COMPSCI 220: Introduction to Graph Algorithms

Michael J. Dinneen

Email: mjd@cs.auckland.ac.nz
Room: 303S.579; Ext: 87868

http://www.cs.auckland.ac.nz/∼mjd

August–September 2007

Outline

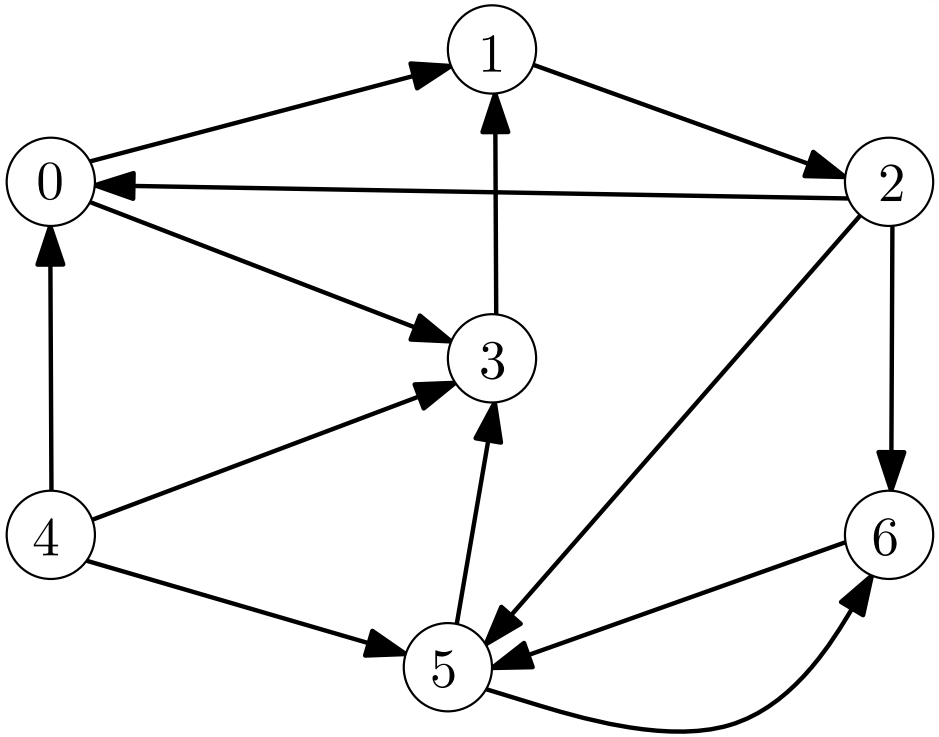
Digraphs

Definition

A digraph $G = (V, E)$ is a finite nonempty set V of nodes together with a (possibly empty) set E of ordered pairs of nodes of G called arcs.

$$V = \{0, 1, 2, 3, 4, 5, 6\}$$

$$E = \{(0,1),(0,3),(1,2),(2,0),(2,5),(2,6),(3,1),(4,0),(4,3),(4,5),(5,3),(5,6),(6,5)\}$$



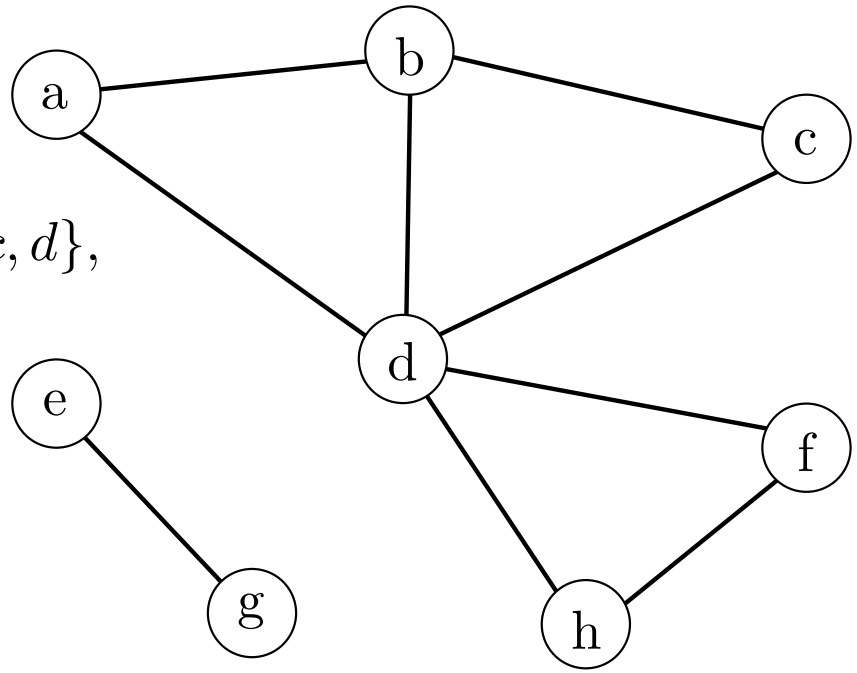
Graphs

Definition

A graph $G = (V, E)$ is a finite nonempty set V of vertices together with a (possibly empty) set E of unordered pairs of vertices of G called edges.

$$V = \{a, b, c, d, e, f, g, h\}$$

$$E = \{\{a, b\}, \{b, d\}, \{a, d\}, \{b, c\}, \{c, d\}, \{d, f\}, \{d, h\}, \{f, h\}, \{e, g\}\}$$



Basic definitions of [di]graphs

Let $G = (V, E)$ be a digraph then:

- If $(u, v) \in E$, we say that v is adjacent to u, that v is an out-neighbor of u, and that u is an in-neighbor of v.
- The order is $|V|$, the number of nodes. (often denoted by variable n)
- The size is $|E|$, the number of arcs. (often denoted by variable m or e)
- The out-degree of a node v is the number of arcs leaving v, and the in-degree of v is the number of arcs entering v.
- A node of in-degree 0 is called a source and a node of out-degree 0 is called a sink.

Walks, paths and cycles

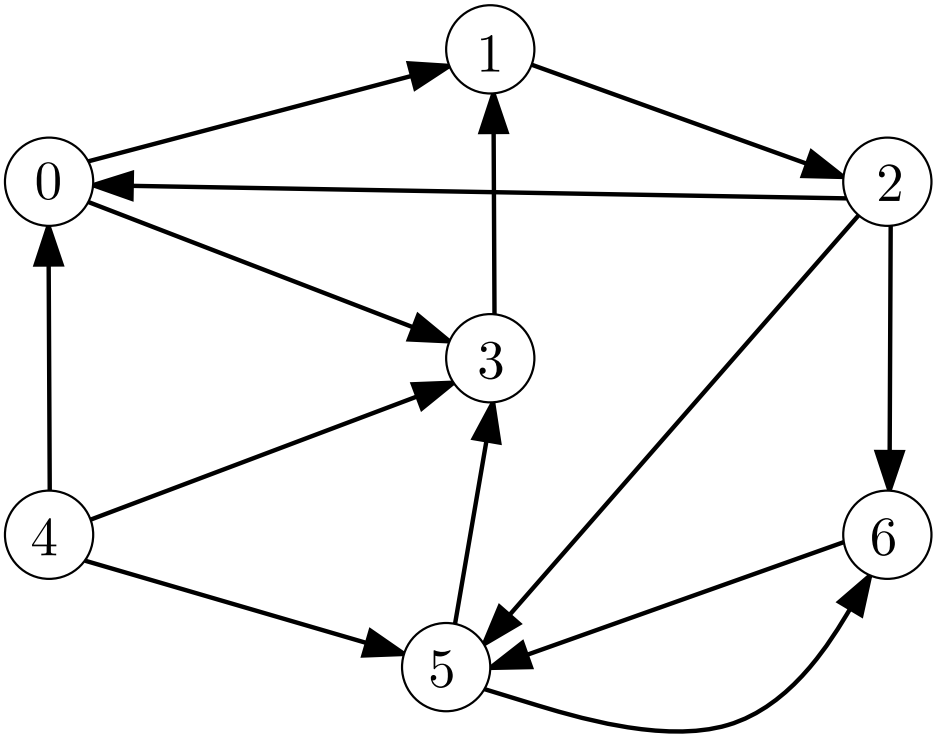
Definition

A walk in a digraph G is a sequence of nodes $v_0\ v_1\ \ldots\ v_n$ such that, for each i with $0 \leq i < n$, (v_i, v_{i+1}) is an arc in G. The length of the walk $v_0\ v_1\ \ldots\ v_n$ is the number n (that is, the number of arcs involved).
A path is a walk in which no node is repeated. A cycle is a walk in which $v_0 = v_n$ and no other nodes are repeated.

Fact

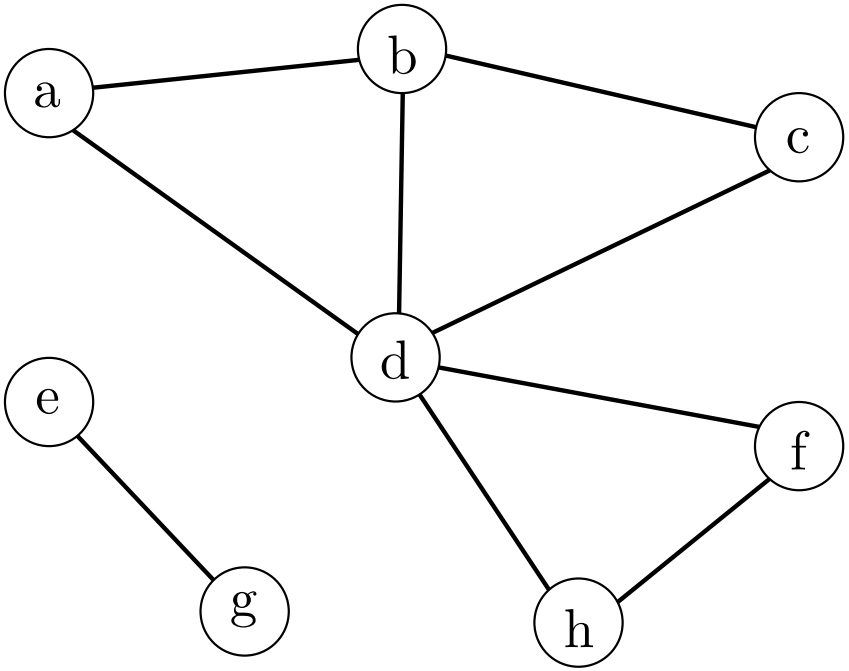
- *By convention, a cycle in a graph is of length at least* 3*.*
- *If there is a walk from u to v, then there is at least one path from u to v.*

Walks, paths and cycles in digraphs



Seq.	walk?	path?	cycle?
0 2 3	no	no	no
3 1 2	yes	yes	no
1 2 6 5 3 1	yes	no	yes
4 5 6 5	yes	no	no
4 3 5	no	no	no

Walks, paths and cycles in graphs



Seq.	walk?	path?	cycle?
a b c	yes	yes	no
e g e	yes	no	no
d b c d	yes	no	yes
d a d f	yes	no	no
a b d f h	yes	yes	no

Distances and diameter of digraphs

Definition

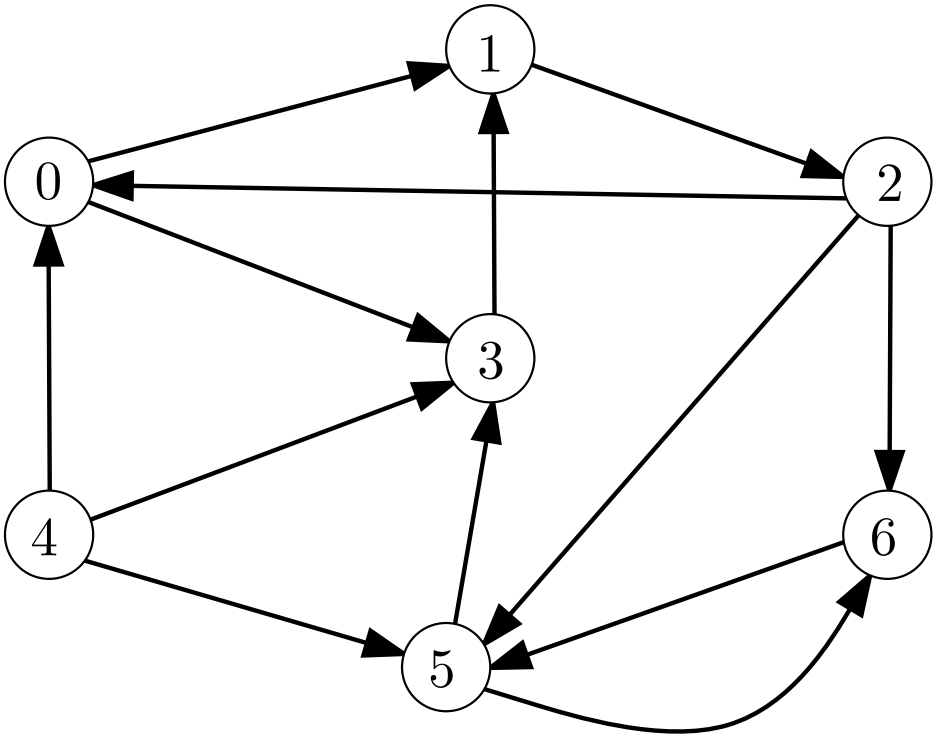
The distance from u to v in G, denoted by $d(u, v)$, is the *minimum* length of a path from u to v. If no path exists, the distance is undefined (or $+\infty$).

For graphs, we have $d(u, v) = d(v, u)$ for all vertices u and v.

Definition

The diameter of a digraph is the *maximum* distance between any two vertices. That is, $\max_{u,v \in V}[d(u, v)]$.

