Algorithms and Data Structures II

(COM2721, Spring 2024)

Graphs

(Haiko Müller)

UNIVERSITY OF LEEDS

Outline

Basic Definitions and Applications

Graph Traversal
- Breadth First Search
- Depth First Search

Connected Components

Connectivity in Directed Graphs
- DAGs and Topological Sort

Graphs

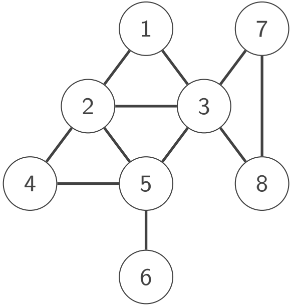
Graphs: Are an important and widespread tool to model networks, relations, and structures.

Example: Viennese underground network

Undirected Graphs

undirected graph: $G = (V, E)$

- V is the set of vertices (or nodes).
- E is the set of edges between pairs of vertices.
- captures pairwise relationships between objects.
- notation for edges between vertex a and vertex b: $\{a, b\}$ or $\{b, a\}$.
- alternatively also ab or ba or a–b or b–a.
- graph size parameters/notation: $n = |V|$, $m = |E|$



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

$E = \{1\text{–}2, 1\text{–}3, 2\text{–}3, 2\text{–}4, 2\text{–}5, 3\text{–}5, 3\text{–}7, 3\text{–}8, 4\text{–}5, 5\text{–}6, 7\text{–}8\}$

$n = 8$

$m = 11$

Undirected Graphs: Further Definitions

adjacent, incident, neighbourhood:
Let $e = \{u, v\}$ be an edge in E.

- u and v are adjacent, *i.e.*, u is a neighbour of v and v is a neighbour of u.
- $N(v) = \{w \mid \{v, w\} \in E\}$ is the open neighbourhood of v.
- $N[v] = \{v\} \cup N(v)$ is the closed neighbourhood of v.
- v (respectively, u) and e are incident.
- $\{u, v\} = \{v, u\}$.

degree: $\deg(v)$ denotes the degree of the vertex v.

- $\deg(v)$ is equal to the number of edges incident to v.
- We have $\sum\limits_{v \in V} \deg(v) = 2 \cdot |E|$ (Handshaking-Lemma).

Undirected Graphs: Further Definitions

Fundamental Definitions:

- multi-edge: more than one edge between the same two vertices.
- loop: an edge between a vertex and itself.

Simple Graph: An undirected graph without multi-edges and loops.

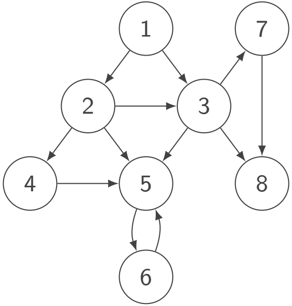
Hints:

- In this module we usually talk about finite simple graphs (if not stated otherwise).
- For certain application areas one also considers weighted graphs (with real-valued weights on the edges or vertices).

Directed Graphs

Directed Graph (Digraph): $G = (V, E)$

- V is the set of vertices (or nodes).
- E is the set of arcs (or directed edges) between pairs of vertices.
- notation for an arc from a to b: (a, b) or $a{\rightarrow}b$
- $(a, b) \neq (b, a)$



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

$$E = \{1{\rightarrow}2, 1{\rightarrow}3, 2{\rightarrow}3, 2{\rightarrow}4, 2{\rightarrow}5, 3{\rightarrow}5, 3{\rightarrow}7, 3{\rightarrow}8, 4{\rightarrow}5, 5{\rightarrow}6, 6{\rightarrow}5, 7{\rightarrow}8\}$$

$$n = 8$$

$$m = 12$$

Hint: Arcs in opposing directions are allowed in simple digraphs.

Directed Graphs: Further Definitions

Let $a = (u, v)$ be an arc in E, then:

- v is the head and u is the tail of a,
- u is an incoming neighbour of v and v is an outgoing neighbour of u,

- $N^-(v)$ is the set of incoming neighbours of v, *i.e.*, $N^-(v) = \{ u : (u, v) \in E \}$,
- $N^+(v)$ is the set of outgoing neighbours of v, *i.e.*, $N^+(v) = \{ u : (v, u) \in E \}$,
- $\deg^-(v)$ is in-degree of v, *i.e.*, $\deg^-(v) = |N^-(v)|$,
- $\deg^+(v)$ is out-degree of v, *i.e.*, $\deg^+(v) = |N^+(v)|$.

We have: $\deg(v) = \deg^+(v) + \deg^-(v)$.

