<table>
<thead>
<tr>
<th></th>
<th>Understanding and analyzing algorithms</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Using graphs and graph algorithms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.1 Graphs and their representation, Trees</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.2 Depth-First-Search (DFS) and Breadth-First-Search (BFS)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.3 Minimum Spanning Trees (MST)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.4 Topological Sorting (TopSort)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.5 Single-Source-Shortest-Paths in Networks (SSSP)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.6 Euler and Hamilton Tours</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Using combinatorial reasoning to quantitatively analyze algorithms and systems</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Using probability to algorithm analysis</td>
<td></td>
</tr>
</tbody>
</table>
Graphs and their representation
What is a graph?

A **directed graph** $G$ is
- a set of vertices $V$, also called nodes, together with
- a set of edges $E$, each pointing from one vertex to another (denoted with an arrow)

An **undirected graph** has the property that if there is an edge from $v$ to $w$, then there is also an edge from $w$ to $v$. In this case, we don't need arrows on the edges (and the edges are assumed to go both ways).
Graphs are important for a lot of different applications. Graphs capture relationships between objects in a visual way.
Graphs

Graphs are important for a lot of different applications. Graphs capture relationships between objects in a visual way.

Quelle: Simchi-Levi et al., Designing & Managing the Supply Chain, 2007
Historical Background

1736: Leonhard Euler
„7 Bridges of Königsberg“
Do you think we can
find a path that crosses
each bridge once?
**Historical Background**

1736: Leonhard Euler

„7 Bridges of Königsberg“

Do you think we can find a path that crosses each bridge once?

**Observation:** It doesn't matter where on the north side you are. You must come and go via a bridge. Collapse the entire area to a point.

We are looking for a path that includes each edge exactly once:

An **Euler Tour**, or Eulerian Tour, of a graph is a path where each edge occurs exactly once.

So we are looking for an Euler Tour in the Seven Bridges puzzle.
Euler Tours and Hamilton Tours

An **Euler Tour** of a graph is a path where each edge occurs exactly once.

→ Euler Tours (when they exist) can be computed in polynomial time.

A **Hamilton Tour** of a graph is a path where each vertex occurs exactly once.

→ Hamilton Tours (in the general case) are computational intractable.
Graphs

**Graph**
- \( G = (V, E) \)
- \( V \) = finite set of Vertices, \( n=|V| \)
- \( E \) = finite set of Edges, \( m=|E| \)

**Undirected Graph**
- \( e = \{u,v\} \) \( e_1 = \{A,B\}, e_4 = \{A,D\}, e_5 = \{B,D\} \)

**Directed Graph**
- \( e = (u,v) \) \( e_1 = (A,B), e_4 = (D,A), e_5 = (B,D) \)

**Network**
- a directed graph together with a cost function \( c: E \rightarrow \mathbb{R} \)
Graphs: Definitions

- (direct, in- and out) neighbors, (in- and out-) degree, sources, sinks

- path (simple path = trail), (self-) loop, (simple) cycle (circuit)

- simple undirected graph: no parallel edges and no loops

- simple directed graph: no parallel edges and no loops
**Connected**

An undirected graph $G$ is connected if for any ordered pair of vertices $(v,w)$ there is a path from $v$ to $w$.

**Disconnected**

An undirected graph $G$ is disconnected if there exists some ordered pair of vertices $(v,w)$ for which there is no path from $v$ to $w$. 
(Vertex) Connectivity (undirected graphs)

- **connected**
  An undirected graph $G$ is connected if for any ordered pair of vertices $(v, w)$ there is a path from $v$ to $w$.

- **disconnected**
  An undirected graph $G$ is disconnected if there exists some ordered pair of vertices $(v, w)$ for which there is no path from $v$ to $w$.

Disconnected graphs can be broken up into pieces where each is connected. These pieces are called **Connected Components**.
(Vertex) Connectivity (directed graphs)

- **strongly connected**
  There is a path from \( u \) to \( v \) and from \( v \) to \( u \) whenever \( u \) and \( v \) are vertices in the graph.

- **weakly connected**
  There is a path between every two vertices in the underlying undirected graph.
A **tree** is a connected undirected graph with no simple cycles.

- Any tree is a simple graph.
- Any tree has a unique simple path between any two of its vertices.
- A tree with $n$ vertices has $n-1$ edges.

A **rooted tree** is a tree in which one vertex has been designated as the root and every edge is directed away from the root.

If $v$ is a vertex other than the root, the **parent** of $v$ is the unique vertex $u$ such that there is a directed edge from $u$ to $v$.