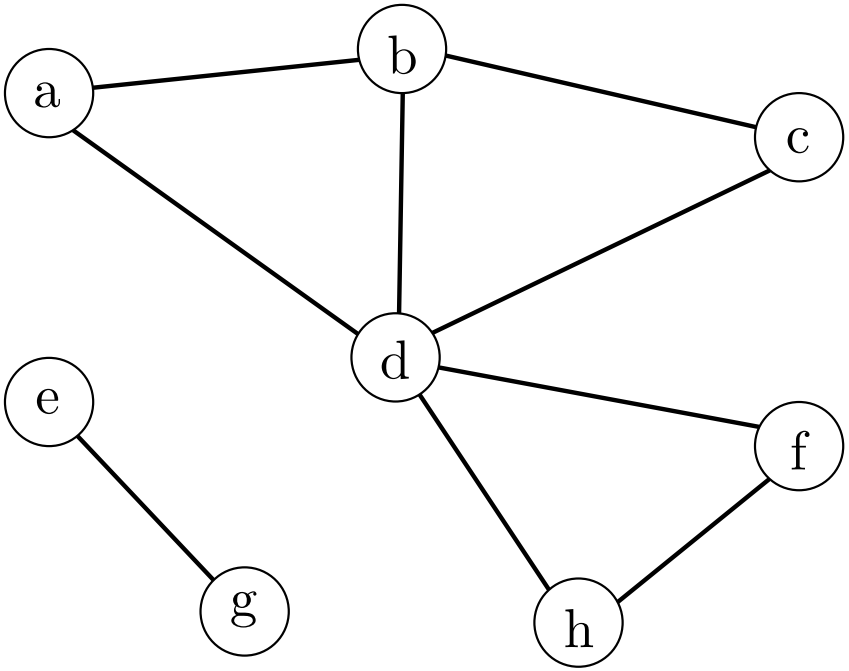
Examples of path distances in digraphs



$d(0,1) = 1$, $d(0,2) = 2$, $d(0,5) = 3$
$d(0,4) = \infty$
$d(5,5) = 0$, $d(5,2) = 3$, $d(5,0) = 4$
$d(4,6) = 2$, $d(4,1) = 2$, $d(4,2) = 3$

diameter is ∞

Examples of path distances in graphs



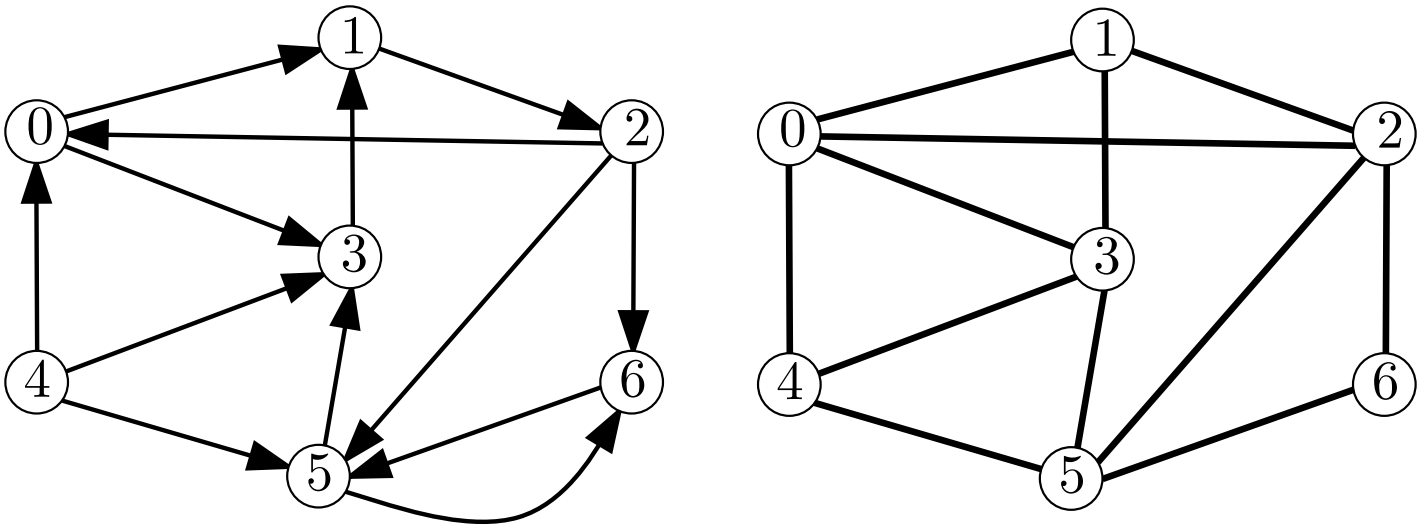
$d(a, b) = 1$, $d(a, c) = 2$, $d(a, f) = 2$
$d(a, e) = \infty$, $d(e, e) = 0$, $d(e, g) = 1$
$d(h, f) = 1$, $d(d, h) = d(h, d) = 1$

diameter is ∞

Underlying graph

Definition

The underlying graph of a digraph $G = (V, E)$ is the graph $G' = (V, E')$ where $E' = \{\{u, v\} \mid (u, v) \in E\}$.



Sub[di]graphs

Definition

A subdigraph of a digraph $G = (V, E)$ is a digraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$.

Definition

A spanning subdigraph is one with $V' = V$; that is, it contains all nodes.

Definition

The subdigraph induced by a subset V' of V is the digraph $G' = (V', E')$ where $E' = \{(u, v) \in E \mid u \in V' \text{ and } v \in V'\}$.

Computer representation of digraphs

When the vertices V are labeled $0, 1, \ldots, n-1$:

Definition

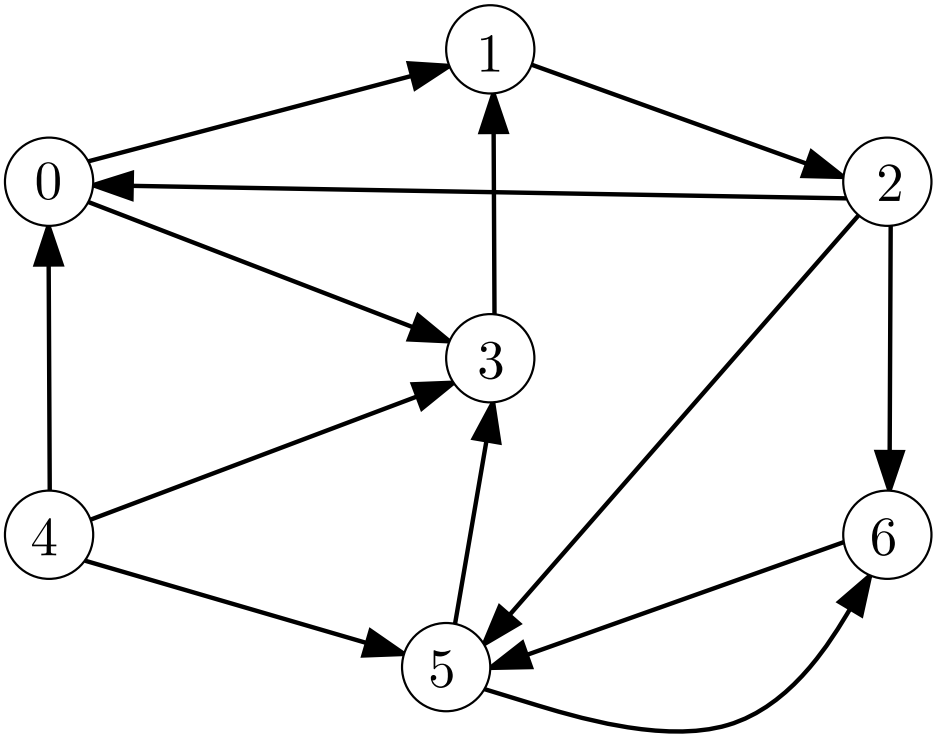
Let G be a digraph of order n. The adjacency matrix of G is the $n \times n$ boolean matrix (often encoded with 0's and 1's) such that entry (i, j) is true if and only if there is an arc from the node i to node j.

Definition

For a digraph G of order n, an adjacency lists representation is a sequence of n sequences, $L_0, \ldots, L_{n-1}$. Sequence L_i contains all nodes of G that are adjacent to node i.

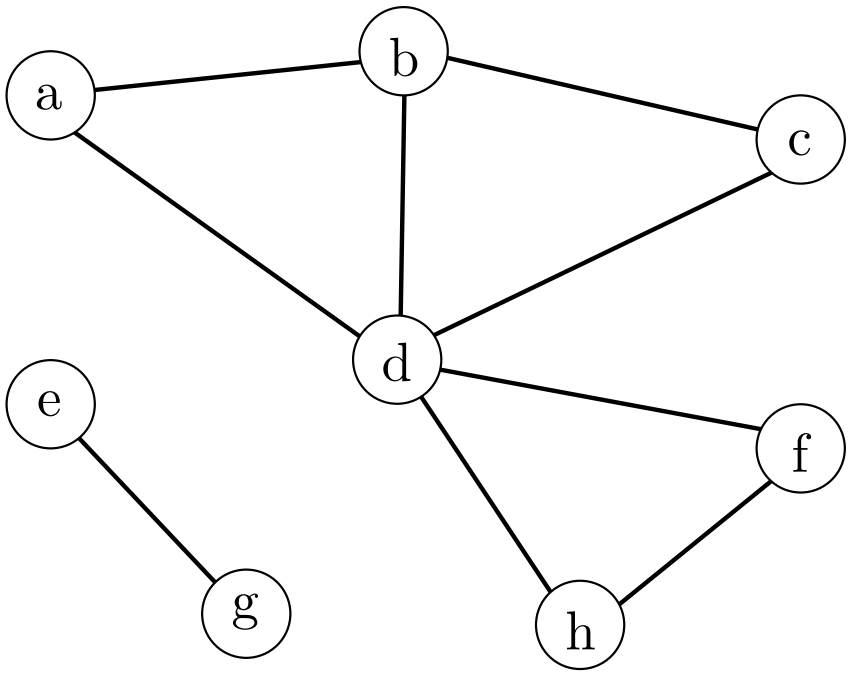
The sequence L_i may not be sorted! But we usually sort them.

Adjacency matrix of a digraph



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

Adjacency lists of a graph



	symbolic	**numeric**
		8
0=	a: b d	1 3
1=	b: a c d	0 2 3
2=	c: b d	1 3
3=	d: a b c f h	0 1 2 5 7
4=	e: g	6
5=	f: d h	3 7
6=	g: e	4
7=	h: d f	3 5

Digraph operations in terms of data structures

Operation	**Adjacency Matrix**	**Adjacency Lists**
arc (i, j) exists?	is entry (i, j) 0 or 1	find j in list i
out-degree of i	scan row, find 1's	size of list i
in-degree of i	scan column, find 1's	for $j \neq i$, find i in list j
add arc (i, j)	change entry (i, j)	insert j in list i
delete arc (i, j)	change entry (i, j)	delete j from list i
add node	create new row/column	add new list at end
delete node i	delete row/column i shuffle other entries	delete list i for $j \neq i$, delete i from list j

Comparative performance of adjacency lists and matrices

Operation	**array/array**	**list/list**[1]
arc (i, j) exists?	$\Theta(1)$	$\Theta(d)$
out-degree of i	$\Theta(n)$	$\Theta(1)$
in-degree of i	$\Theta(n)$	$\Theta(n+e)$
add arc (i, j)	$\Theta(1)$	$\Theta(1)$
delete arc (i, j)	$\Theta(1)$	$\Theta(d)$
add node	$\Theta(n)$	$\Theta(1)$
delete node i	$\Theta(n)$	$\Theta(n+e)$

[1] Here d denotes size of the adjacency list for vertex i.

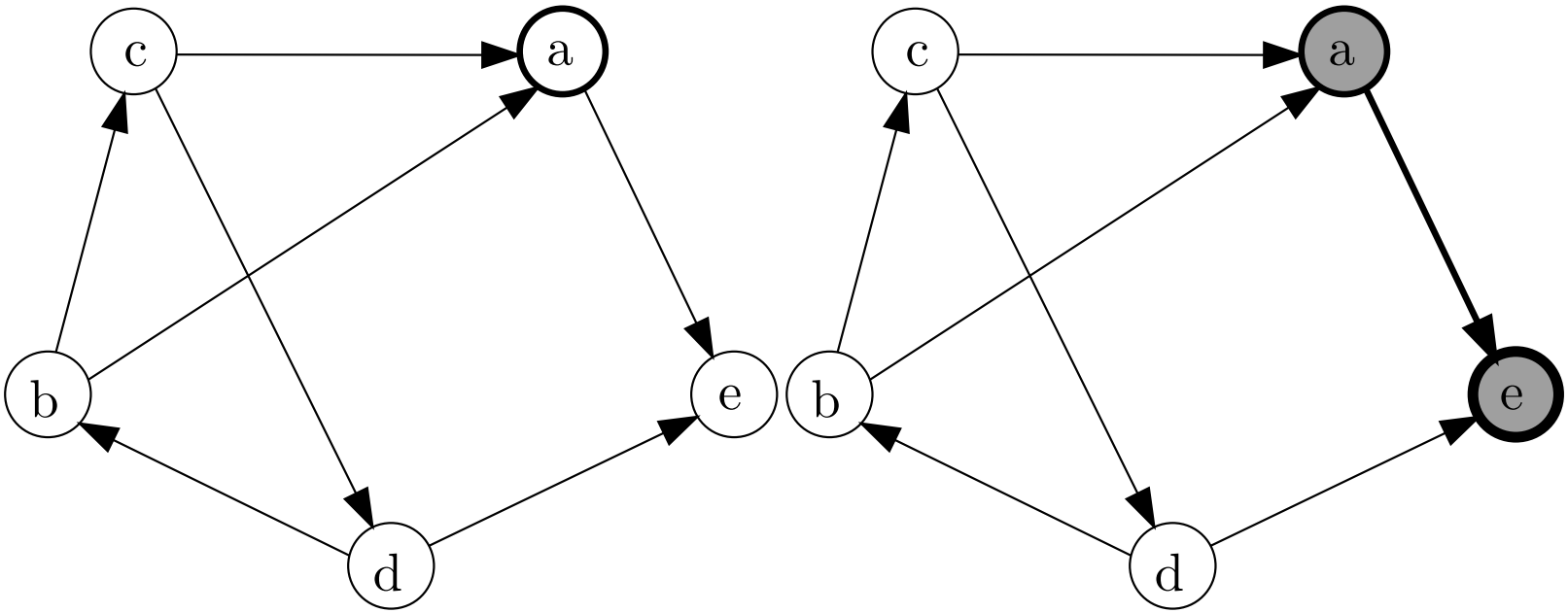
General graph traversal algorithm (part 1)

algorithm traverse
    Input: digraph G
begin
    array $color[n]$, $pred[n]$
    for $u \in V(G)$ **do**
        $color[u] \leftarrow$ WHITE
    end for
    for $s \in V(G)$ **do**
        if $color[s] =$ WHITE **then**
            `visit`(s)
        end if
    end for
    return *pred*
end

General graph traversal algorithm (part 2)

algorithm visit
    Input: node s of digraph G
begin
    $color[s] \leftarrow$ GREY; $pred[s] \leftarrow$ NULL
    while there is a grey node **do**
        choose a grey node u
        if there is a white neighbor of u
            choose such a neighbor v
            $color[v] \leftarrow$ GREY; $pred[v] \leftarrow u$
        else $color[u] \leftarrow$ BLACK
        end if
    end while
end

Illustrating the general traversal algorithm



visit(a)
e is white ne

Traversal arc classifications

Suppose we have performed a traversal of a digraph G, resulting in a search forest F. Let $(u, v) \in E(G)$ be an arc.