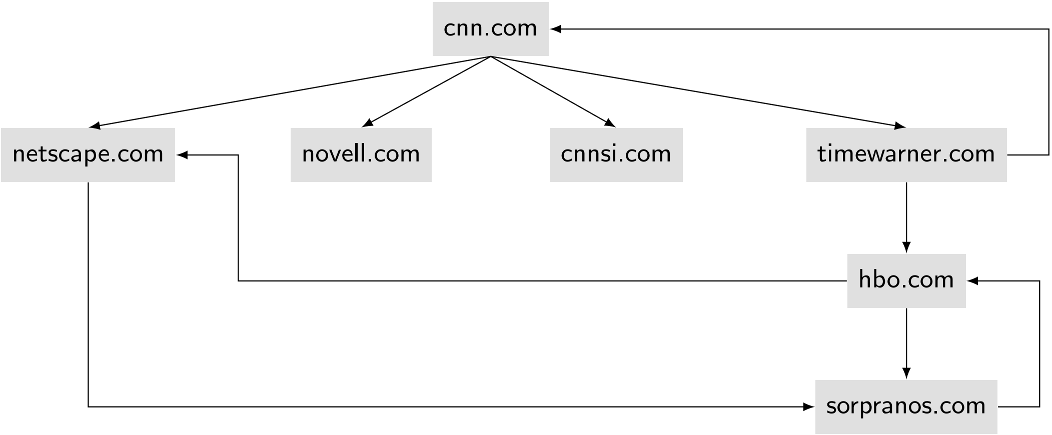
Some Graph Applications

Graph	*Nodes*	*Edges*
transportation	street intersections	highways
communication	computers	fibre optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

World Wide Web

Web graph.

- Node: web page.
- Edge: hyperlink from one page to another.



9-11 Terrorist Network

Social network graph.

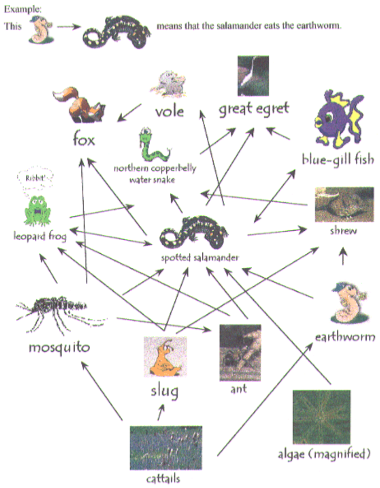
- Node: people.
- Edge: relationship between two people.

Reference: Valdis Krebs, http://www.firstmonday.org/issues/issue7_4/krebs

Ecological Food Web

Food web graph.

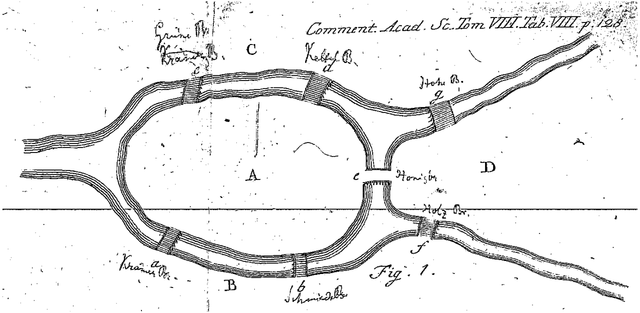
- Node = species.
- Edge = from prey to predator.



Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

Seven Bridges of Königsberg [Euler 1736]

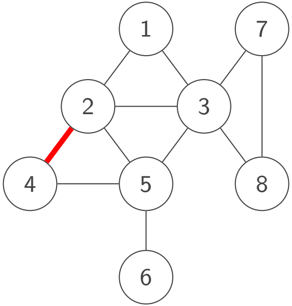
SOLVTIO PROBLEMATIS
AD
GEOMETRIAM SITVS
PERTINENTIS.
AVCTORE
Leonh. Eulero.



Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if $\{u, v\}$ is an edge.

- vertices: $1, 2, \ldots, n$.
- Two representations of each edge.
- Space proportional to n^2.
- Checking if $\{u, v\}$ is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.

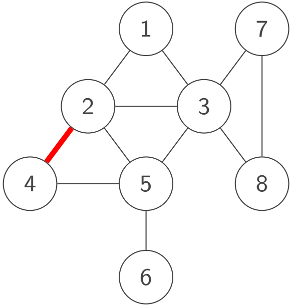


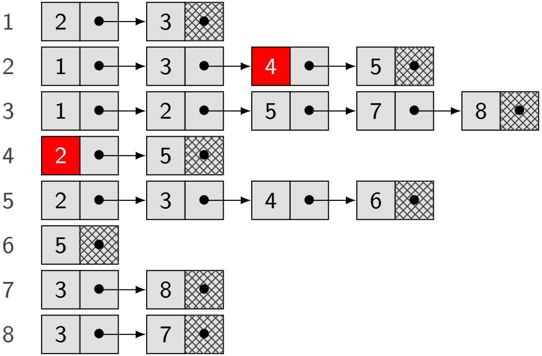
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- vertices: $1, 2, \ldots, n$.
- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if $\{u, v\}$ is an edge takes $\mathcal{O}(\deg(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.





Adjacency matrix or adjacency lists?

Number of edges:

- a graph can have up to $\binom{n}{2} = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$ edges.
- for such a dense graph both representations use the same amount of space and are comparable.

Practice:

- graphs coming from applications are usually much less dense.
- often $m = \mathcal{O}(n)$.
- in these cases the representation via adjacency lists is preferable.

Hint: If we say that an algorithm runs in linear-time we assume the representation via adjacency lists and refer to a run-time of $\mathcal{O}(n+m)$.