When $u$ is the parent of $v$, $v$ is called a **child** of $u$.

A vertex of a rooted tree is called a **leaf** if it has no children. Vertices that have children are called **internal vertices**.
Trees: Definitions

The **level** of a vertex $v$ in a rooted tree is the length of the unique path from the root to its vertex. The level of the root is defined to be zero.

The **height** of a rooted tree is the maximum of the levels of vertices.

A rooted binary tree of height $h$ is **balanced** if all leaves are at levels $h$ or $h-1$.

A **binary tree** is a rooted tree where every vertex has no more than two children.
- There are at most $2^h$ leaves in a binary tree of height $h$.

In a **full binary tree** every internal vertex has exactly two children.
- A full binary tree with $i$ internal vertices contains $n=2i+1$ vertices.
Max. Number of Edges in Directed Graphs

If $G$ is a directed graph with $n$ vertices, what is the maximum number of ordered pairs of vertices $(v,w)$ that could be connected by edges in $G$?

A) $n$
B) $2n$
C) $n^2$
D) $n(n-1)/2$
E) $2^n$
Max. Number of Edges in Directed Graphs

If $G$ is a directed graph with $n$ vertices, what is the maximum number of ordered pairs of vertices $(v,w)$ that could be connected by edges in $G$?

A) $n$
B) $2n$
C) $n^2$
D) $n(n-1)/2$
E) $2^n$
Max. Number of Edges in Undirected Graphs

If $G$ is a simple undirected graph with $n$ vertices, what is the maximum number $m$ of edges that $G$ can have?

A) $n^2$
B) $n^2/2$
C) $n(n-1)/2$
D) $n(n+1)/2$
E) $n$
Max. Number of Edges in Undirected Graphs

If $G$ is a simple undirected graph with $n$ vertices, what is the maximum number $m$ of edges that $G$ can have?

A) $n^2$
B) $n^2/2$
C) $n(n-1)/2$
D) $n(n+1)/2$
E) $n$
Representing Graphs

Adjacency Matrix:

\[
G = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 0 & 1 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 \\
3 & 1 & 0 & 0 & 1 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 \\
5 & 0 & 1 & 0 & 1 & 0 \\
6 & 0 & 0 & 1 & 0 & 1 \\
\end{pmatrix}
\]
An Inefficient Representation

When is an adjacency matrix an inefficient way to store a graph?

How can we represent a graph without a lot of edges (or rather, in which the density of edges is low compared to the total number of vertices)?
Representing Graphs

Adjacency List:

```
1 → 2 → 3
2
3 → 1 → 4 → 6
4 → 1
5 → 3 → 5
6 → 2 → 4 → 5
```

The diagram illustrates a graph with vertices labeled 1 to 6 and edges connecting them. The adjacency list is shown to the left, and the graph is represented on the right with arrows indicating the connections between vertices.
DFS and BFS
Depth-First-Search (DFS) is a method to explore a graph in a systematic way. DFS has been popularized by R. Tarjan (Depth-first search and linear graph algorithms, *SIAM J. Comput.* 1 (1972), 146-160).

```pseudo
procedure DFS(G)
    t=0;
    for all u ∈ V do
        if color(u) = white then DFS-Visit;

procedure DFS-Visit(u)
    color(u) = gray;
    t++;
    d[u] = t;
    for all v ∈ V with (u,v) ∈ E do
        if color(v) = white then DFS-Visit(v);
    color(u) = black;
```

**Correctness.**
Find the loop invariant and prove the correctness of DFS(G).

**Time Analysis.**
Remember that $n = |V|$ and $m = |E|$. The running time of DFS(G) is $O(n+m)$.
Depth First Search

DFS( u )

1. Mark the „neighbor vertex“ v of vertex u „with the smallest number“ (if it hasn‘t been marked already).
2. Repeat Step 1 until all vertices have been marked.
BFS( u )

1. Mark all incident vertices v of vertex u „in lexicographic order“ from left to right (mark a vertex only if it hasn‘t been marked already).
2. Repeat Step 1 until all vertices have been marked.
Minimum Spanning Trees
Minimum Spanning Trees: Motivation

Imagine you have a business with several offices and you want to lease phone lines to connect them up with each other.
The phone company charges different amounts of money to connect different pairs of cities.
You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn’t a tree you can always remove some edges and save money.

In general, you can replace “phone lines” by something like electrical, hydraulic, TV cable, computer, road, water pipes and so on.