Definition

The arc is called a tree arc if it belongs to one of the trees of F. If the arc is not a tree arc, there are three possibilities:

1. a forward arc if u is an ancestor of v in F,
2. a back arc if u is a descendant of v in F, and
3. a cross arc if neither u nor v is an ancestor of the other in F.

Facts about traversal trees

Theorem

Suppose we run algorithm traverse on G, resulting in a search forest F. Let $v, w \in V(G)$.

1. *Let T_1 and T_2 be different trees in F and suppose that T_1 was explored before T_2. Then there are no arcs from T_1 to T_2.*
2. *Suppose that G is a graph. Then there can be no edges joining different trees of F.*
3. *Suppose that v is visited before w and w is reachable from v in G. Then v and w belong to the same tree of F.*
4. *Suppose that v and w belong to the same tree T in F. Then any path from v to w in G must have all nodes in T.*

Run-time analysis of traverse

In the while-loop of subroutine visit let:

- a and A be lower and upper bounds on the time to choose a grey node.
- b and B be lower and upper bounds on the time to choose a white node.

The running time[2] of traverse is:

- $O(An + Be)$ and $\Omega(an + be)$ if adjacency lists are used, and
- $O(An + Bn^2)$ and $\Omega(an + bn^2)$ if adjacency matrices are used

[2] Recall n is the order and e is the size of G.

Depth-first-search (DFS) algorithm (part 1)

algorithm dfs
    Input: digraph G
begin
    stack S; array $color[n], pred[n], seen[n], done[n]$
    for $u \in V(G)$ **do**
        $color[u] \leftarrow$ WHITE; $pred[u] \leftarrow$ NULL
    end for
    $time \leftarrow 0$
    for $s \in V(G)$ **do**
        if $color[s] =$ WHITE **then**
            dfsvisit(s)
        end if
    end for
    return $pred, seen, done$
end

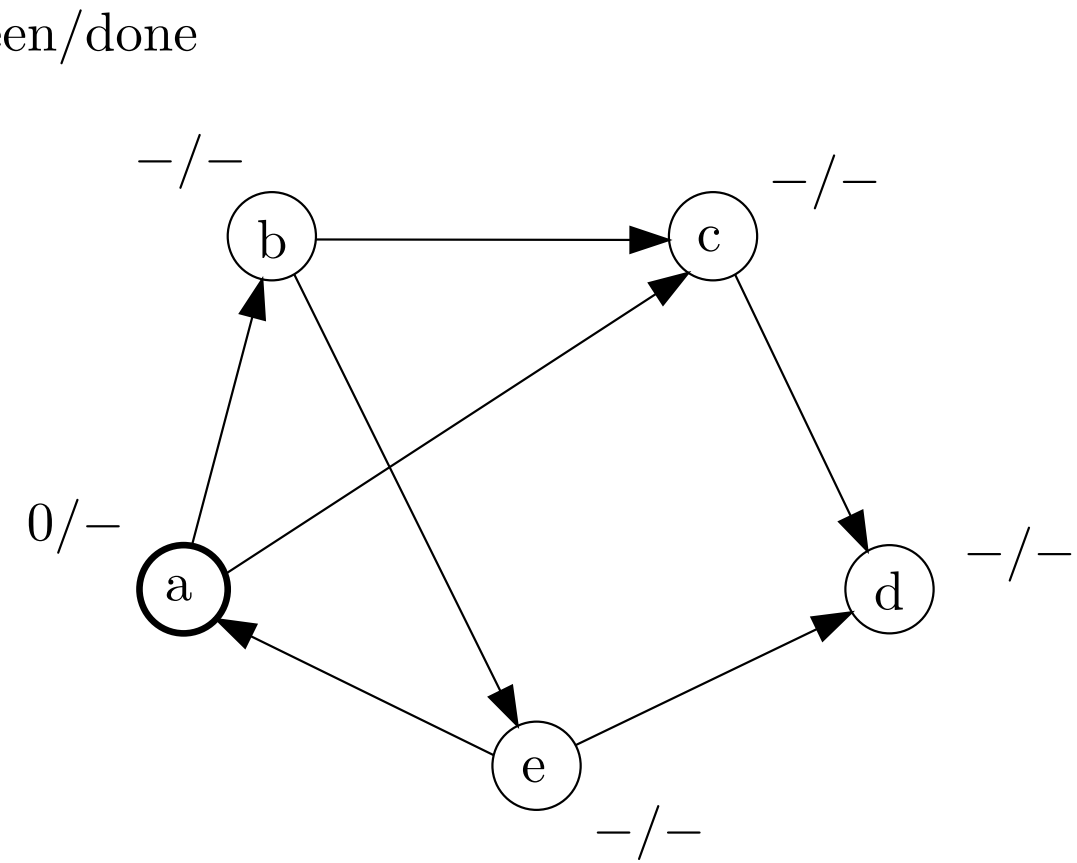
Depth-first-search (DFS) algorithm (part 2)

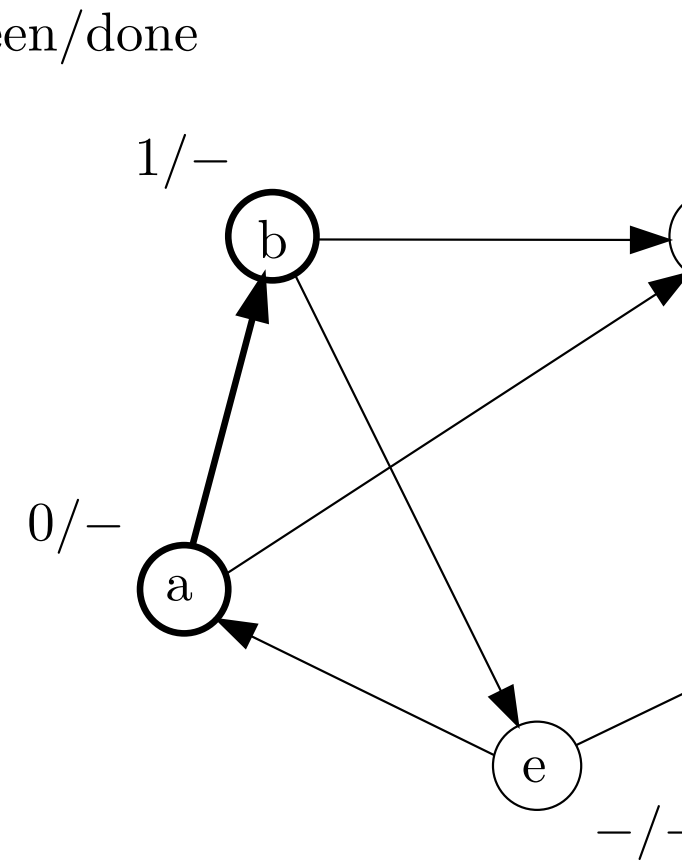
```
algorithm dfsvisit
    Input: node s
begin
    color[s] ← GREY; seen[u] ← time++; S.push_top(s)
    while not S.isempty() do
        u ← S.get_top()
        if there is a v adjacent to u and color[v] = WHITE then
            color[v] ← GREY; pred[v] ← u
            seen[v] ← time++; S.push_top(v)
        else
            S.del_top(); color[u] ← BLACK; done[u] ← time++;
        end if
    end while
end
```

Recursive view of DFS algorithm

```
algorithm rec_dfs_visit
    Input: node s
begin
    color[s] ← GREY
    seen[s] ← time++
    for each v adjacent to s do
        if color[v] = WHITE then
            pred[v] ← s
            rec_dfs_visit(v)
        end if
    end for
    color[s] ← BLACK
    done[s] ← time++
end
```

A DFS example





Basic properties of depth-first search

- Each call to dfs_visit(v) terminates only when all nodes reachable from v via a path of white nodes have been seen.
- Suppose that (v, w) is an arc. Cases:
  - tree or forward arc:
    $seen[v] < seen[w] < done[w] < done[v]$;
  - back arc: $seen[w] < seen[v] < done[v] < done[w]$;
  - cross arc: $seen[w] < done[w] < seen[v] < done[v]$.

  Hence on a graph, there are no cross edges.

Using DFS to determine ancestors of a tree

Theorem

Suppose that we have performed DFS on a digraph G, resulting in a search forest F. Let $v, w \in V(G)$ and suppose that $seen[v] < seen[w]$.

- *If v is an ancestor of w in F, then*

$$seen[v] < seen[w] < done[w] < done[v] \quad .$$

- *If v is not an ancestor of w in F, then*

$$seen[v] < done[v] < seen[w] < done[w] \quad .$$

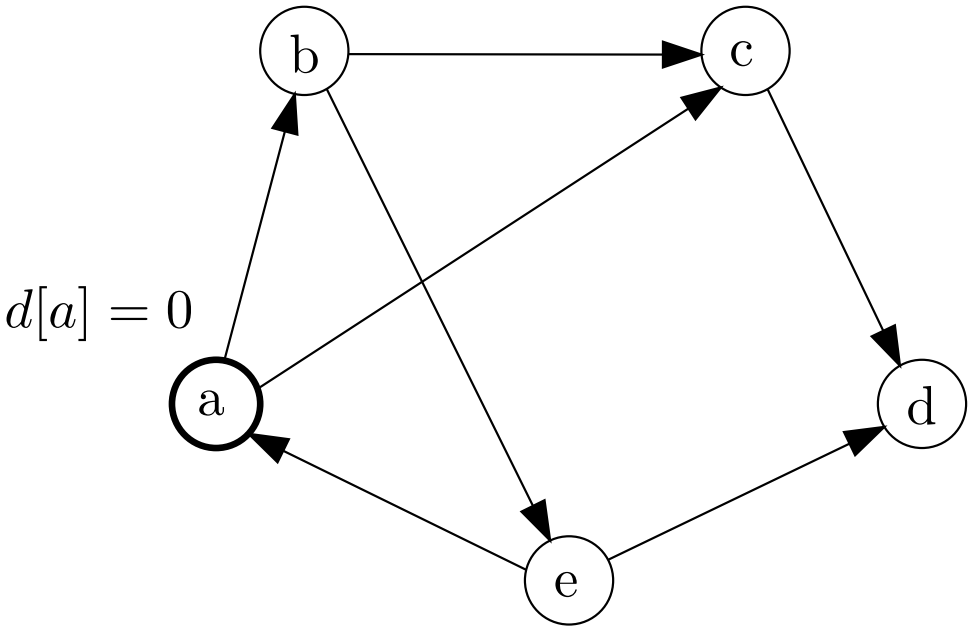
Breadth-first-search (BFS) algorithm (part 1)

algorithm bfs
    Input: digraph G
begin
    queue Q
    array $color[n], pred[n], d[n]$
    for $u \in V(G)$ **do**
        $color[u] \leftarrow$ WHITE; $pred[u] \leftarrow$ NULL
    end for
    for $s \in V(G)$ **do**
        if $color[s] =$ WHITE **then**
            bfsvisit(s)
        end if
    end for
    return $pred, d$
end

Breadth-first-search (BFS) algorithm (part 2)

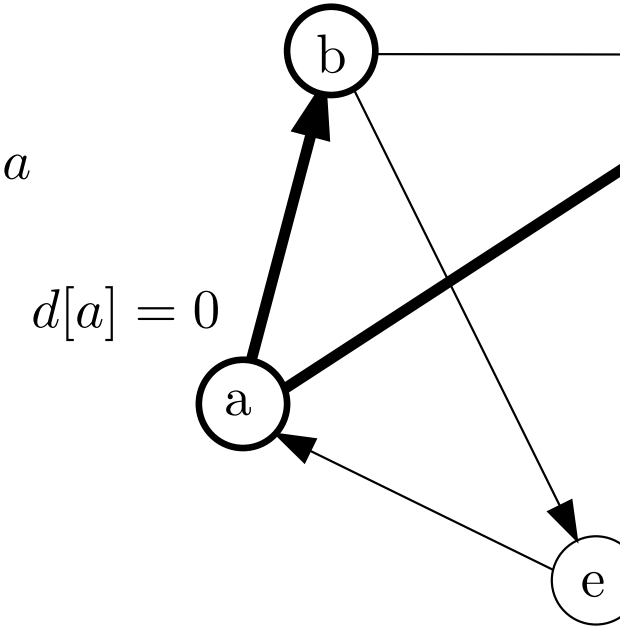
```
algorithm bfsvisit
    Input: node s
begin
    color[s] ← GREY; d[s] ← 0; Q.enqueue(s)
    while not Q.isempty() do
        u ← Q.get_head()
        for each v adjacent to u do
            if color[v] = WHITE then
                color[v] ← GREY; pred[v] ← u; d[v] ← d[u] + 1
                Q.enqueue(v)
            end if
        end for
        Q.dequeue(); color[u] ← BLACK
    end while
end
```