Undirected Graphs: output all edges

Adjacency matrix:

```
input  : adjacency matrix M and vertices numbered 0, ..., n − 1
for u ← 0 to n − 2 do
    for v ← u + 1 to n − 1 do
        if M[u, v] == 1 then
            print the edge {u, v}
```

Adjacency list:

```
input  : adjacency lists L and vertices numbered 0, ..., n − 1
for u ← 0 to n − 1 do
    for v ∈ L[u] do
        if u < v then print the edge {u, v};
```

Directed Graphs: output all edge

Adjacency matrix:

```
input  : adjacency matrix M and vertices numbered 0, ..., n − 1
for u ← 0 to n − 1 do
    for v ← 0 to n − 1 do
        if M[u, v] == 1 then
            print the arc (u, v)
```

Adjacency list:

```
input  : adjacency lists L and vertices numbered 0, ..., n − 1
for u ← 0 to n − 1 do
    for v ∈ L[u] do
        print the arc (u, v)
```

Paths and Connectivity

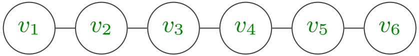
Definition: A path in an undirected graph $G = (V, E)$ is a sequence $(v_1, v_2, \ldots, v_{k-1}, v_k)$, $k \geq 1$, of vertices with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.
The length of a path is $k - 1$.

Hints: We also say that the path goes from v_1 to v_k and that the path is a v_1-v_k-path.

Definition: A vertex u is reachable from a vertex v in G, if G contains an u-v-path.

Definition: An undirected graph is connected, if for every pair u and v of vertices, there is a path between u and v.

A path of length 5:



Shortest Paths and Distance

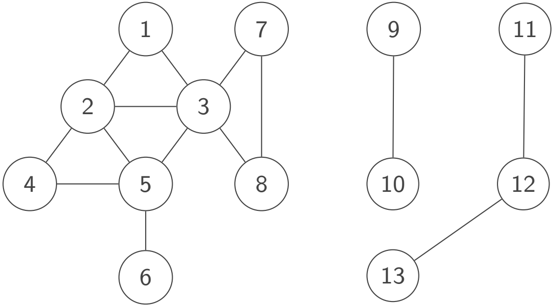
Definition: A shortest u-v-path between two vertices u and v is a path between u and v of shortest length. Note that a shortest path is always simple, *i.e.*, no vertex occurs twice on the path.

Definition: The distance between vertices u and v in an undirected graph is the length of a shortest u-v-path.

Remark: If u is not reachable from v, we assume the distance to be ∞.

Connectivity: Example

disconnected graph:

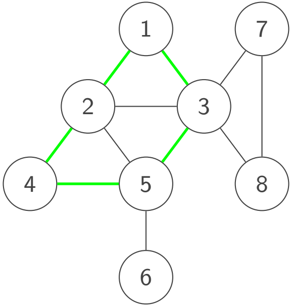


not connected: For instance, there is no path between 1 and 10.

Example for connectedness: The vertices 1 to 8 and their incident edges form a connected graph.

Cycle

Definition: A cycle is a path $(v_1, v_2, \ldots, v_{k-1}, v_k)$ for which $\{v_1, v_k\}$ is an edge too, $k \geq 3$. The length of this cycle is k.



Example of a cycle: $C = (1, 2, 4, 5, 3)$

Paths and Cycles in Digraphs

Definition: A path in a digraph $G = (V, E)$ is a sequence $(v_1, v_2, \ldots, v_{k-1}, v_k)$, $k \geq 1$, of vertices with the property that every consecutive pair v_i, v_{i+1} is connected by a directed arc from v_i to v_{i+1}.

Remarks:

- The path goes from a start vertex to an end vertex.
  The reverse does not necessarily hold.
- v can be reached from u, if there is an u-v-path.
- shortest u-v-paths are simple (*i.e.*, do not contain any vertex twice).

Cycle: A directed cycle is a path $(v_1, v_2, \ldots, v_{k-1}, v_k)$ with $v_k \rightarrow v_1$ and $k \geq 2$.

Trees

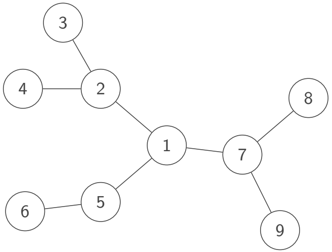
Definition: An undirected graph is a tree, if it is connected and acyclic (*i.e.*, does not contain any cycle).

Theorem: Let G be an undirected graph with n vertices.
Every two of the following conditions imply the third:

- G is connected.
- G does not contain a cycle.
- G has $n-1$ edges.

Properties and Notions of Trees:

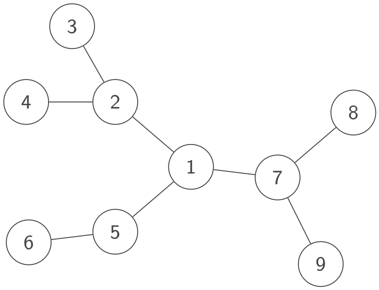
- a leaf is a vertex of degree one
- every tree with at least 2 vertices has at least one leaf
- there is a unique path between any two vertices in a tree



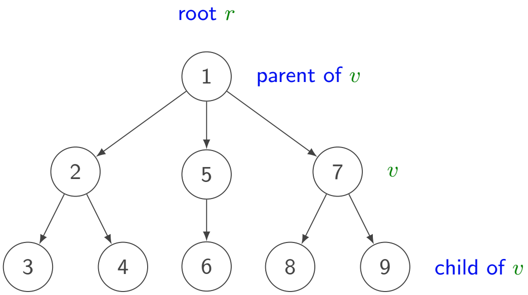
Rooted Tree or Arborescence

Rooted Tree: Obtained from a tree by choosing an arbitrary vertex r as the root and directing all edges away from r.

