Algorithms and Data Structures

Graphs: Introduction

Ulf Leser

Content of this Lecture

- Graphs
- Definitions
- Representing Graphs
- Traversing Graphs
- Connected Components

Graphs

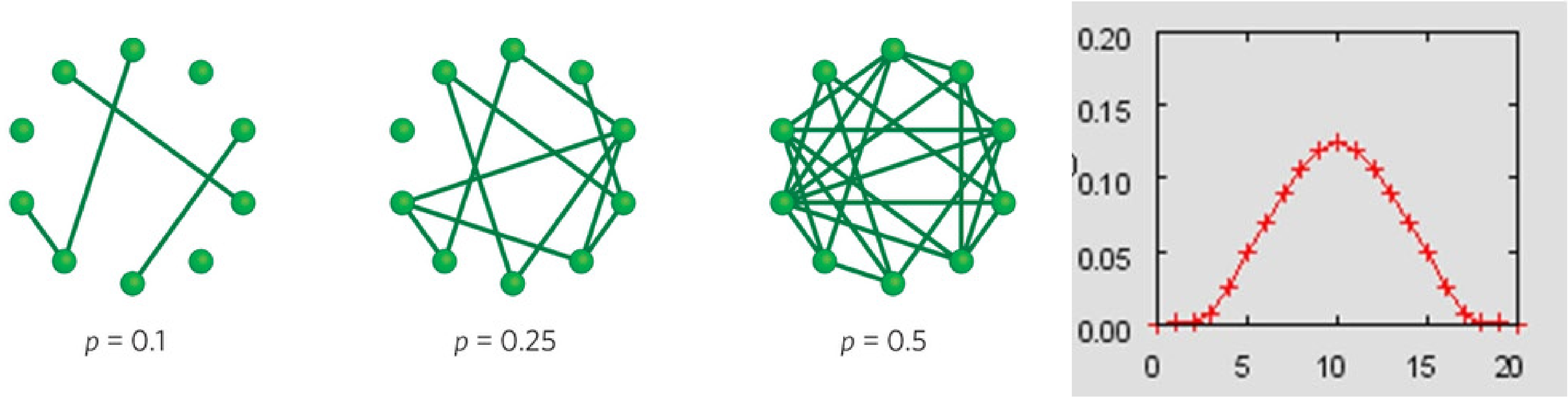
- There are objects and there are relations between objects
- Directed trees can represent hierarchical relations
  - Relations that are asymmetric, cycle-free, binary
  - Examples: parent_of, subclass_of, smaller_than, …
- Undirected trees can represent cycle-free, binary relations
- This excludes many (cyclic) real-life relations
  - friend_of, similar_to, reachable_by, html_linked_to, …
- (Classical) Graphs can represent all binary relationships
- N-ary relationships: Hypergraphs
  - exam(student, professor, subject), borrow(student, book, library)

Types of Graphs

- Most graphs you will see are binary
- Most graphs you will see are simple
  - Simple graphs: At most one edge between any two nodes
  - Extension: multigraphs
- Some graphs you will see are undirected, some directed
- This lecture: Binary, simple (finite) graphs