A BFS example



Queue: a

$d[b] = 1$



Priority-first-search (PFS) algorithm (part 1)

algorithm pfs
    Input: digraph G
begin
    priority queue Q;
    array $color[n], pred[n]$
    for $u \in V(G)$ **do**
        $color[u] \leftarrow$ WHITE; $pred[u] \leftarrow$ NULL
    end for
    for $s \in V(G)$ **do**
        if $color[s] =$ WHITE **then**
            pfsvisit(s)
        end if
    end for
    return *pred*
end

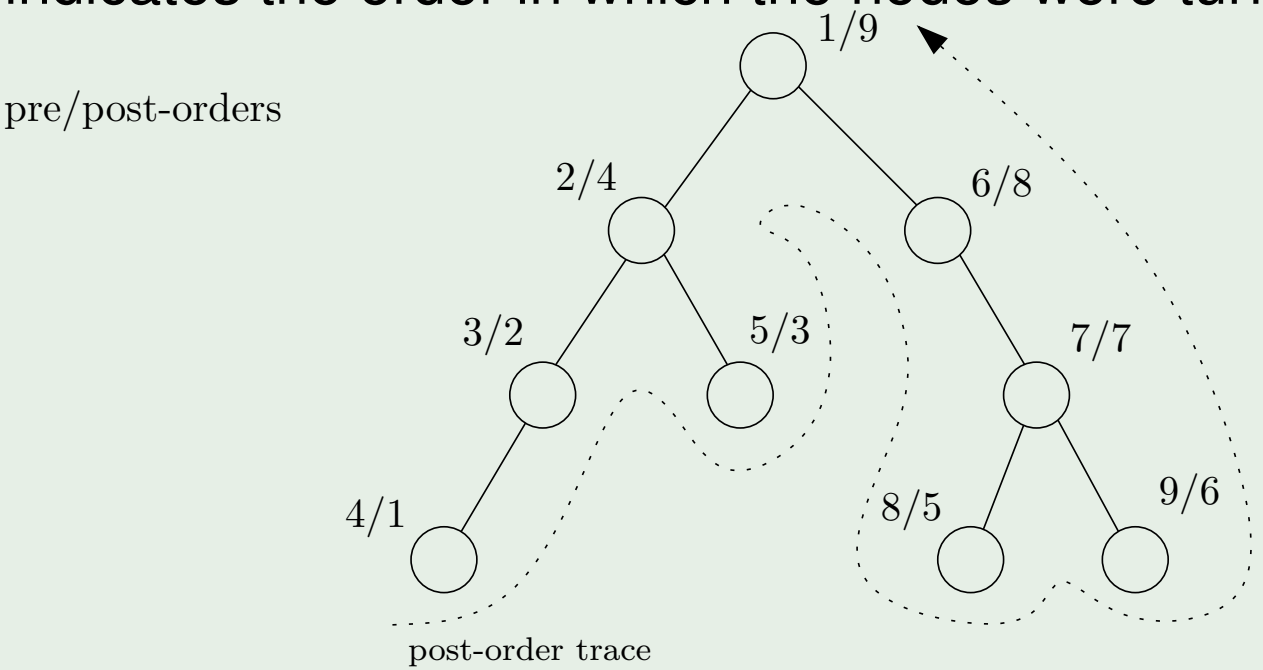
Priority-first-search (PFS) algorithm (part 2)

```
algorithm pfsvisit
    Input: node s
begin
    color[s] ← GREY; Q.insert(s, setkey (s))
    while not Q.isempty() do
        u ← Q.get_min()
        if v adjacent to u and color[v] = WHITE then
            color[v] ← GREY;
            Q.insert(v, setkey (v))
        end if
        Q.del_min(); color[u] ← BLACK
    end while
end
```

Pre-order and post-order labelings

Example

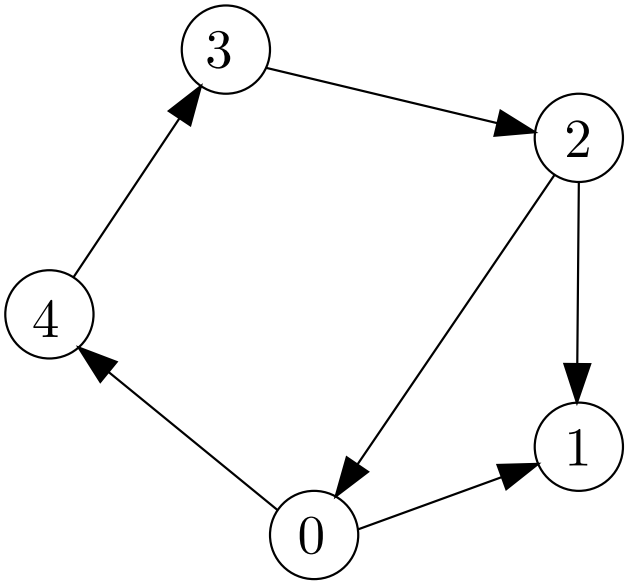
DFS allows us to give a so-called pre-order and post-order labeling to a digraph. The pre-order label indicates the order in which the nodes were turned grey. The post-order label indicates the order in which the nodes were turned black.

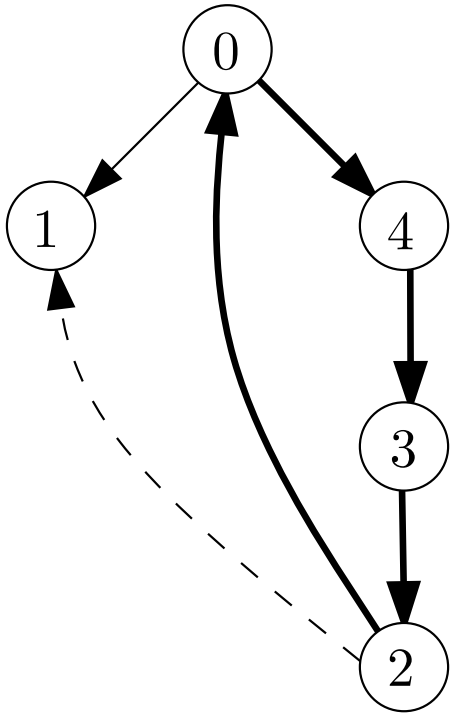


Cycle detection

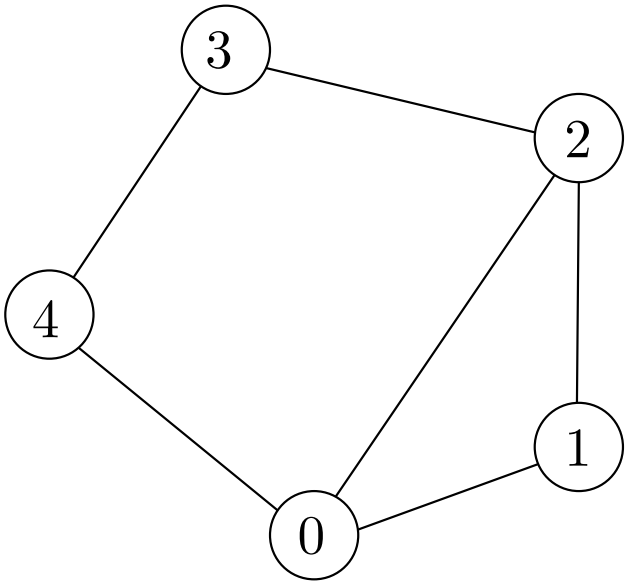
- Suppose that there is a cycle in G and let v be the node in the cycle visited first by DFS. If (u, v) is an arc in the cycle then it must be a back arc.
- Conversely if there is a back arc, we must have a cycle. So a digraph is acyclic iff there are no back arcs from DFS.
- An acyclic digraph is called a directed acyclic graph (DAG). An acyclic graph is a forest.
- Cycles can also be easily detected in a graph using BFS. Finding a cycle of minimum length in a graph is not difficult using BFS (better than DFS).