Importance: Modelling of hierarchical structures.



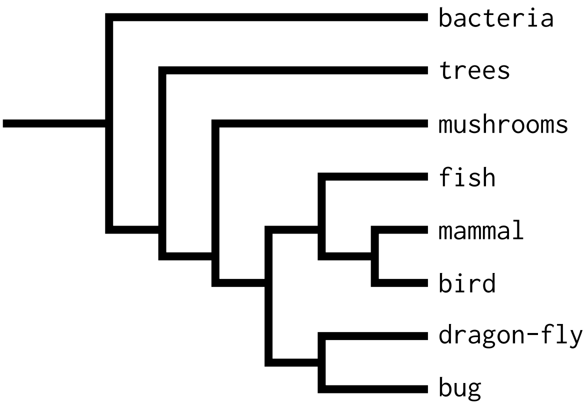
a tree



A corresponding rooted tree with root 1

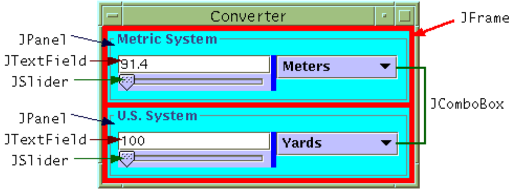
Phylogenetic Tree

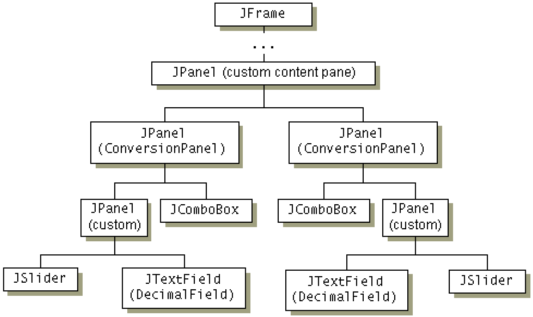
Phylogenetic Tree: Models the evolutionary relationships between different species.



GUI-Hierarchies

GUI-Hierarchies: Describe the organisation of GUI components.





Outline

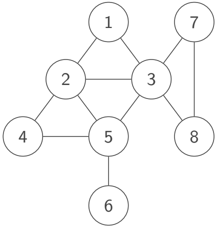
Graph Traversal: Applications

s-t connectivity problem: Is there a path between two given vertices s and t?

s-t shortest paths: The length of a shortest path between s and t (*i.e.*, the distance between s and t)?

Applications:

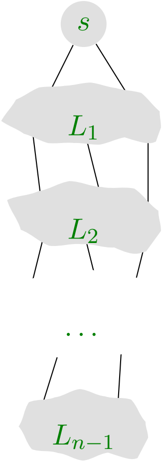
- facebook.
- maze traversal.
- Kevin Bacon number.
- Fewest number of hops in a communication network.



Outline

Breadth First Search (BFS)

BFS Intuition: Explore outward from a start vertex s in all possible directions, adding nodes one "layer" at a time.



BFS Algorithm:

- $L_0 = \{s\}$.
- $L_1 =$ all neighbours of vertices in L_0.
- $L_2 =$ all vertices that are not in L_0 or L_1 and that are adjacent to a vertex in L_1.
- $L_{i+1} =$ all vertices that are not in any previous layer and that are connected via an edge to a vertex in L_i.

BFS: Theorem

Theorem

L_i contains all vertices of distance i from s for every i.

Proof.

Let $(v_0, v_1, v_2, \ldots, v_n)$ be a shortest path between v_0 and v_n.

- v_0 is in L_0.
- v_1 is in L_1, since v_1 is a neighbour of v_0.
- v_2 is in L_2, since v_2 is a neighbour of v_1 and not a neighbour of v_0 (otherwise there would be a shorter path between v_0 and v_n using the edge between v_0 and v_2).
- This argument applies for all other vertices, which implies that $v_i \in L_i$ for every i. $\square$

BFS: Implementation using a Queue

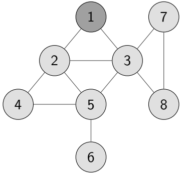
Implementation: Array visited, Queue Q

```
input  : graph G = (V, E) and start vertex s.

for v ∈ V do visited[v] ← false;
Q ← {s}; visited[s] ← true;
while Q ≠ ∅ do
    remove the first vertex u from Q            /* dequeue */;
    print u                  /* or any other operation on u */;
    for v ∈ N(u) do
        if !visited[v] then
            visited[v] ← true;
            add v at the end of Q                            /* enqueue */
```

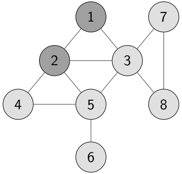
- $N(u)$ is the set of neighbours of u in G

BFS: Example

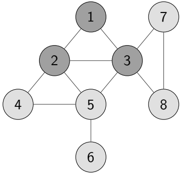
Possible execution: start vertex = 1, visited vertices are in darkgray



