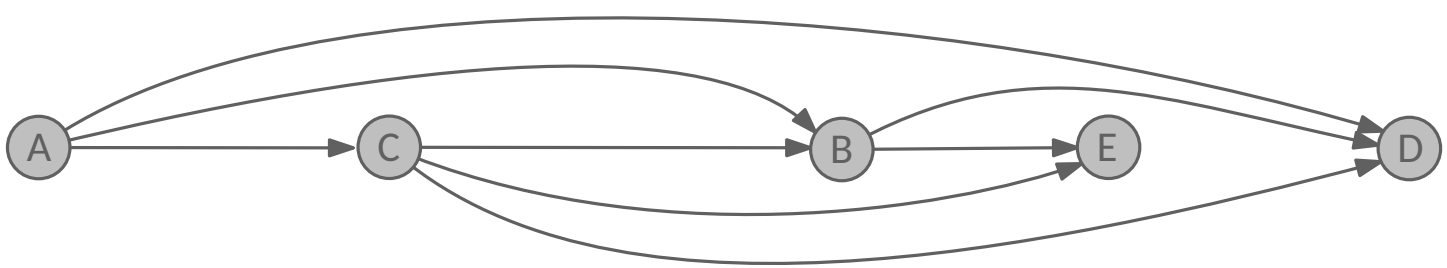
A directed graph is a DAG iff it has a topological ordering.

Directed Acyclic Graphs

function **dag-shortest-path**(G, ℓ, s)
for all $u \in V$ **do**
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0
Linearize G
for each $u \in V$, in linearized order **do**
    for all edges $(u, v) \in E$ **do**
        update(u, v)

procedure **update**$((u, v))$
if dist(v) $>$ dist(u) + $\ell(u, v)$:
        dist(v) = dist(u) + $\ell(u, v)$
        prev(v) = u



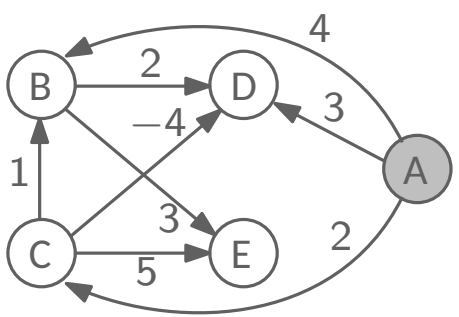


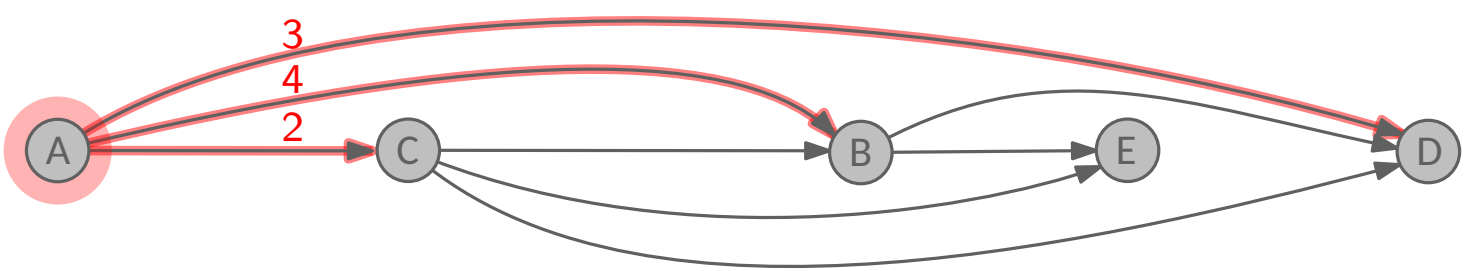
dist(A) = 0 dist(C) = ∞ dist(B) = ∞ dist(E) = ∞ dist(D) = ∞