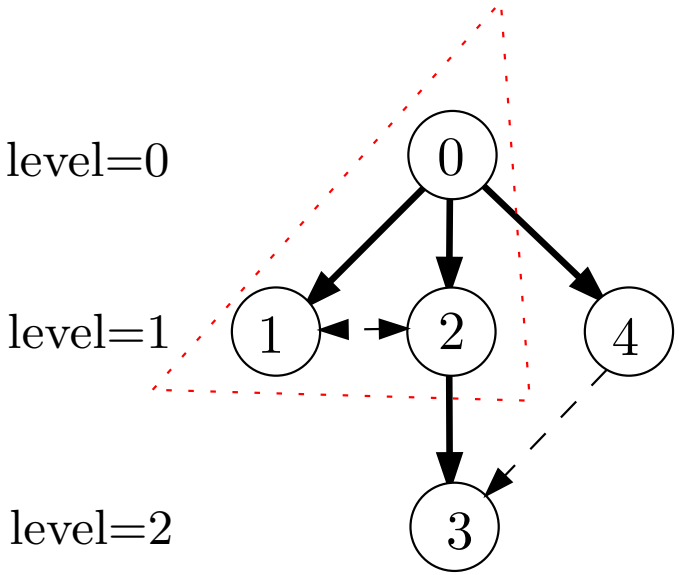
Using DFS to find cycles in digraphs





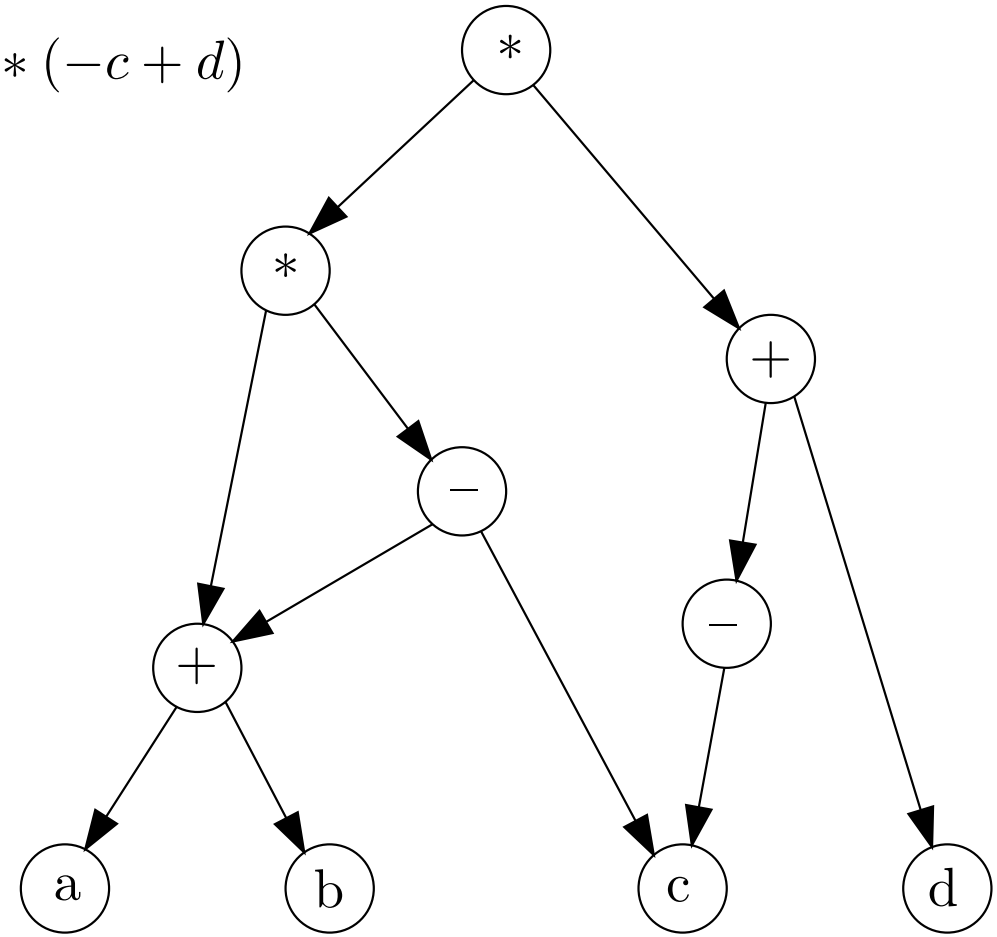
Using BFS to find cycles in graphs





Arithmetic expression digraphs

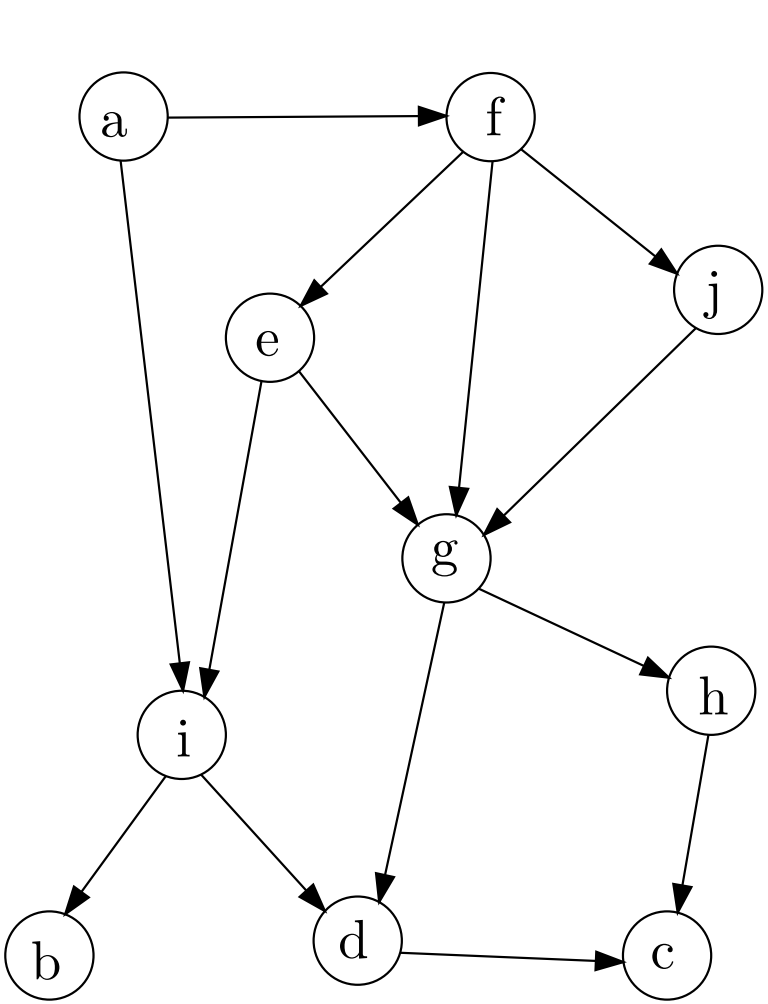
$$(a + b) * (c - (a + b)) * (-c + d)$$

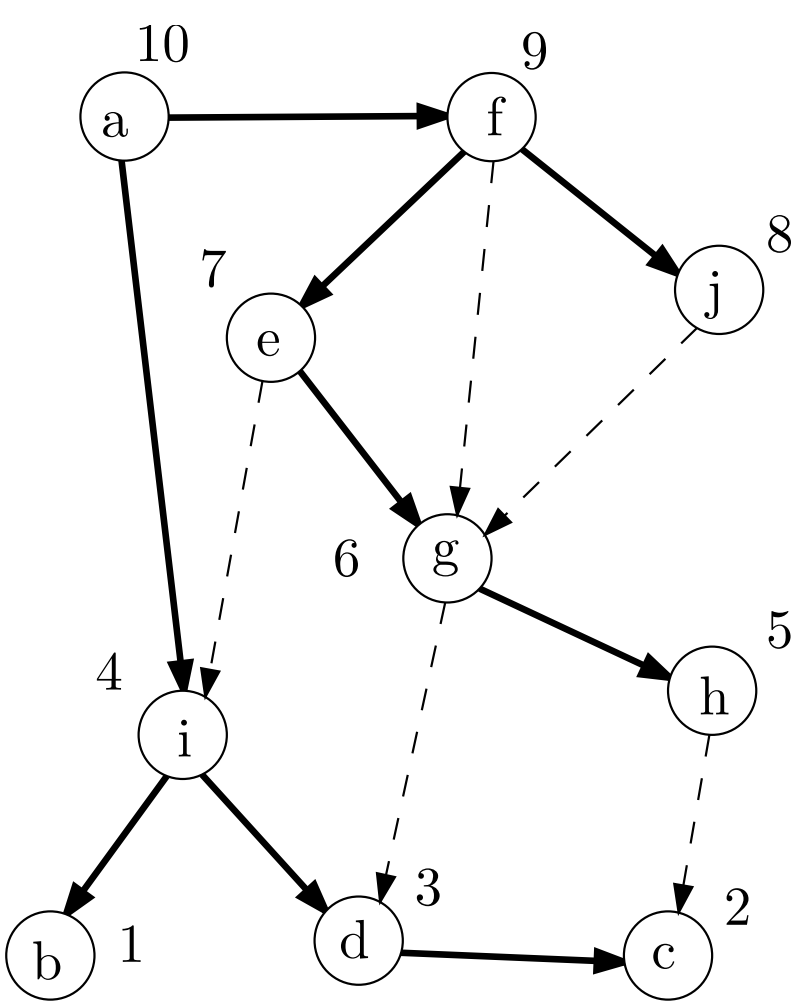


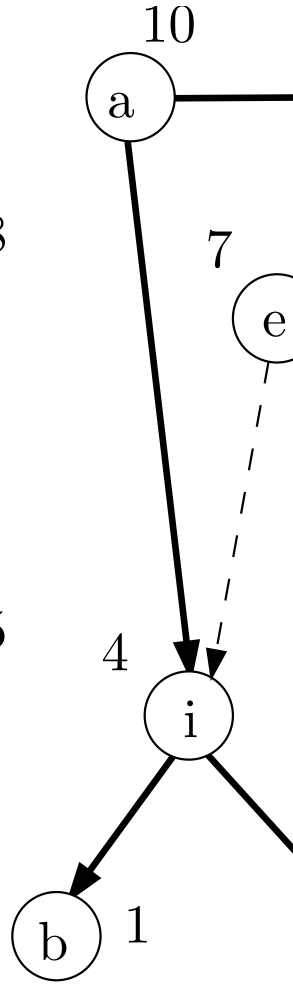
Topological sorting

- To place nodes of a digraph on a line so all arcs go in one direction. Possible if and only if digraph is a DAG.
- Main application: scheduling events (arithmetic expressions, university prerequisites, etc).
- List of finishing times for depth-first search, in reverse order, solves the problem (since there are no back arcs, each node finishes before anything pointing to it).
- Another solution: zero in-degree sorting. Find node of in-degree zero, delete it and repeat until all nodes listed. Less efficient(?)

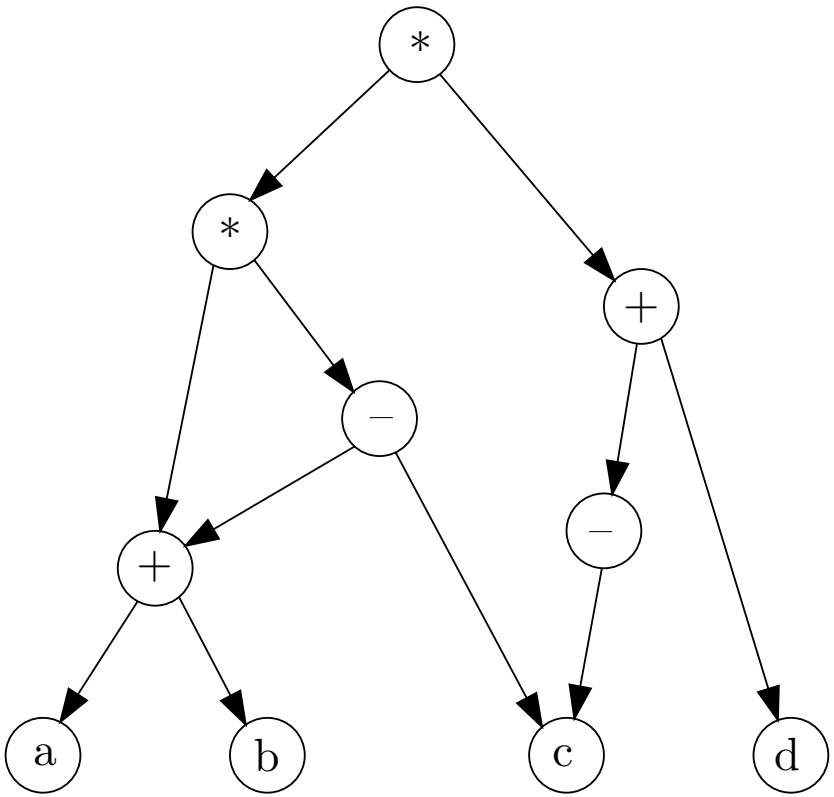
Topological sorting via DFS

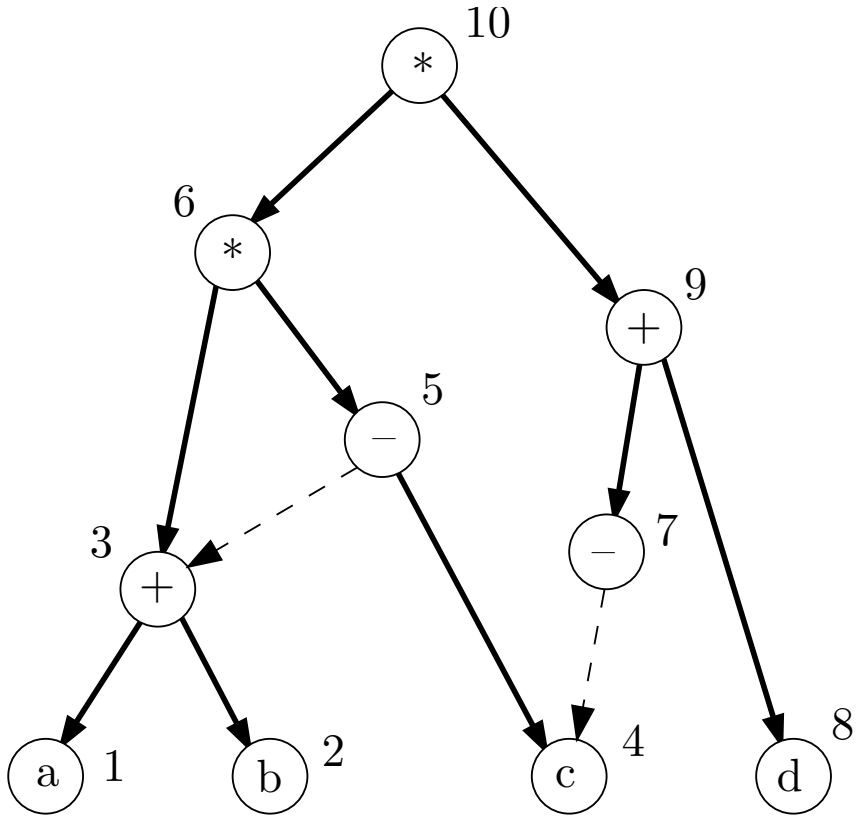




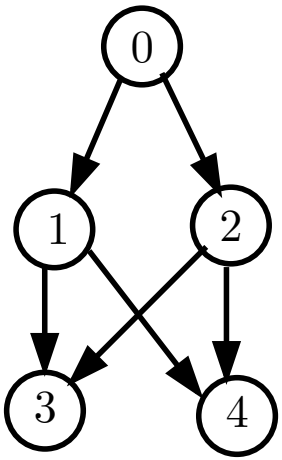


Arithmetic expression evaluation order





Many topological orders per DAG

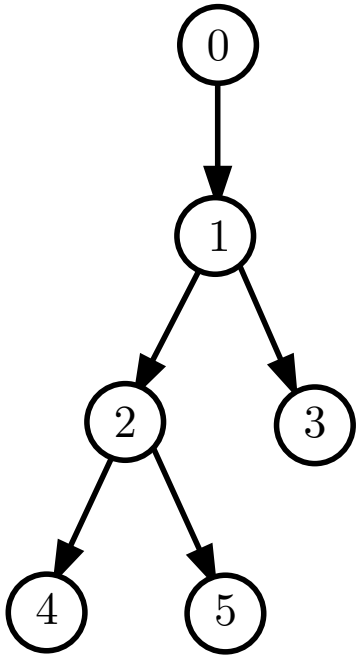


0,1,2,3,4

0,1,2,4,3

0,2,1,3,4

0,2,1,4,3



0,1,3,2,4,5

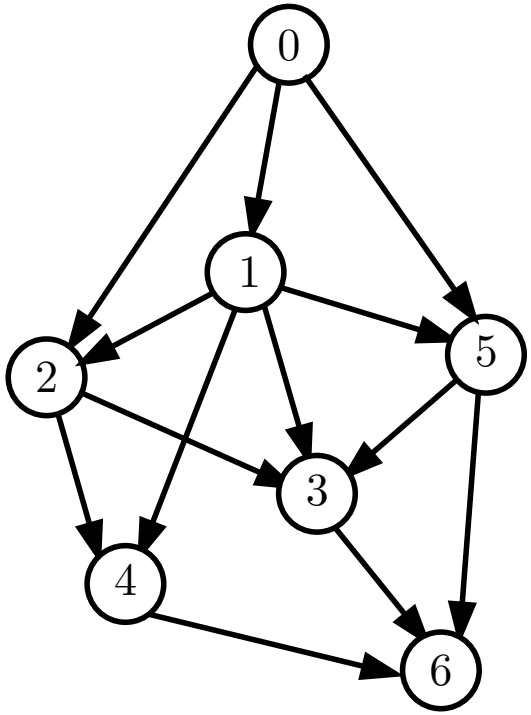
0,1,3,2,5,4

0,1,2,3,4,5

0,1,2,3,5,4

⋮

0,1,2,5,4,3



0,1,2,4,5,3,6

0,1,2,5,3,4,6

0,1,2,5,4,3,6

0,1,5,2,3,4,6

0,1,5,2,4,3,6

Graph connectivity

Definition

A graph G is connected if for each pair of vertices $u, v \in V(G)$, there is a path between them.

Definition

A graph G is disconnected if it is not connected and the maximum induced connected subgraphs are call the components of G.

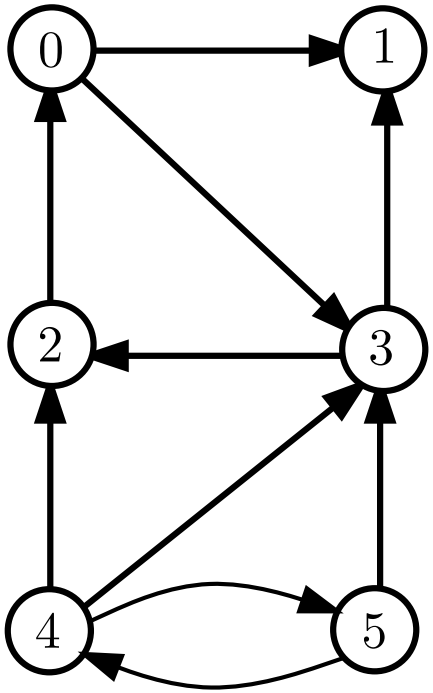
Theorem

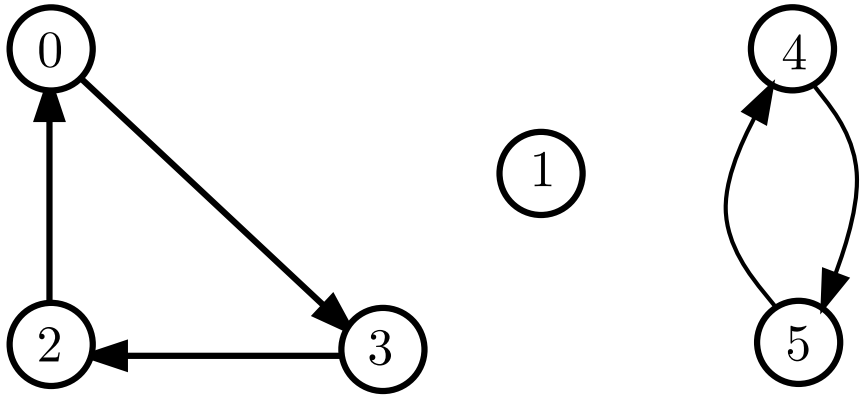
Let G be a graph and suppose that DFS or BFS is run on G. Then the connected components of G are precisely the subgraphs spanned by the trees in the search forest.