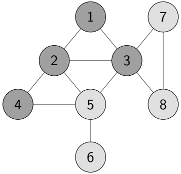
(a) 1



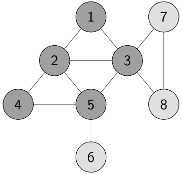
(b) 1,2



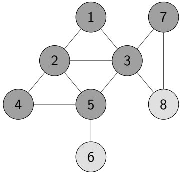
(c) 1,2,3



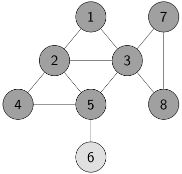
(d) 1,2,3,4



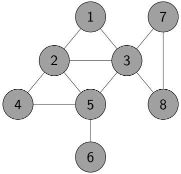
(e) 1,2,3,4,5



(f) 1,2,3,4,5,7



(g) 1,2,3,4,5,7,8



(h) 1,2,3,4,5,7,8,6

BFS: Analysis

Theorem: BFS has a run-time of $\mathcal{O}(n+m)$.

Run-time: We need to consider three parts:

- initialisation
- while-loop
- for-loop (inside while-loop)

BFS: Analysis

initialisation:

- every vertex is considered once
- constant time per vertex.
- Therefore, the run-time for initialisation is $\mathcal{O}(n)$.

while-loop:

- every vertex is added to Q at most once, since it will be marked visited thereafter
- therefore, every vertex is considered only once in the while-loop.

BFS: Analysis

for-loop (inside while-loop):

- Let u be the current vertex before entering the for-loop.
- Then all neighbours of u are considered in the for-loop.
- Therefore, the for-loop is executed exactly $\deg(u)$ times with each execution requiring only constant time.

Total:

- The total run-time is therefore $\mathcal{O}\big(n + \sum_{u \in V} \deg(u)\big)$.
- Since $\sum_{u \in V} \deg(u) = 2m$, we obtain the run-time of $\mathcal{O}(n + m)$.

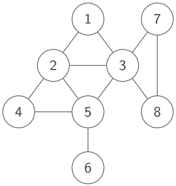
BFS-Tree

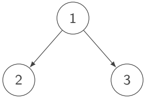
BFS-Tree: BFS creates a tree (BFS-tree), whose root is the start vertex s containing all vertices reachable from s.

Creation: Starting at s, whenever a vertex v is visited for the first time as a neighbour of a vertex u it becomes the child of u in the BFS-tree.

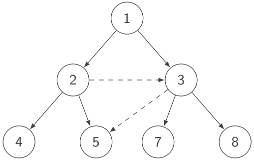
BFS-Tree: Property

Property: Let T be a BFS-tree of $G = (V, E)$ and let $\{x, y\}$ be an edge of G. Then the level of x and y in T can differ by at most one.

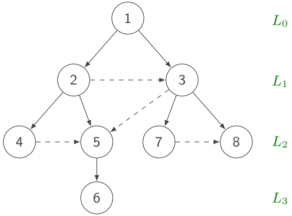




(a)



(b)



(c)

BFS: Calculating the Levels

Application of BFS: Calculate the level for every vertex.

Implementation: Array level, Queue Q

```
input  : graph G = (V, E) and start vertex s.

for v ∈ V do level[v] ← −1;
Q ← {s}; level[s] ← 0;
while Q ≠ ∅ do
    remove the first vertex u from Q;
    for v ∈ N(u) do
        if level[v]==−1 then
            level[v] ← level[u]+1;
            add v at the end of Q
```

Outline

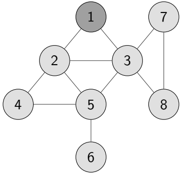
Depth First Search (DFS)

Idea: From a visited node go first to another not yet visited neighbour (recursive call) before visiting its other neighbours.

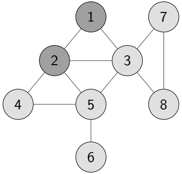
```
// global array visited
input  : graph G = (V, E) and start vertex s.

begin
    for v ∈ V do visited[v] ← false;
    DFS(G, s);

procedure DFS(G, v)
    visited[v] ← true;
    print v                    /* or any other operation on v */;
    for n ∈ N(v) do
        if !visited[n] then
            DFS(G, n)
```

DFS: Example

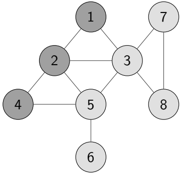
Possible run: start node = 1, visited nodes are darkgray



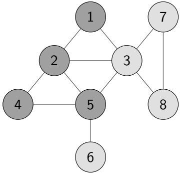
(a) 1



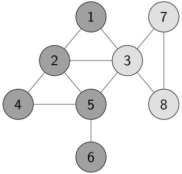
(b) 1,2



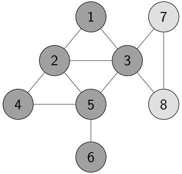
(c) 1,2,4



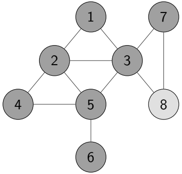
(d) 1,2,4,5



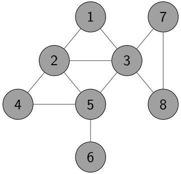
(e) 1,2,4,5,6



(f) 1,2,4,5,6,3



(g) 1,2,4,5,6,3,7



(h) 1,2,4,5,6,3,7,8

DFS: Analysis

Theorem: DFS runs in time $\mathcal{O}(n + m)$.