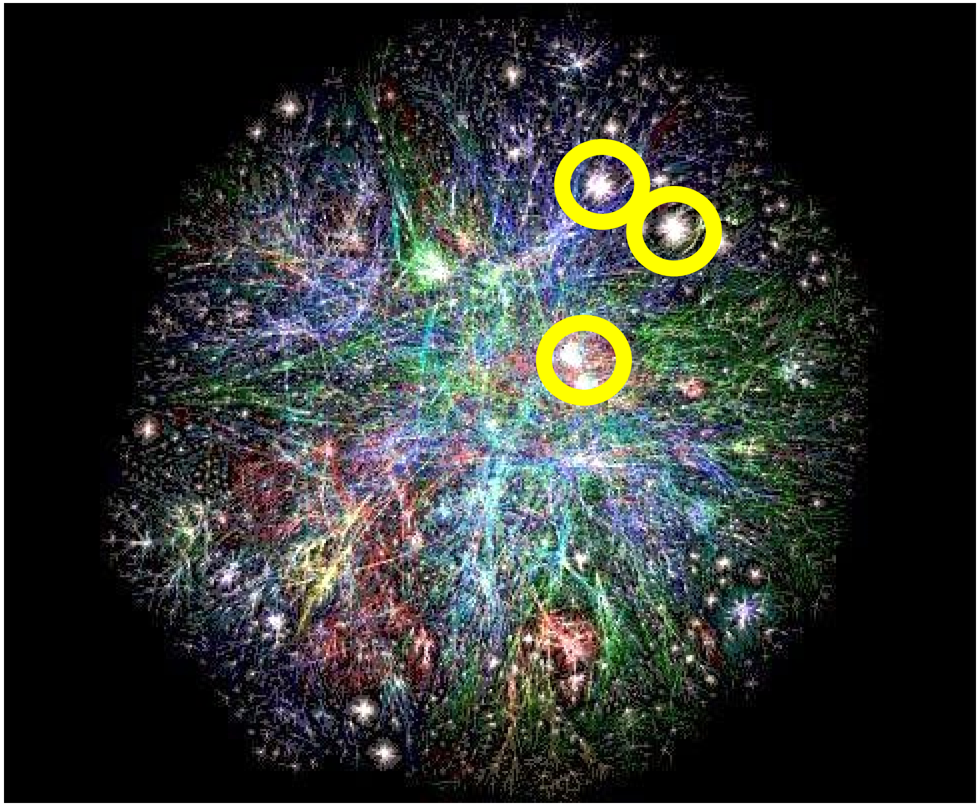
Exemplary Graphs

- Classical theoretical model: Random Graphs
  - Create every possible edge with a fixed probability p



  - In a random graph, the degree of every node has expected value p*n, and the degree distribution follows a Poisson distribution