Nice DFS application: strong components

- Nodes v and w are mutually reachable if there is a path from v to w and a path from w to v. The nodes of a digraph divide up into disjoint subsets of mutually reachable nodes, which induce strong components.
- (Strong) components are precisely the equivalence classes under the mutual reachability relation.
- A digraph is strongly connected if it has only one strong component.
- Components of a graph are found easily by BFS or DFS. However, this doesn't work well for digraphs (a digraph may have a connected underlying graph yet not be strongly connected). A new idea is needed.

A digraph's strongly connected components



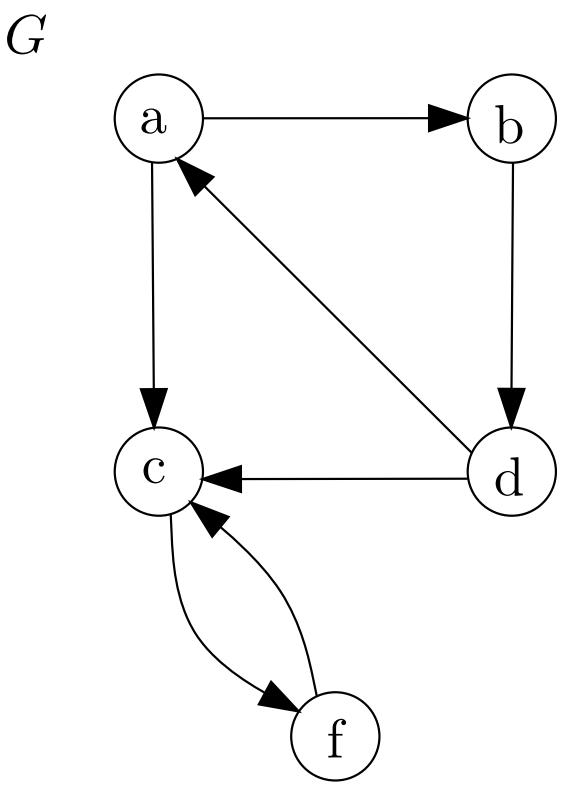


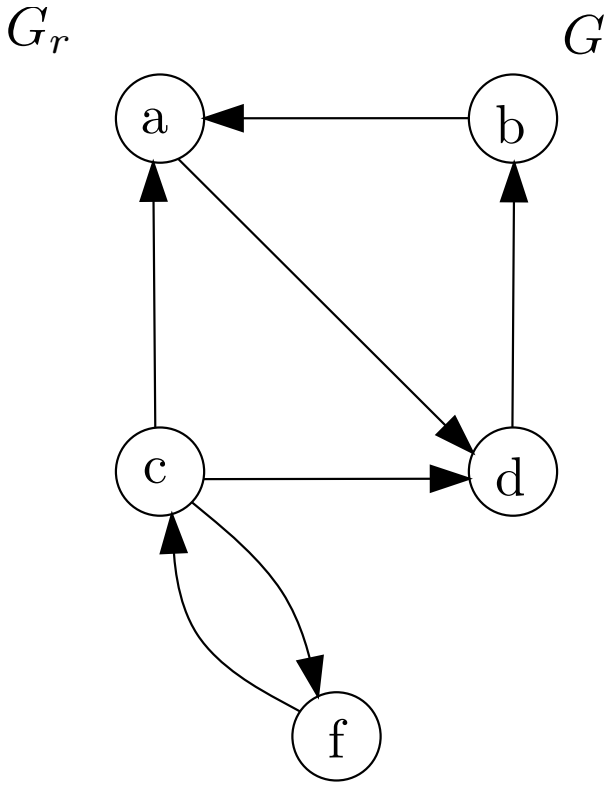
Strong components algorithm

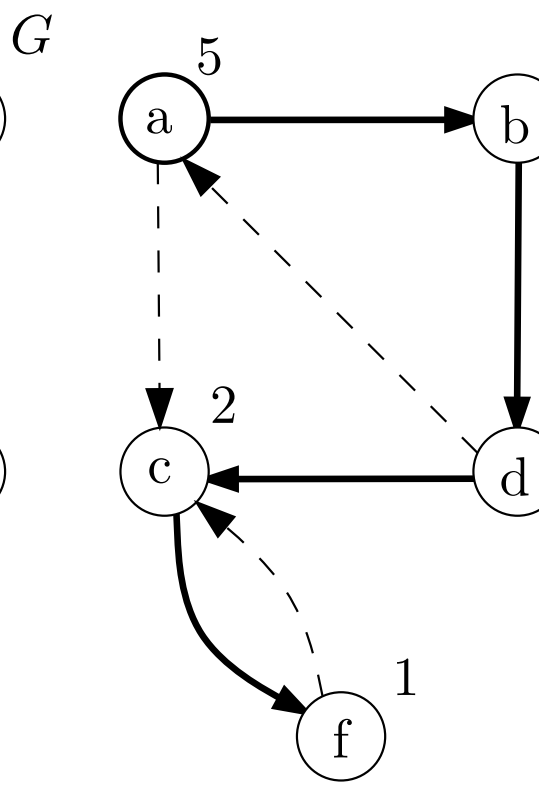
- Run DFS on G, to get depth-first forest F. Create reverse digraph G_r by reversing all arcs. Run DFS on G_r; choose root from unseen nodes finishing latest in F. This gives a forest F_r.
- Suppose v in tree of F_r with root w. Consider the four possibilities in F:
  - $seen[w] < seen[v] < done[v] < done[w]$
  - $seen[w] < done[w] < seen[v] < done[v]$
  - $seen[v] < seen[w] < done[w] < done[v]$
  - $seen[v] < done[v] < seen[w] < done[w]$

  By root choice, 2nd and 3rd impossible. By root choice and since w reachable from v in G, 4th impossible. So v is descendant of w in F, and v, w are in the same strong component. The converse is easy.

Strong connected components example







Girth of a graph (digraph)

Definition

For a graph (with a cycle), the length of the shortest cycle is called the girth of the graph. If the graph has no cycles then the girth is undefined but may be viewed as $+\infty$.

Fact

For a digraph we use the term girth for its underlying graph and the (maybe non-standard) term directed girth for the length of the smallest directed cycle.

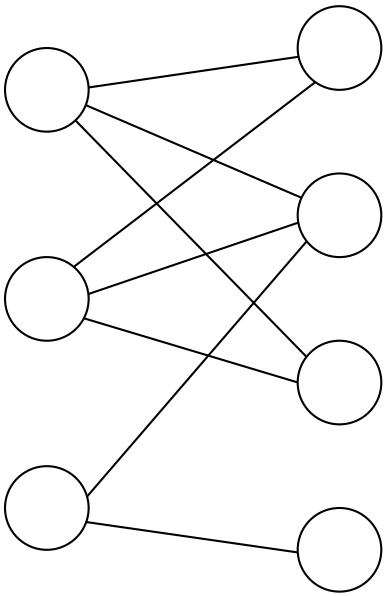
Computing girth of a graph

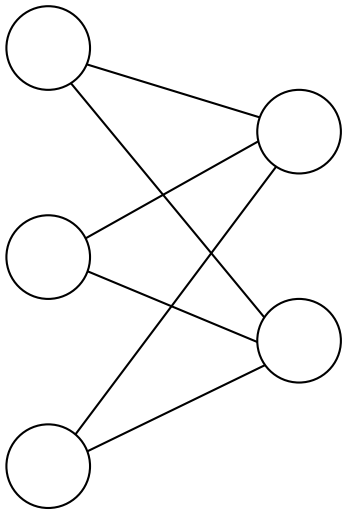
- If vertex v is on at least one cycle then BFS starting at v will find it.
- On detection of a cross arc between descendants u and w, determine whether u and w are in different subtrees.
  - If yes, then a cycle of length $d[u] + d[w] + 1$ is found.
  - If no, then a cycle of shorter length is found (but avoids v).
- If $d[u] = d[w]$ then odd length, where v common ancestor.
- Otherwise, WLOG, $d[u] + 1 = d[w]$ and even length.
- To compute girth, we run the above procedure once for each $v \in V(G)$ and take minimum.

Bipartite graphs (digraphs)

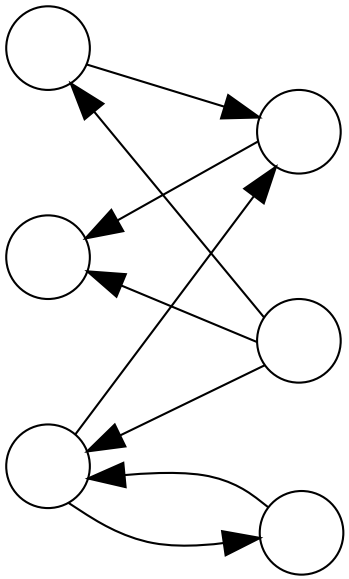
Definition

A graph G is bipartite if $V(G)$ can be partitioned into two nonempty disjoint subsets V_0, V_1 such that each edge of G has one endpoint in V_0 and one in V_1. [Similar for digraphs.]





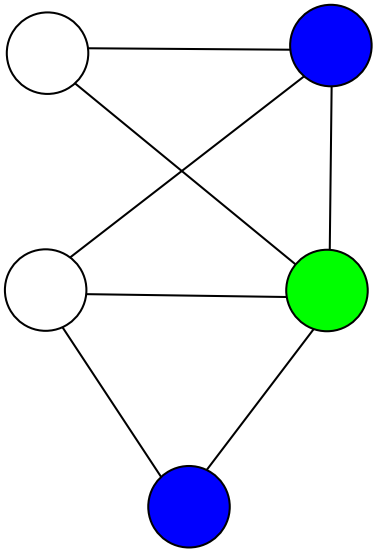
$K_{3,2}$

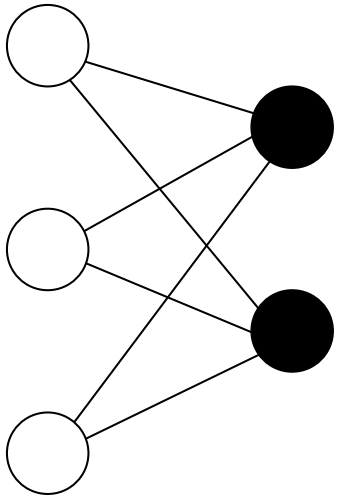


k-colorable graphs

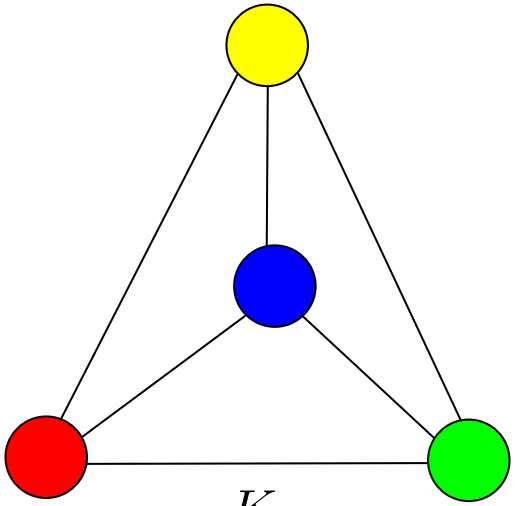
Definition

Let k be a positive integer. A graph G has a k-coloring if $V(G)$ can be partitioned into k nonempty disjoint subsets such that each edge of G joins two vertices in different subsets (colors).





$K_{3,2}$



K_4

Even cycles lengths

Theorem

The following conditions on a graph G are equivalent.

- *G is bipartite;*
- *G has a 2-coloring;*
- *G does not contain an odd length cycle.*

Fact

A version of BFS can be used to check if a graph is bipartite (e.g. 2-colorable).

Weighted (di)graphs

- Very common in applications, also called “networks". Optimization problems on networks are important in operations research.
- Each arc carries a real number “weight", usually positive, can be $+\infty$. Weight typically represents cost, distance, time.
- Representation: weighted adjacency matrix or double adjacency list.
- Standard problems concern finding a minimum or maximum weight path between given nodes (covered here), spanning tree (here and CS 225), cycle or tour (e.g TSP), matching, flow, etc.