Runtime: We need to consider:

- initialisation
- for-loop

Initialisation:

- initialisation takes time $\mathcal{O}(n)$.
- DFS(G,v) is called at most once per node.

DFS: Analysis

for-loop in DFS(G,v):

- considers all $\deg(v)$ nodes n in the adjacency list of v.
- therefore, for loop is executed $\deg(v)$ times.
- all commands within the for-loop take constant time (apart from the recursive call of DFS(G,v), whose run-time is considered in the analysis for node v).

Total:

- therefore, the total the run-time is $\mathcal{O}\big(n + \sum_{v \in V} \deg(v)\big)$.
- since $\sum_{v \in V} \deg(v) = 2m$, we obtain $\mathcal{O}(n + m)$.

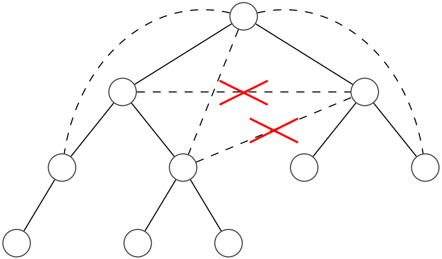
DFS-Tree

Creating a DFS-tree works in the same manner as for BFS, *i.e.*:

Starting at s, whenever a vertex v is visited for the first time as a neighbour of a vertex u it becomes the child of u in the DFS-tree.

DFS-tree property:

- the edges of the graph must be between vertices on the same root-to-leaf path
- in other words: the edges of the graph must not be between different branches of the DFS-tree

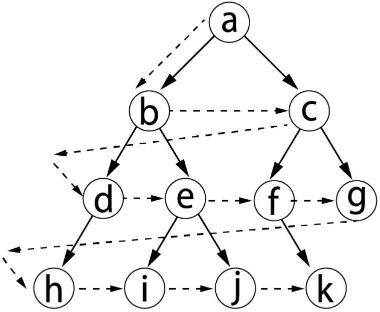


DFS vs BFS Search Order

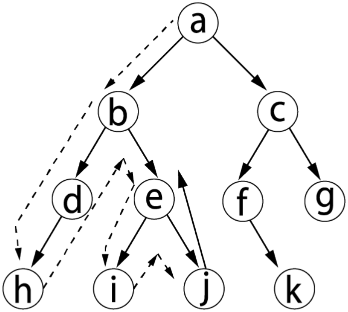
Search Order: DFS differs from BFS:

- one first tries to get as far away from the start node as possible.
- once there are no unvisited nodes left in the neighbourhood, the recursion backtracks to find the first node having an unvisited neighbour
- edges in the graph only go between ancestors and descendants in the DFS-tree but can jump levels in the tree

Example: DFS vs BFS



Breadth-first search



Depth-first search

Search Order:

- BFS: a, b, c, d, e, f, g, h ,i, j, k
- DFS: a, b, d, h, e, i, j, c, f, k, g

Outline

Connected Components

Connectivity (Reminder): An undirected graph is connected if there is a path between every pair of nodes in the graph.

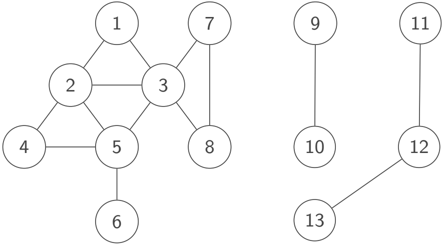
Disconnected: If the graph contains a pair of nodes without a path between them, the graph is not connected.

Subgraph: A graph $G_1 = (V_1, E_1)$ is a subgraph of a graph $G_2 = (V_2, E_2)$, if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

Connected Component: A maximal connected subgraph of a graph is called a connected component. A disconnected graph is the disjoint union of its connected components.

Connected Component

Example: A disconnected graph with 3 connected components.



Connected Component

Identifying connected components: Find all nodes that can be reached from s.

Connected Component

Identifying connected components: Find all nodes that can be reached from s.

Solution:

- call DFS(G,s) or BFS(G,s).
- A node u is reachable from s if and only if visited[u]=true.

Counting Connected Components

DFSNUM algorithm:

```
input  : graph G = (V, E), global array visited (shared with DFS).

procedure DFSNUM(G)
    for v ∈ V do visited[v] ← false;
    c ← 0;
    for v ∈ V do
        if !visited[v] then
            c ← c + 1;
            DFS(G, v)
    return c
```

Counting Connected Components

Runtime: The runtime is in $\mathcal{O}(n+m)$.

Analysis:

- Let $G = (V, E)$ be the given graph and $G_1 = (V_1, E_1), \ldots, G_r = (V_r, E_r)$ its connected components.
- Let $|V| = n$, $|E| = m$, and $|V_i| = n_i$ and $|E_i| = m_i$, for every i with $1 \leq i \leq r$.
- Clearly $n = n_1 + \cdots + n_r$ and $m = m_1 + \cdots + m_r$.
- for every connected component G_i $(1 \leq i \leq r)$ the algorithm performs DFS. This has a runtime of $\mathcal{O}(n_i + m_i)$.
- initialisation requires $\mathcal{O}(n)$ time.
- therefore, we obtain $\mathcal{O}\left(n + \sum_{i=1}^{r}(n_i + m_i)\right) = \mathcal{O}(2n + m) = \mathcal{O}(n + m)$ as the total run-time.