Directed Acyclic Graphs

function **dag-shortest-path**(G, ℓ, s)
for all $u \in V$ **do**
    dist$(u) = \infty$
    prev(u) = nil
dist$(s) = 0$
Linearize G
for each $u \in V$, in linearized order **do**
    for all edges $(u, v) \in E$ **do**
        update(u, v)

procedure **update**$((u, v))$
if dist$(v) >$ dist$(u) + \ell(u, v)$:
        dist(v) = dist$(u) + \ell(u, v)$
        prev$(v) = u$





dist$(A) = 0$

dist$(C) = \infty$
dist$(C) = 2$

dist$(B) = \infty$
dist$(B) = 4$

dist$(E) = \infty$

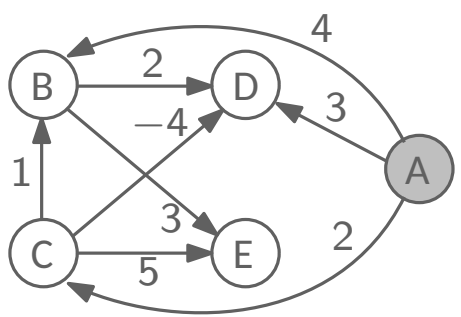
dist$(D) = \infty$
dist$(D) = 3$

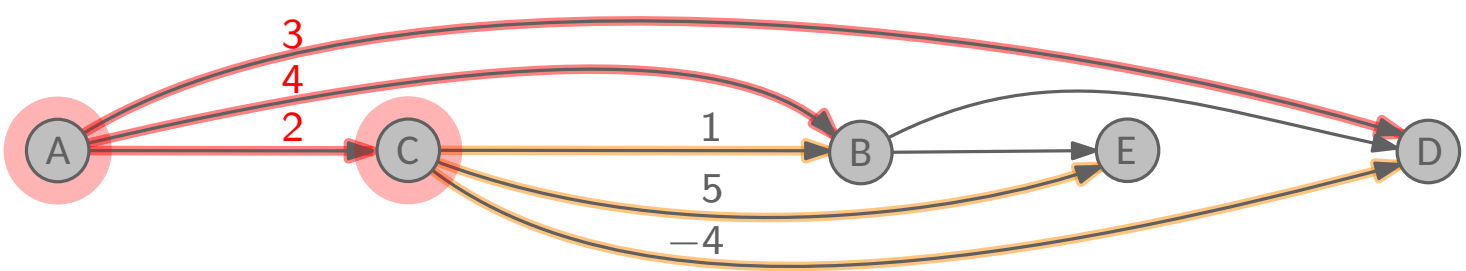
Directed Acyclic Graphs

function **dag-shortest-path**(G, ℓ, s)

for all $u \in V$ **do**
  dist(u) = ∞
  prev(u) = nil
dist(s) = 0
Linearize G
for each $u \in V$, in linearized order **do**
  for all edges $(u, v) \in E$ **do**
    update(u, v)

procedure **update**$((u, v))$
if dist(v) $>$ dist(u) + $\ell(u, v)$:
  dist(v) = dist(u) + $\ell(u, v)$
  prev(v) = u