Minimum Spanning Trees play a crucial role in Network design.
Minimum Spanning Trees

Prim’s algorithm

![Diagram of a graph with nodes B, K, M, S, N and edges with weights 2, 3, 4, 5, 6, and 3. The algorithm starts at node B.]
Minimum Spanning Trees

Prim's algorithm

Diagram of a graph with weighted edges and a red path representing Prim's algorithm.
Minimum Spanning Trees

Prim's algorithm

2 + 3
Minimum Spanning Trees

Prim’s algorithm

2 + 3 + 3
Minimum Spanning Trees

Prim's algorithm

2+3+3+4
Minimum Spanning Trees

Prim’s algorithm

2 + 3 + 3 + 4 + 4 = 16
Minimum Spanning Trees

Prim’s algorithm

2 + 3 + 3 + 4 + 4 = 16
Minimum Spanning Trees: Literature

Computing a minimum spanning tree in an undirected network has been already described in 1926 by Boruvka who also gave a first solution.


Prim‘s and Kruskal‘s algorithms are examples of greedy algorithms (that decide only with the help of easy and local optimality criteria).
Minimum Spanning Tree: Running time analysis

Remember that $n=|V|$ and $m=|E|).

Prim‘s algorithm needs running time $O(n \log n + m)$ if the priority queue (that is needed to keep track of the nodes that are not yet included in the minimum spanning tree) is implemented with e.g. a Fibonacci-Heap.

Kruskal‘s algorithm needs running time $O(m \log m)$ (because the edges have to be sorted).

Which algorithm is better, i.e. needs less time?
Remember … $m \leq n(n-1)/2$

But it depends …
1. The number $m$ of edges (in reasonable problem settings) can range from about $n$ to $n(n-1)/2$. We must take that into consideration when we analyze the running times.
2. If the edge costs are not arbitrary but e.g. small integers that can be sorted in linear time, then Kruskal‘s algorithm needs only $O(n+ma(n, m/n))$, almost linear time.
Topological Sorting
Topological Sorting: Motivation

A must be finished before B starts and so on. Can this project be accomplished?
Topological Sorting: Motivation

A must be finished before B starts and so on. Can this project be accomplished?

Whenever there's a cycle, we can't find a prerequisite ordering.
Topological Sorting

Applications:
- Project Management
- Shortest Paths in Networks

Precondition:
- directed graphs that have no cycles are called directed acyclic graphs, or DAGs
- every DAG has a source
- G is a DAG if and only if G–v is a DAG

Topological Ordering:
- In a topological ordering, it must be true that for every edge v → w, v comes before w in the ordering.

A must be finished before B starts and so on.
Can this project be accomplished?

Whenever there's a cycle, we can't find a prerequisite ordering.
Topological Sorting

TopSort: Example

Activities:
A: Building Walls
B: Roof Timbering
C: Roof Tiling
D: Rendering inside
E: Rendering outside

Dependencies:
A before B, C, D, E
B before C, D, E
C before D, E
TopSort: Example

Activities:
A: Building Walls
B: Roof Timbering
C: Roof Tiling
D: Rendering inside
E: Rendering outside

Dependencies:
A before B, C, D, E
B before C, D, E
C before D, E
TopSort: Example

Activities:
A: Building Walls
B: Roof Timbering
C: Roof Tiling
D: Rendering inside
E: Rendering outside

Implementation:
• Maintain an integer array, InDegree[], where for each $i$ between 1 and $n$, InDegree[$i$] is the number of incoming edges to vertex $i$.
• Maintain a list of sources, $S$, either a stack or a queue.

Choose the first $x$ in $S$;
for(each $y$ adjacent to $x$)
    InDegree[$y$]--;
if(InDegree[$y$] == 0) add $y$ to $S;$
TopSort: Example

Activities:
A: Building Walls
B: Roof Timbering
C: Roof Tiling
D: Rendering inside
E: Rendering outside

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TopSort: Example

Activities:
A: Building Walls
B: Roof Timbering
C: Roof Tiling
D: Rendering inside
E: Rendering outside

A → B → C → {D, E}

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>-1 = 0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>-1 = 1</td>
<td>-1 = 0</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>-1 = 2</td>
<td>-1 = 1</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>-1 = 2</td>
<td>-1 = 1</td>
</tr>
</tbody>
</table>
TopSort: Example

Solution:
\[ \{A, G\} \rightarrow \{B, C, E\} \rightarrow \{D, H\} \rightarrow \{F, I\} \]