Outline

Search in Directed Graphs

Directed Reachability: Find all nodes that can be reached from a given node s.

directed shortest s-t path: Find a shortest path from s to t for two given nodes s and t.

Search in directed graphs: BFS and DFS can also be applied for directed graphs.

Example Web-crawler: Start from a website s. Find all websites that are directly or indirectly linked from s.

Strong Connectivity

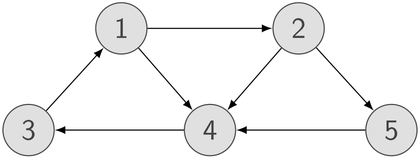
Definition: Two nodes u and v are strongly connected if there is a path from u to v and vice versa.

Definition: A directed graph is strongly connected if every pair of nodes is strongly connected.

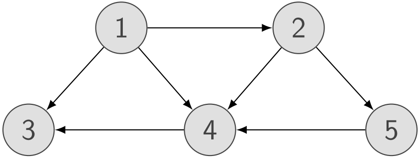
Remark: A directed graph is called weakly connected if the underlying undirected graph, *i.e.*, the undirected graph that one obtains by ignoring the direction of edges, is connected.

Strong Connectivity: Example

Strongly connected:



Not strongly connected (but weakly connected): node 1 cannot be reached by any other node; node 3 cannot reach any other node.



Strong Connectivity

Lemma

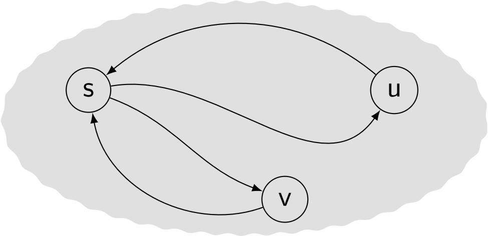
Let s be an arbitrary node in a directed graph G. G is strongly connected if and only if every node can be reached from s and every node can reach s.

Proof:

$\Rightarrow$ Follows from the definition.

$\Leftarrow$ path from u to v: combine a u-s path with a s-v path. path from v to u: combine a v-s path with a s-u path.

□ *paths might and are allowed to overlap*



Strong Connectivity: Algorithm

Theorem: Testing whether G is strongly connected can be achieved in $\mathcal{O}(n+m)$ time.

Proof:

- choose an arbitrary node s.
- execute BFS with start node s in.
- execute BFS with start node s in G^{rev}.
- return yes if and only if all nodes can be reached in both executions of BFS above.
- correctness follows immediately from the previous lemma. □

□ *G^{rev} obtained from G by reversing direction of every arc in G*

Outline

Directed Acyclic Graph (DAG)

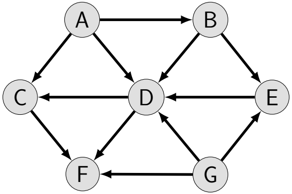
Definition: A DAG is a directed graph with no directed cycles.

Example: Ordering Constraint: arc (u, v) means that task u needs to be achieved before task v.

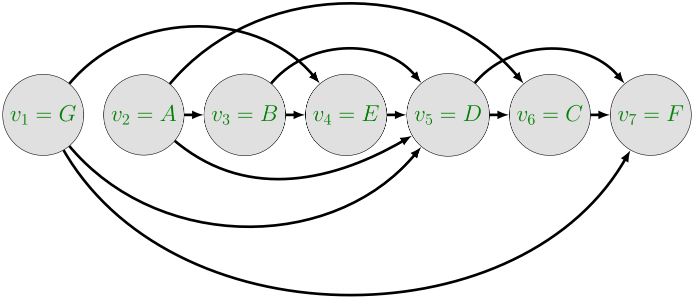
Definition: A topological ordering of a directed graph $G = (V, E)$ is a linear order of its nodes, denoted by $v_1, v_2, \ldots, v_n$, such that $i < j$ for every arc (v_i, v_j).

Topological Order: Example

A DAG:



A topological ordering:



Ordering Constraints

Ordering Constraints: arc (u, v) means that task u has to be achieved before task v.

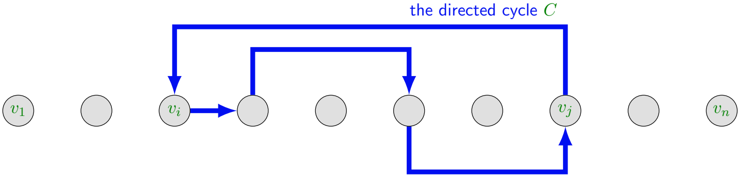
Applications:

- Preconditions for modules: module u has to be completed before module v.
- Compilation: class u has to be compiled before class v.
- pipeline between processes: output of process u is required for the input to process v.

DAG

Lemma: If G has a topological ordering then G is a DAG.
Proof: (by contradiction)

- Assume that G has a topological ordering $v_1, \ldots, v_n$ and contains a directed cycle C.
- Let v_i be the node with the smallest index in C and let v_j be the node before v_i in C, *i.e.*, there is an arc (v_j, v_i).
- Due to the choice of i, it holds that $i < j$.
- On the other hand, because (v_j, v_i) is an arc and $v_1, \ldots, v_n$ is a topological ordering it should be the case that $j < i$. A contradiction. $\square$



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

DAG

Lemma: If G has a topological ordering then G is a DAG.