dist(A) = 0

dist(C) = ∞
dist(C) = 2

dist(B) = ∞
dist(B) = 4
dist(B) = 3

dist(E) = ∞
dist(E) = 7

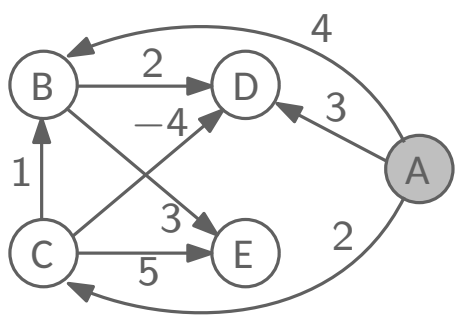
dist(D) = ∞
dist(D) = 3
dist(D) = -2

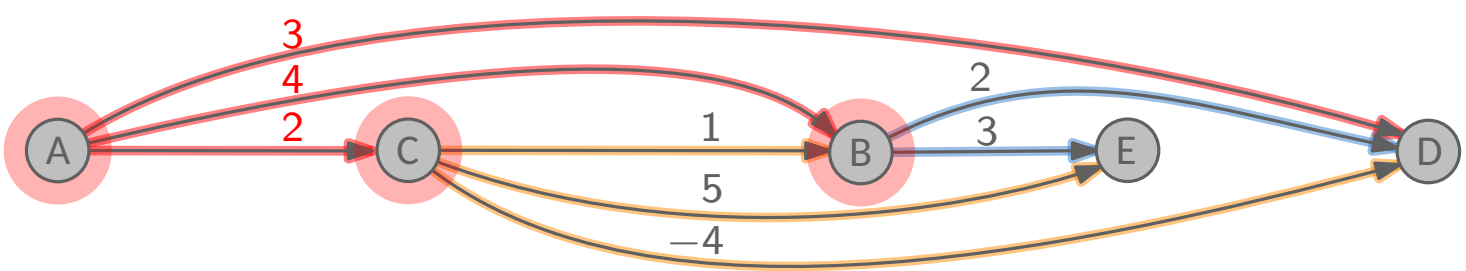
Directed Acyclic Graphs

function **dag-shortest-path**(G, ℓ, s)

for all $u \in V$ **do**
  dist(u) = ∞
  prev(u) = nil
dist(s) = 0
Linearize G
for each $u \in V$, in linearized order **do**
  for all edges $(u, v) \in E$ **do**
    update(u, v)

procedure **update**$((u, v))$
if dist(v) $>$ dist(u) + $\ell(u, v)$:
  dist(v) = dist(u) + $\ell(u, v)$
  prev(v) = u





dist(A) = 0

dist(C) = ∞
dist(C) = 2

dist(B) = ∞
dist(B) = 4
dist(B) = 3

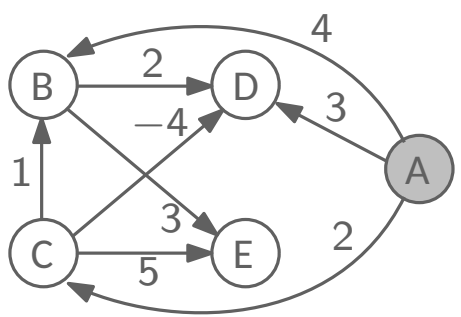
dist(E) = ∞
dist(E) = 7
dist(E) = 6

dist(D) = ∞
dist(D) = 3
dist(D) = −2

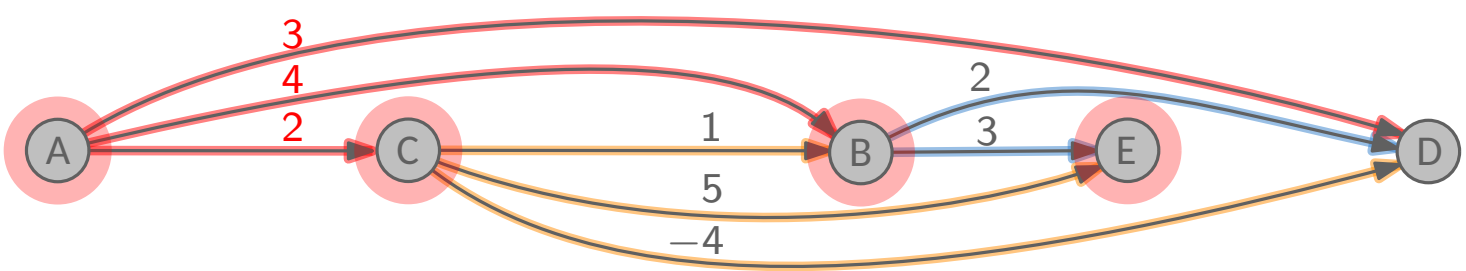
Directed Acyclic Graphs

function **dag-shortest-path**(G, ℓ, s)
for all $u \in V$ **do**
  dist$(u) = \infty$
  prev(u) = nil
dist$(s) = 0$
Linearize G
for each $u \in V$, in linearized order **do**
  for all edges $(u, v) \in E$ **do**
    update(u, v)

procedure **update**$((u, v))$
if dist$(v) >$ dist$(u) + \ell(u, v)$:
  dist(v) = dist$(u) + \ell(u, v)$
  prev$(v) = u$