Web Graph



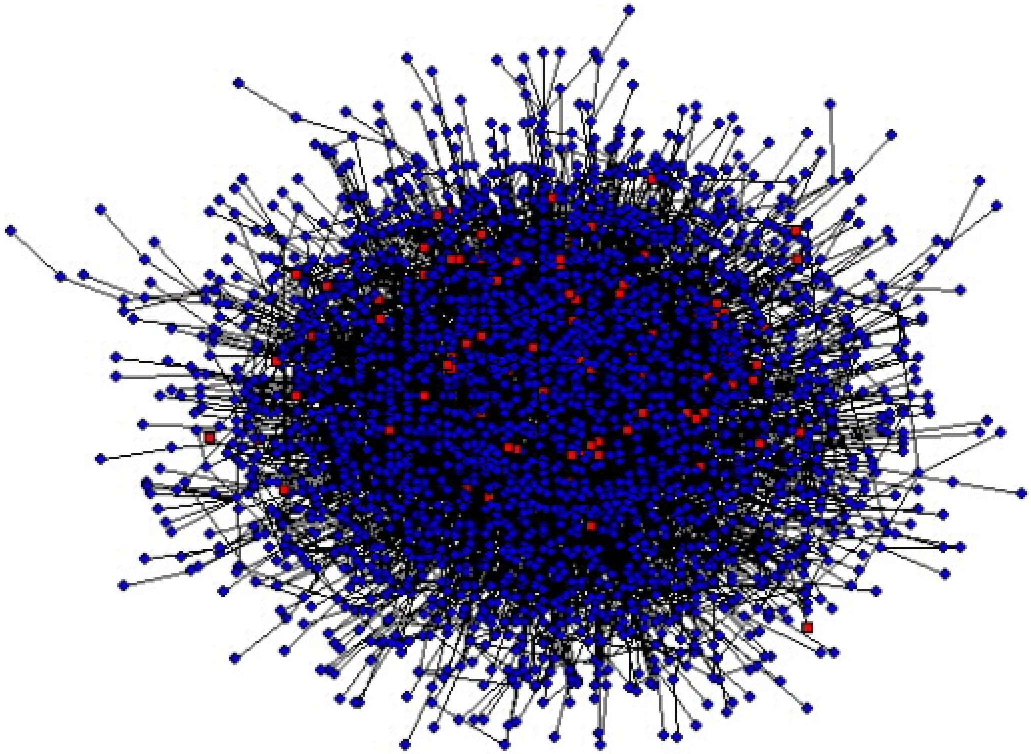
Note the strong local clustering

This is not a random graph

- Graph layout is difficult

[http://img.webme.com/pic/c/chegga-hp/opte_org.jpg]

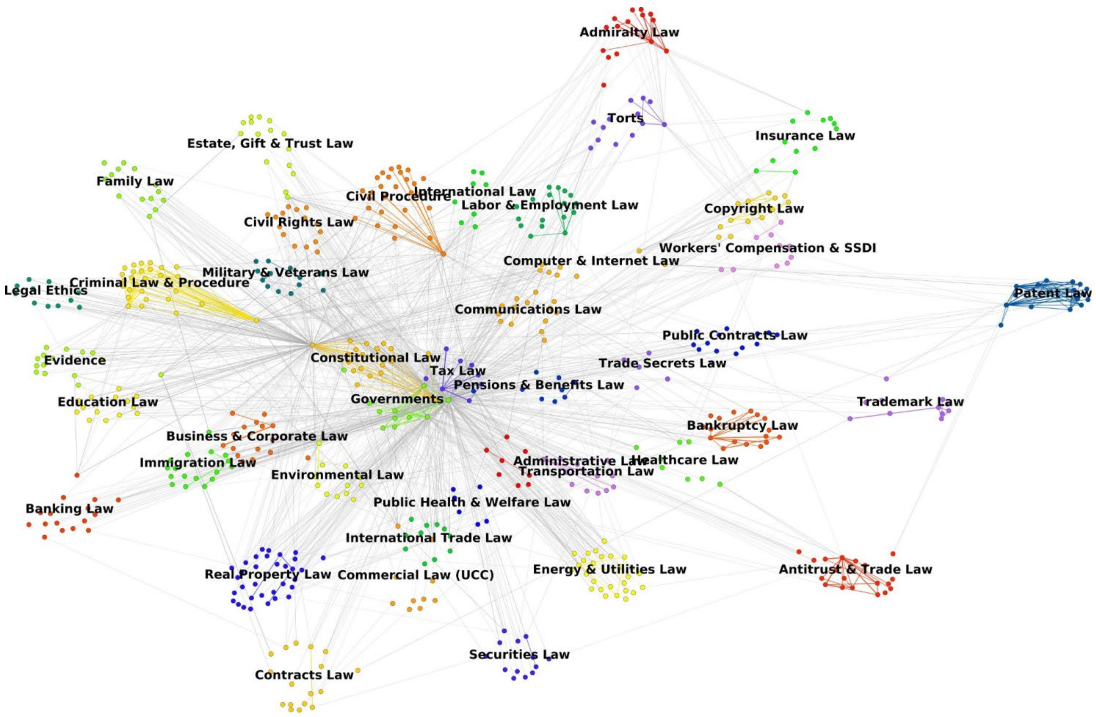
Human Protein-Protein-Interaction Network



- Still terribly incomplete
- Proteins that are close in the graph likely share function

[http://www.estradalab.org/research/index.html]