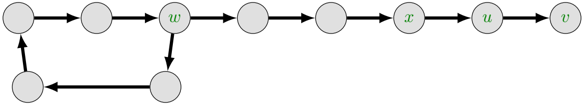
Question: Does every DAG have a topological ordering?

Question: If so, how to we calculate it?

DAG

Lemma: If G is a DAG then G has a node without incoming arcs.
Proof: (by contradiction)

- Assume that G is a DAG and every node has at least one incoming arc.
- Chose an arbitrary node v and follow the arcs from v in the reverse direction. Since v has at least one incoming arc (u, v), we can reach u in this manner.
- Since u has at least one incoming arc (x, u), we can reach x in this manner.
- This can be repeated until we reach a node w for a second time; since G has only finitely many nodes.
- Then the sequence C of nodes between two visits of the same node w is a cycle, a contradiction. □



DAG

Lemma: If G is a DAG, then G has a topological ordering.

Proof: (Induction for n)

- induction start: true if $n = 1$.
- Consider a DAG G with $n > 1$ nodes. Take a node v without any incoming arcs.
- Consider the digraph $G - \{v\}$, *i.e.*, the digraph G after removing v and all its incident arcs.
- $G - \{v\}$ is a DAG, since removing v cannot create any cycles.
- By the induction hypothesis, it holds that $G - \{v\}$ has a topological ordering.
- place v at the first position of a topological ordering and append the topological ordering for $G - \{v\}$. This is valid since v has no incoming arcs. $\square$

Topological Ordering

Algorithm: auxiliary array count for removal of nodes, initially empty queue/stack Q.

input : DAG $G = (V, E)$
for $v \in V$ **do** count$[v] \leftarrow 0$;
for $v \in V$ **do**
  for $(v, w) \in E$ **do** count$[w] \leftarrow$ count$[w]+1$;
for $v \in V$ **do**
  if count$[v]$ $==$ 0 **then**
    Add v to the beginning of Q
while Q is non-empty **do**
  remove the first element v from Q;
  `print` v ;
  for $(v, w) \in E$ **do**
    count$[w] \leftarrow$ count$[w]-1$;
    if count$[w]$ $==$ 0 **then** Add w to the beginning of Q ;

Topological Ordering: Runtime

Theorem: The algorithm finds a topological ordering in time $\mathcal{O}(n+m)$.

Proof: We need to consider the following parts:

- initialisation
  - first for-loop for count.
  - second (nested) for-loop.
  - third for-loop for generating Q.
- while-loop (with nested for-loop).

initialisation:

- the first foreach-loop, which initialises count, takes time $\mathcal{O}(n)$.
- for the second nested for-loops: the inner-loop is executed $\deg^+(v)$ times for every node v. This takes time $\mathcal{O}(n+m)$.
- the third for-loop for generating Q takes time $\mathcal{O}(n)$.
- Therefore, the initialisation requires time $\mathcal{O}(n+m)$.

Topological Ordering: Runtime

while-loop:

- every node v is removed at most once from Q.
- therefore, the while-loop is executed at most once per node.

foreach-loop:

- Let v be the currently active node before the for-loop is executed.
- then the for-loop is executed once for every outgoing neighbour w of v.
- those are exactly $\deg^+(v)$ many. Therefore the for-loop is executed $\deg^+(v)$ times and the single commands inside the for-loop take constant time.
- every node w is added at most once to Q.

Topological Ordering: Runtime

Total:

- Initialisation takes time $\mathcal{O}(n+m)$
- while-loop takes time $\mathcal{O}(n+m)$
- therefore $\mathcal{O}(n+m)$ is the total runtime.

Topological sort: find an ordering à la DFS

The idea of topological sort:

- Start DFS at the first unvisited vertex.
- Follow only the outgoing edges, *i.e.* replace $N(v)$ by $N^+(v)$.
- Number a vertex (in decreasing order) after all its out-neighbours are numbered.
- Repeat until no unvisited vertices are left.

Topological sort: the pseudocode

input : a dag $G = (V, A)$
output: a topological ordering $\sigma : V \to \mathbb{N}$ of G

begin
  $i \leftarrow |V|$;
  for $v \in V$ **do** mark v unvisited;
  while there is a unvisited vertex $v \in V$ **do** `TS-visit`(v);

procedure `TS-visit` (v)
  if v is marked stacked **then stop**;
  if v is marked unvisited **then**
    mark v stacked;
    for $w \in N^+(v)$ **do** `TS-visit`(w);
    mark v visited;
    $\sigma(v) \leftarrow i$; $i \leftarrow i - 1$

Topological sort: correctness

Lemma

The algorithm topological sort computes a topological ordering of its input graph.

Proof.

Let vw be a directed edge of the digraph $G = (V, A)$.
When `TS-visit`(v) is executed the vertex $w \in N^+(v)$ is considered in line 10.
If w is already visited at the time then $\sigma(v) \leq i < \sigma(w)$ holds.
Otherwise `TS-visit`(w) will terminate before `TS-visit`(v) terminates and therefore we have $\sigma(v) < \sigma(w)$. $\square$

Topological sort: running time

Lemma

Topological sort runs in linear time.

Proof.

As for DFS. $\square$