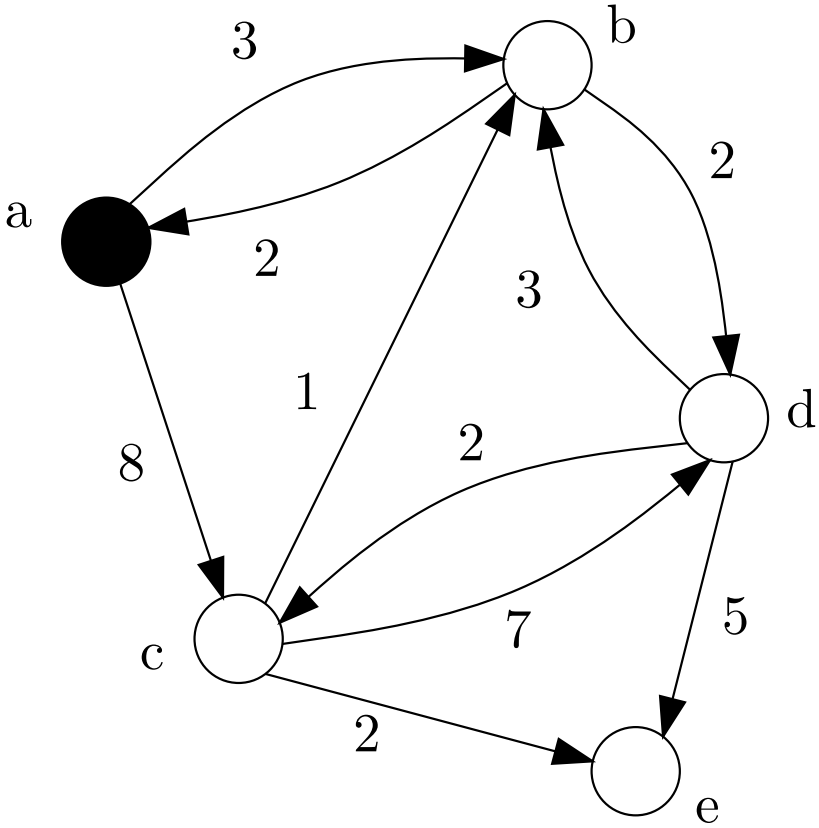
Single-source shortest path problem

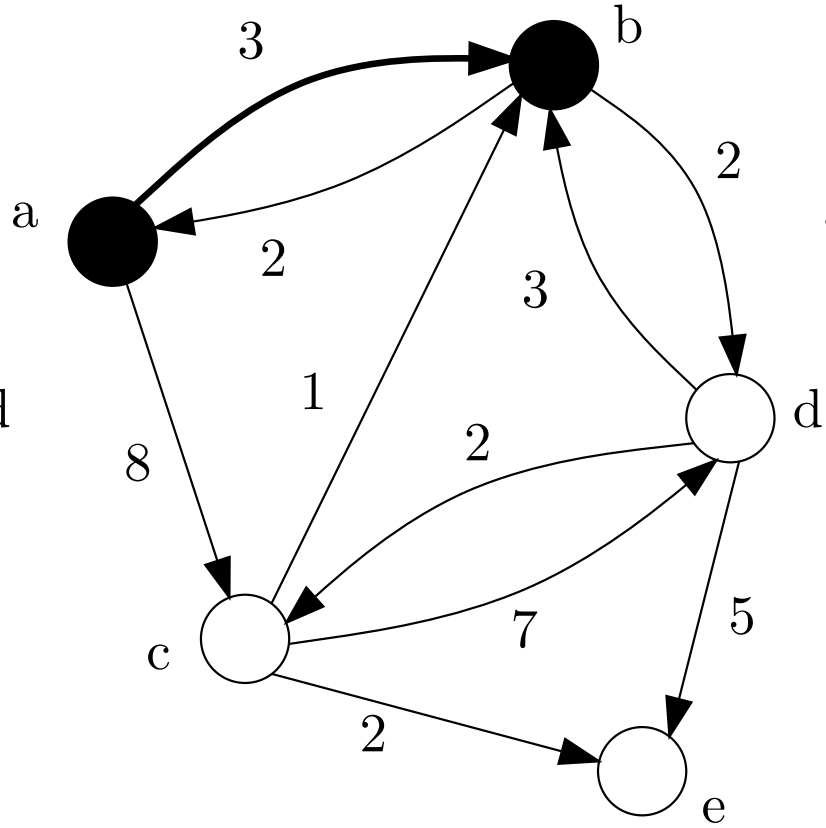
- Given an originating node v, find shortest (minimum weight) path to each other node. If all weights are equal then BFS works, otherwise not.
- Several algorithms are known; we present one, Dijkstra's algorithm . An example of a greedy algorithm; locally best choice is globally best. Doesn't work if weights can be negative.
- Maintain list S of visited nodes (say using a priority queue). Choose closest unvisited node u that is on a path with internal nodes in S. Update distances (of remaining unvisted nodes) from source in case adding u has established shorter paths. Repeat.
- Complexity depends on data structures used, especially for priority queue; $O(e + n \log n)$ is possible.

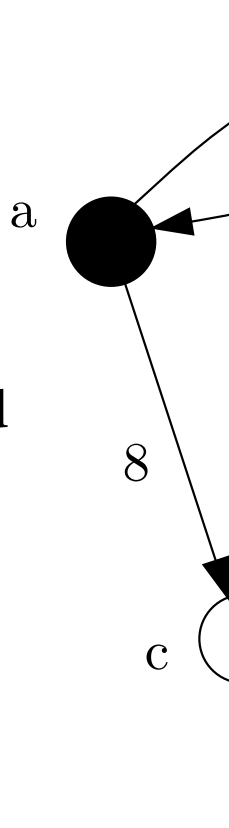
Dijkstra's algorithm

algorithm Dijkstra(weighted digraph (G, c), node $s \in V(G)$)
    array $color[n] = \{\text{WHITE}, \ldots\}$; $color[s] \leftarrow$ BLACK
    array $dist[n] = \{c[s, 0], \ldots, c[s, n-1]\}$
    while there is a white node adjacent to a black node **do**
        pick a white node u so that $dist[u]$ is minimum
        $color[u] \leftarrow$ BLACK
        for x neighbor of u **do**
            if $color[x] =$ WHITE **then**
                $dist[x] \leftarrow \min\{dist[x], dist[u] + c[u, x]\}$
            end if
        end for
    end while
    return *dist*
end

Illustrating Dijkstra's algorithm







Why Dijkstra's algorithm works

Let a *S*-path be a path starting at node s and ending at node w with all nodes colored black except possibly w.

Fact

At the top of `while` *loop, these properties hold for all* $w \in V(G)$*:*

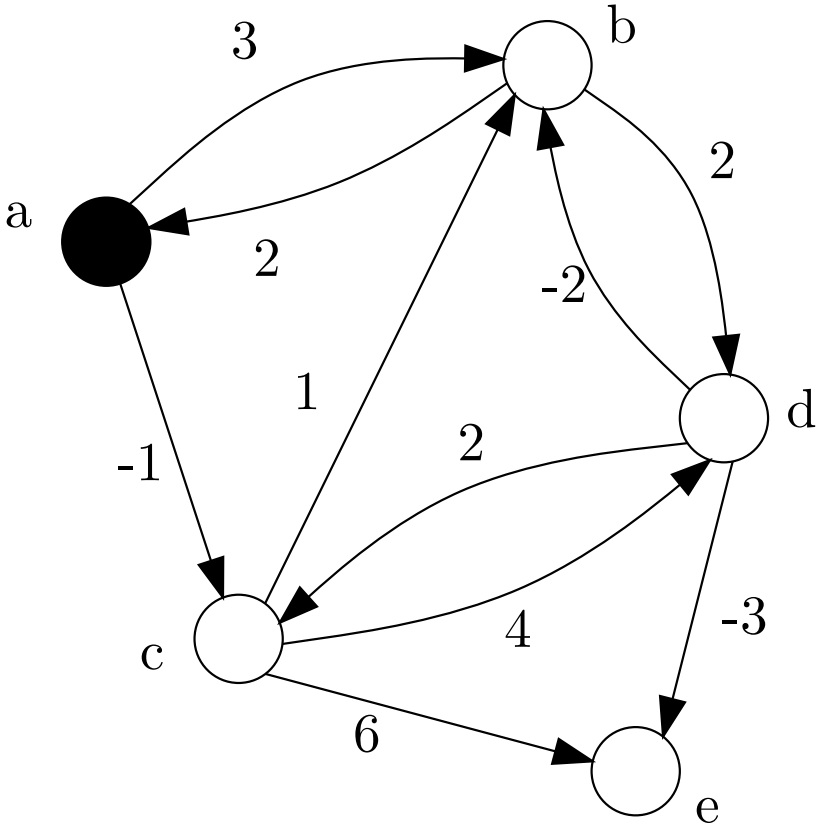
*P*1: *dist*$[w]$ *is the minimum weight of an S-path to* w*.*

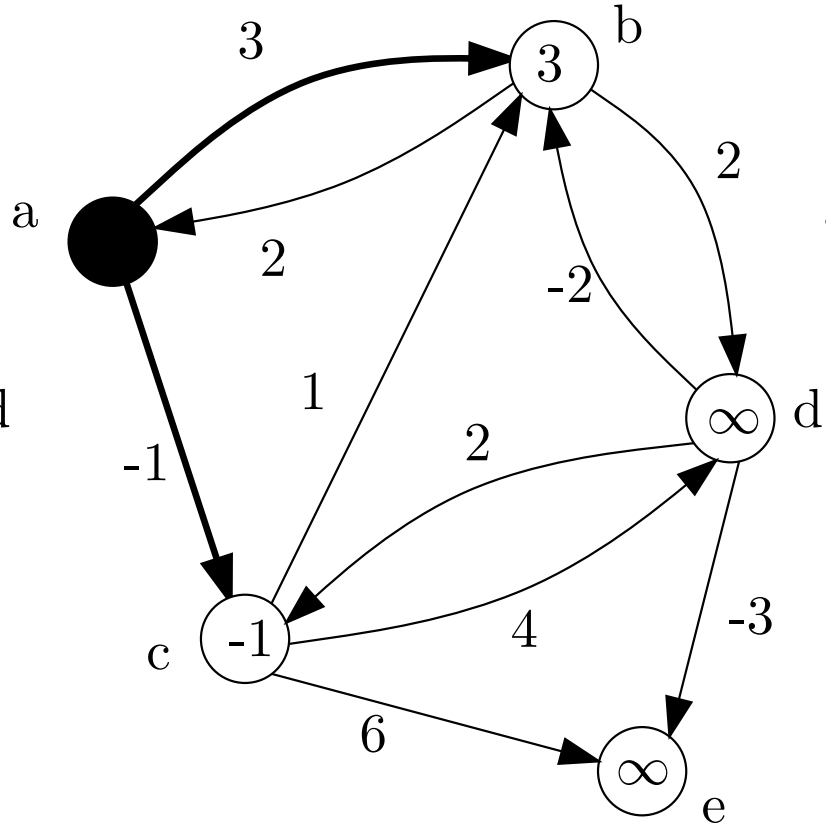
*P*2: *if color*$[w] =$ *BLACK, dist*$[w]$ *is the minimum weight.*

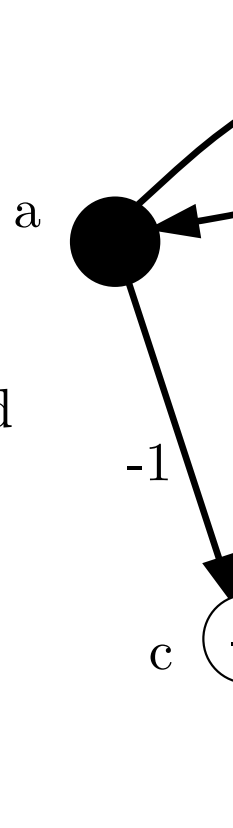
Bellman-Ford algorithm

algorithm Bellman-Ford(weighted digraph (G, c); node s)
    array $dist[n] = \{\infty, \ldots\}$
    $dist[s] \leftarrow 0$
    for i **from** 0 **to** $n - 1$ **do**
        for $x \in V(G)$ **do**
            for $v \in V(G)$ **do**
                $dist[v] \leftarrow \min(dist[v], dist[x] + c(x, v))$
            end for
        end for
    end for
    return *dist*
end

Illustrating Bellman-Ford's algorithm







Comments on Bellman-Ford algorithm

Fact

- *This (non-greedy) algorithm handles negative weight arcs but not negative weight cycles.*
- *Runs slower than Dijkstra's algoirthm since considers all nodes at "level" 0, 1, . . . , $n-1$, in turn.*
- *Runs in time $O(ne)$ since the two inner-most* ***for*** *loops can be replaced with:* ***for*** $(u, v) \in E(V)$.
- *Can be modified to detect negative weight cycle. (see Exercise 6.3.4)*

All pairs shortest path problem

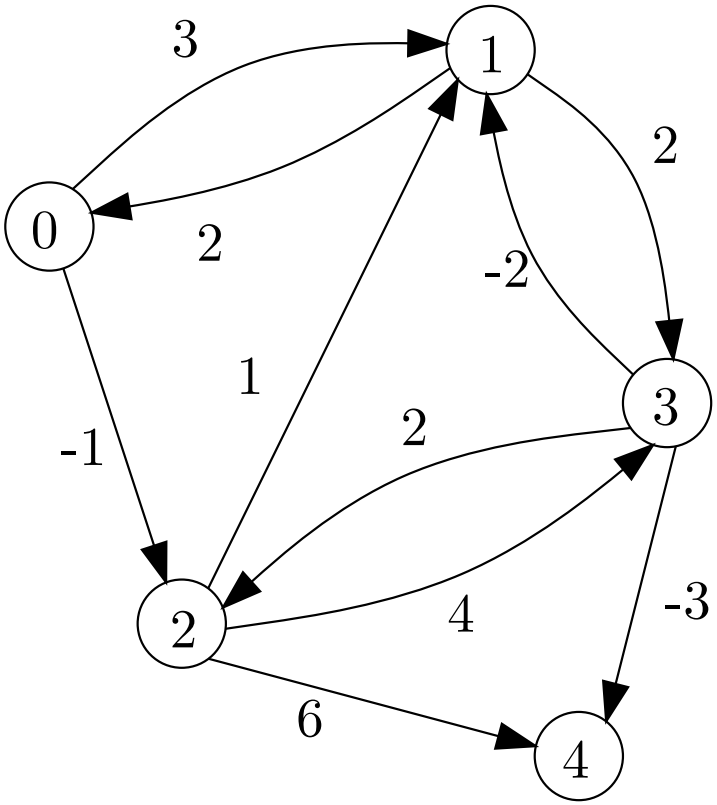
- Several algorithms are known; we present one, Floyd's algorithm. Alternative to running Dijkstra from each node.
- Number nodes (say from 0 to $n-1$) and at each step k, maintain matrix of shortest distances from node i to node j not passing through nodes higher than k. Update at each step to see whether node k shortens current best distance.
- Need triply nested `for` loop, so runs in $O(n^3)$ time. Better than Dijkstra for dense graphs, probably not for sparse ones.
- Based on Warshall's algorithm (just tells whether there is a path from nod e i to node j, not concerned with length).

Floyd's algorithm

algorithm Floyd(weighted digraph (G, c))
for $x \in V(G)$ **do**
    for $u \in V(G)$ **do**
        for $v \in V(G)$ **do**
            $c[u, v] \leftarrow \min\{c[u, v], c[u, x] + c[x, v]\}$

This algorithm is based on dynamic programming principles.
At the bottom of the outer `for` loop, for each $u, v \in V(G)$, $c[u, v]$ is the length of the shortest path from u to v passing through intermediate nodes that have been seen in `for` x loop.