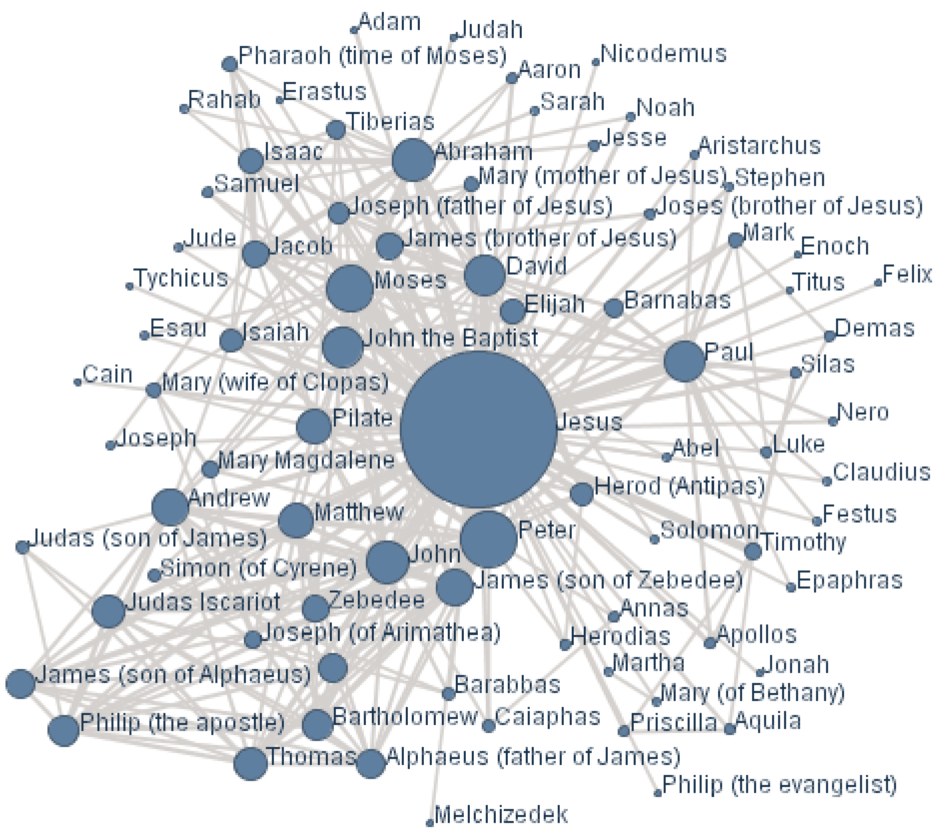
Word Co-Occurrence



- Words that are close have similar meaning
  - Close: Appear in the same contexts
- Words cluster into topics

[http://www.michaelbommarito.com/blog/]