$O(|V| + |E|)$ time



dist$(A) = 0$

dist$(C) = \infty$
dist$(C) = 2$

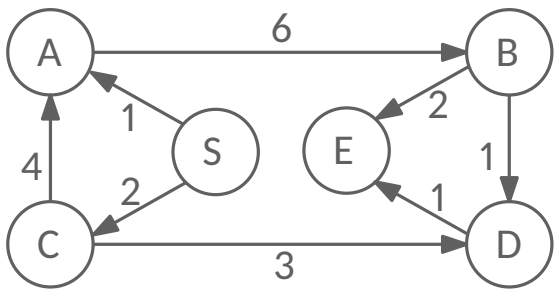
dist$(B) = \infty$
dist$(B) = 4$
dist$(B) = 3$

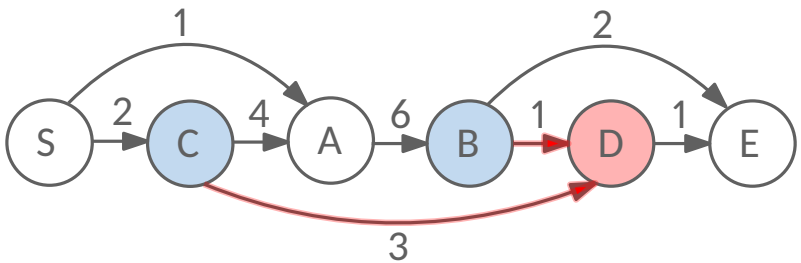
dist$(E) = \infty$
dist$(E) = 7$
dist$(E) = 6$

dist$(D) = \infty$
dist$(D) = 3$
dist$(D) = -2$

Shortest Paths in DAGs

DAG (Directed Acyclic Graphs) $\Rightarrow$ nodes can be linearized.





The only way to get to D is through its predecessors, B or C.

$$\text{dist(D)} = \min\{\text{dist(B)} + 1, \text{dist(C)} + 3\}.$$

```
initialize all dist[·] values to ∞
dist[s] = 0
for each v ∈ V\{s}, in linearized order:
    for each (u, v) ∈ E
        if dist[u] + l(u, v) < dist[v]
            dist[v] = dist[u] + l(u, v)
```

Time complexity: $O(n + m)$.
Space complexity: $O(n)$
(in addition to input).

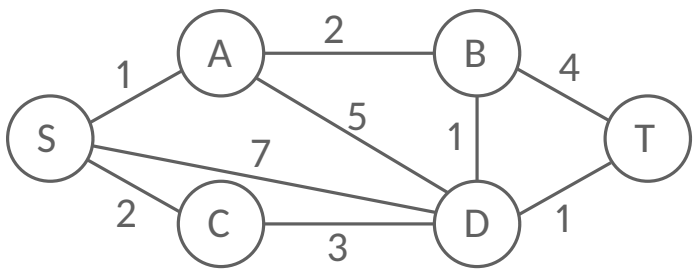
if $\text{dist}[u] + l(u, v) > \text{dist}[v]$
$\Rightarrow$ Longest path!

Shortest Reliable Paths

Input : a graph G, along with two nodes s and t and an integer k.
Goal : find a shortest path from s to t that uses *at most* k edges.

For example, consider a shortest path from s to t.

- $k = 2$: SDT of length 8.
- $k = 3$: SCDT of length 6.
- $k = 4$: SABDT of length 5.



What is the appropriate subproblem ?

> all vital information (number of edges allowed, shortest path length) is *remembered* and *forwarded*...

Dijkstra's algorithm does not remember the number of edges in the path...

Let $\text{dist}(v, i)$ be the **shortest path from** s **to** v **that uses** i **edges** with $\text{dist}(v, 0) = \infty$ and $\text{dist}(s, *) = 0$.

$$\text{dist}(v, i) = \min_{(u,v)\in E}\{\text{dist}(u, i-1) + \ell(u, v)\}$$

The set of $\text{dist}(v, i)$s, together with E, forms a dag!
Once we have the subproblem, the algorithm is straight-forward.
What's the running time?

All Pairs Shortest Paths

A simple solution would be to use an algorithm for "single-source shortest paths" for every starting node.