Illustrating Floyd's algorithm



0	3	-1	∞	∞
2	0	∞	2	∞
∞	1	0	4	6
∞	-2	2	0	-3
∞	∞	∞	∞	0

adj/cost matrix

$x =$

0	3	-1
2	0	**1**
∞	1	0
∞	-2	2
∞	∞	∞

Minimum spanning tree problem

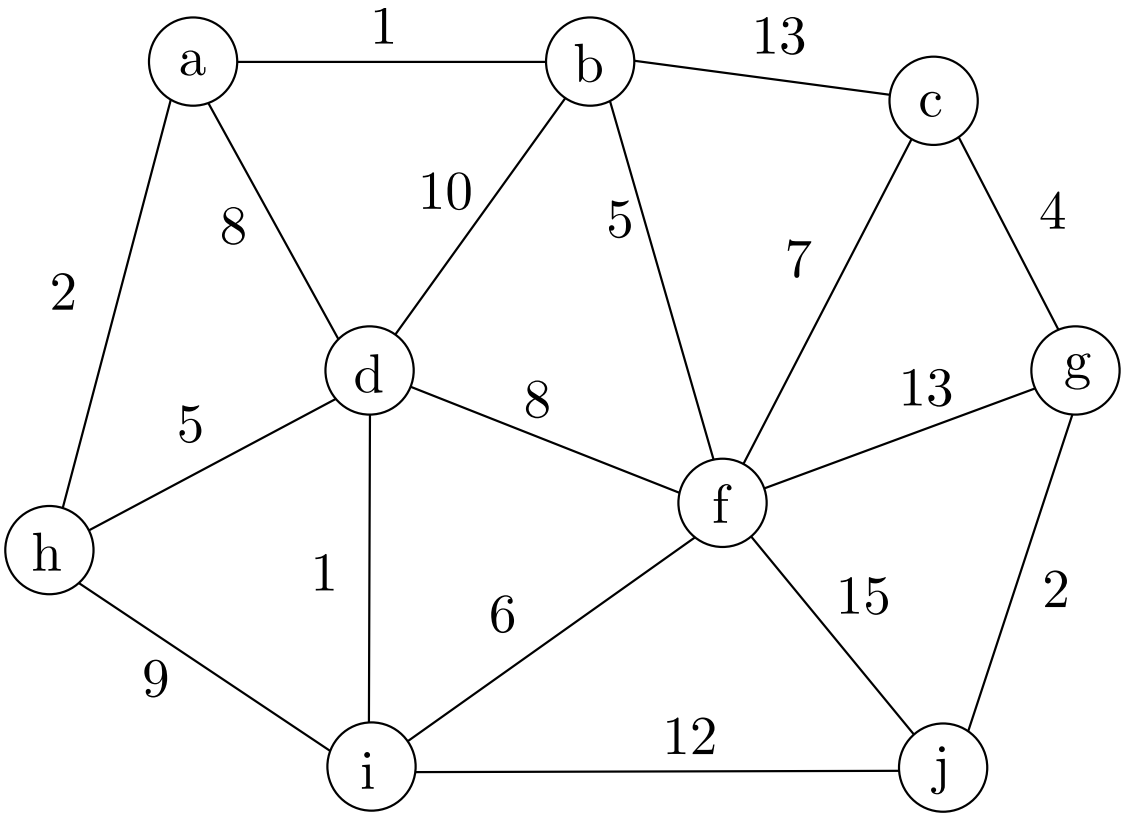
- Given a connected weighted graph, find a spanning tree (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.
- Two efficient greedy algorithms presented here: Prim's and Kruskal's.
- Each selects edges in order of increasing weight but avoids creating a cycle.
- Prim maintains a tree at each stage that grows to span; Kruskal maintains a forest whose trees coalesce into one spanning tree.
- Prim implementation very similar to Dijkstra, get $O(e + n \log n)$; Kruskal uses disjoint sets ADT and can be implemented to run in time $O(e \log n)$.

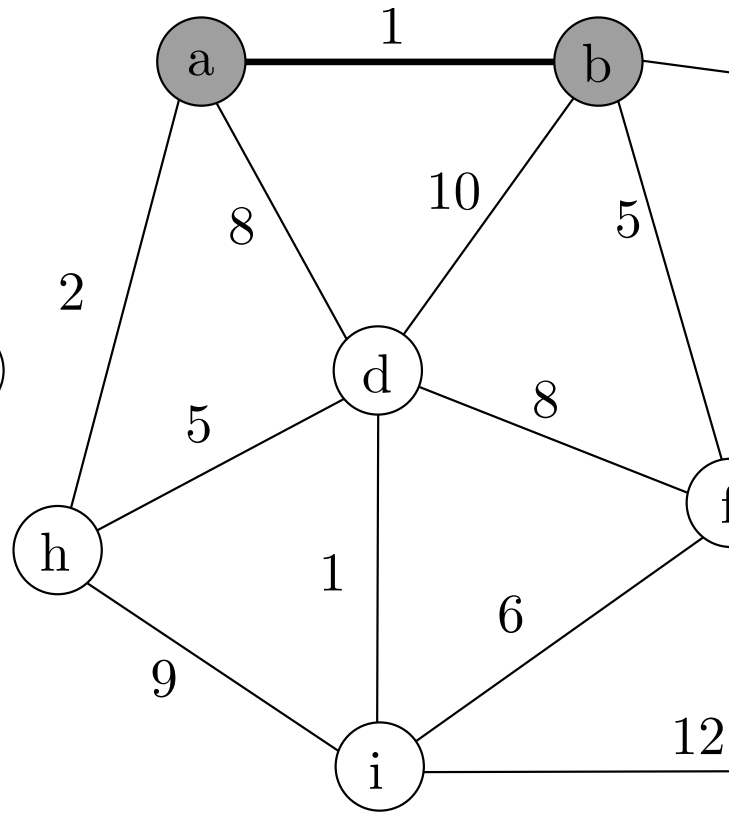
Prim's algorithm

algorithm Prim(weighted graph (G, c), vertex s)
array $w[n] = \{c[s, 0], c[s, 1], \ldots, c[s, n-1]\}$
$S \leftarrow \{s\}$ first vertex added to MST
while $S \neq V(G)$ **do**
    find $u \in V(G) \setminus S$ so that $w[u]$ is minimum
    $S \leftarrow S \cup \{u\}$ adding an edge adjacent to u to MST
    for $x \in V(G) \setminus S$ **do**
        $w[x] \leftarrow \min\{w[x], c[u, x]\}$
end while

Very similar to Dijkstra—should use a priority queue for selection of best edge weights $w[\ldots]$. Most time taken by EXTRACT-MIN and DECREASE-KEY operations.

Illustrating Prim's algorithm



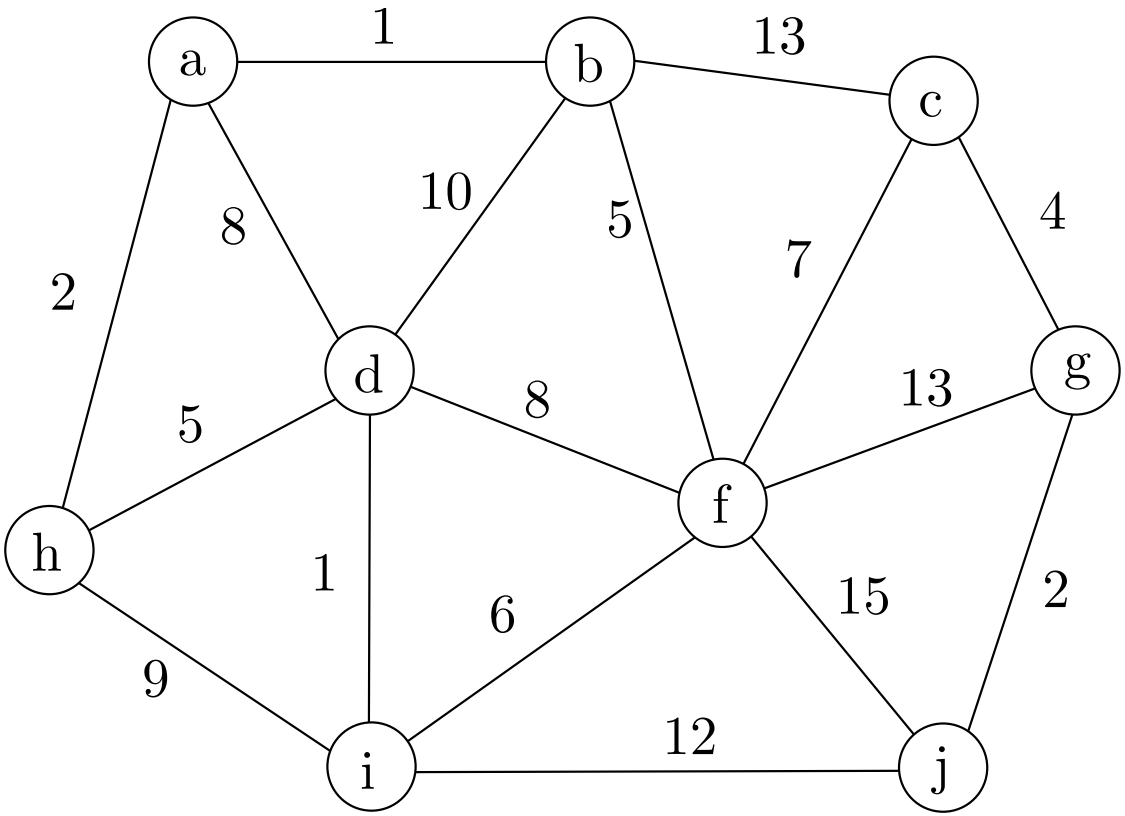


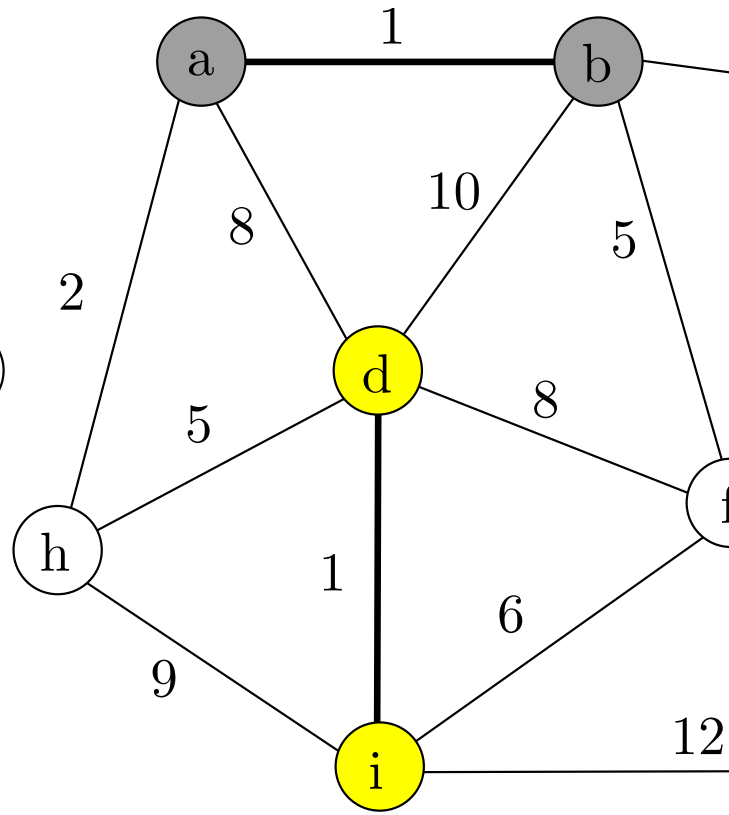
Kruskal's algorithm

algorithm Kruskal(weighted graph (G, c))
$T \leftarrow \emptyset$
insert $E(G)$ into a priority queue
for $e = \{u, v\} \in E(G)$ in increasing order of weight **do**
    if u and v are not in the same tree **then**
        $T \leftarrow T \cup \{e\}$
        merge the trees of u and v
    end if
end for

Keep track of the trees using disjoint sets ADT, with standard operations FIND and UNION. They can be implemented efficiently so that the main time taken is the sorting step.

Illustrating Kruskal's algorithm





Minimum spanning tree (MST) summary

Can you prove these two facts?

Fact

1. *The most expensive edge, if unique, of a cycle in an edge-weighted graph G is not in any MST. (Otherwise, at least one of those equally expensive edges of the cycle must not be in each MST.)*
2. *The minimum cost edge, if unique, between any non-empty strict subset S of $V(G)$ and the $V(G) \setminus S$ is in the MST. (Otherwise, at least one of these minimum cost edges is in each MST.)*

Other (di)graph optimization/decision problems

There are many more graph and network computational problems.

- Many do not have easy or polynomial-time solutions.
- However a few of them are in a special category in that their solutions can be varified in polynomial-time (NP).
- In addition, many of these are proven to be harder than anything else in NP (NP-complete).
- Other algorithm design techniques like backtracking, branch-and-bound or approximation algoirthms are needed (which are covered in CompSci 320).

Examples of NP-complete graph problems (part 1)

- Vertex Cover: Is there a subset of k vertices such that every edge is covered?
- Hamiltonian Path: Is there a path using all vertices?
- Dominating Set: Is there a subset D of k vertices such that each vertex is in the neighborhood (distance ≤ 1) of D?
- Feedback Arc Set: Is there a subset F of k nodes such that $G \setminus F$ is a DAG?
- Maximum Clique: Is there an induced subgraph of order k which is complete?
- Maximum Cut: Can the vertices of G be partitioned into two non-empty sets V_1 and V_2 such that the cost of the "cut" is at most k?

Examples of NP-complete graph problems (part 2)

- Induced Path: Is there an induced subgraph of order k which is a simple path?
- Bandwidth: Is there a linear ordering of V with bandwidth k or less? (each edge spans at most k vertices)
- Subgraph Isomorphism: Is H a sub(di)graph of G?
- Minimum Broadcast Time: Given an originating node for a digraph G, can we (point-to-point) broadcast to all other nodes in at most k time steps?
- Traveling Salesman: Is there a cycle of length $|G|$ in an edge-weighted graph G that costs at most c?
- Disjoint Connecting Paths: Given k pairs of source and sink vertices for a graph G, is there k vertex-disjoint paths connecting each pair?

Thank you!