- For unweighted graphs, BFS takes $O(VE) = O(V^3)$ time.
- For acyclic graphs, scanning the vertices in topological order takes $O(VE) = O(V^3)$ time.
- For nonnegatively weighted graphs, Dijkstra's algorithm takes $O(VE \log V) = O(V^3 \log V)$ time.
- For general weighted graphs, Bellman-Ford algorithm takes $O(V^2E) = O(V^4)$ time.

But there is a better alternative using dynamic programming.

What is the appropriate subproblem ?

all vital information is *remembered* and *forwarded*...

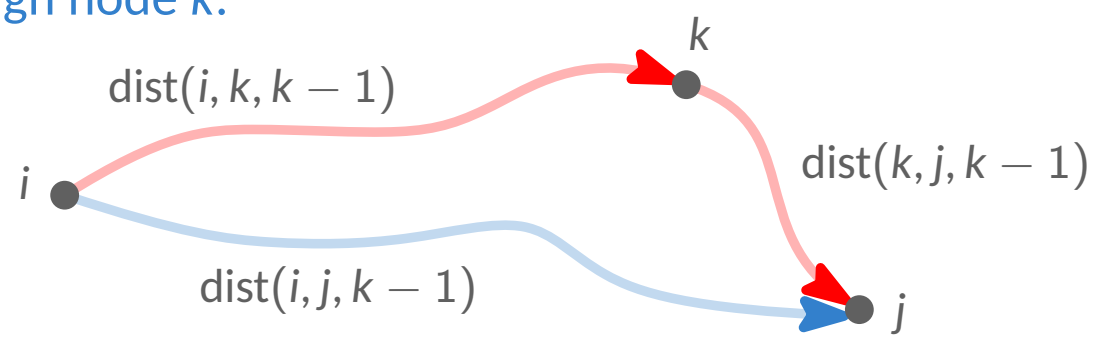
Consider a pair of nodes, i and j, and their shortest path.

Here we focus on the set of **permissible intermediate nodes.**
$S_k = \{1, 2, \dots, k\}$: the set of permissible intermediate nodes
$\text{dist}(i, j, k)$: the shortest path from i to j using only nodes in S_k.
We gradually expand the set S_k, from $\emptyset$ to V.
Does the set of $\text{dist}(i, j, k)$ form a dag?

All Pairs Shortest Paths

What happens when we include a node k?

There are two possible options: either (1) $\text{dist}(i, j, k)$ passes through node k or (2) it does not pass through node k.



$$\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k-1) + \text{dist}(k, j, k-1), \text{dist}(i, j, k-1)\}$$

```
for i = 1 to n
    for j = 1 to n
        dist(i, j, 0) = ∞
for all (i, j) ∈ E
    dist(i, j, 0) = l(i, j)
```

Time complexity: $O(n^3)$.
Space complexity: $O(n^3)$.

```
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            dist(i, j, k) = min{dist(i, k, k − 1) + dist(k, j, k − 1), dist(i, j, k − 1)}
```

The Traveling Salesman Problem

Input : n cities, $N = \{1, \dots, n\}$, the hometown being 1, and the intercity distance values d_{ij} for all i, j.
Goal : find a tour that starts and ends at 1, visits all cities exactly once, and has minimum total length.

It is highly unlikely to be solvable in polynomial time.

What is the appropriate subproblem ?

Imagine that our tour started at 1, and now has reached node j.
What information do we need?
the set S of cities visited so far, and the shortest path length from 1 to j.

$C(S, j) =$
the length of shortest path, from 1 to j, visiting each node in S exactly once.

$$C(S, j) = \min_{i \in S, i \neq j} \{C(S - \{j\}, i) + d_{ij}\}$$

The Traveling Salesman Problem

$C(\{1\}, 1) = 0$
for $s = 2$ to n
  for all subsets $S \subseteq \{1, 2, \cdots, n\}$ of size s and containing 1
    $C(S, 1) = \infty$
    for all $j \in S, j \neq 1$
      $C(S, j) = \min\{C(S - \{j\}, i) + d_{ij} : i \in S : i \neq j\}$
return $\min_j C(\{1, \cdots, n\}, j) + d_{j1}$

subproblems?

Total running time?