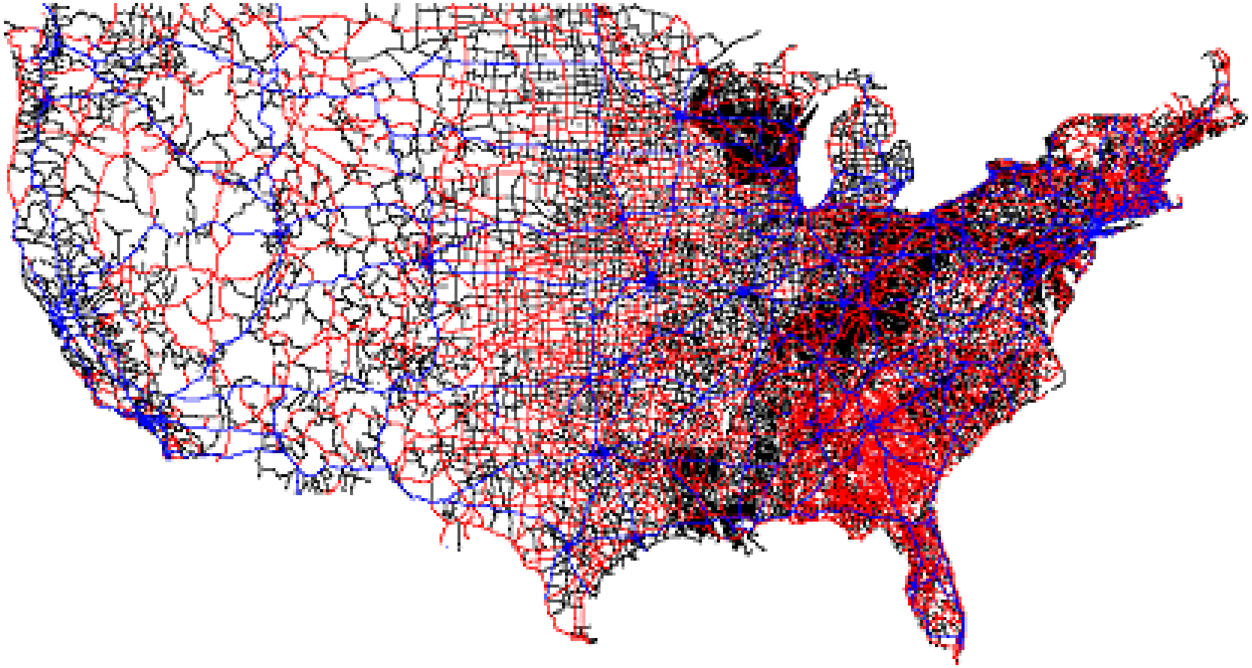
Social Networks



- Six degrees of separation

[http://tugll.tugraz.at/94426/files/-1/2461/2007.01.nt.social.network.png]

Road Network



- Specific property: Planar graphs

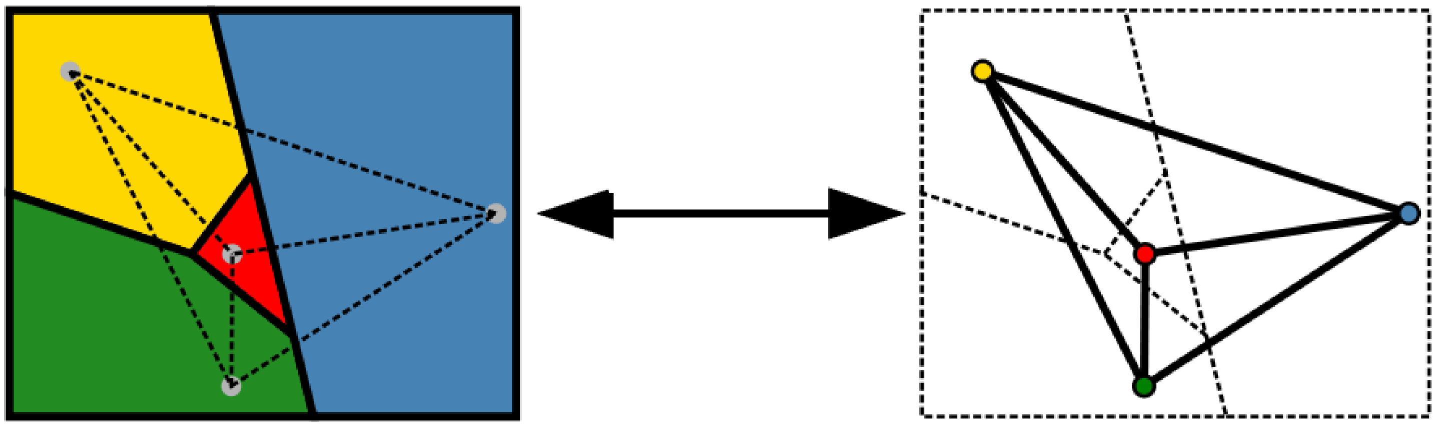
[Sanders, P. &Schultes, D. (2005).Highway Hierarchies Hasten Exact Shortest Path Queries. In *13th European Symposium on Algorithms (ESA), 568-579.*]

More Examples

- Graphs are also a wonderful abstraction

Coloring Problem

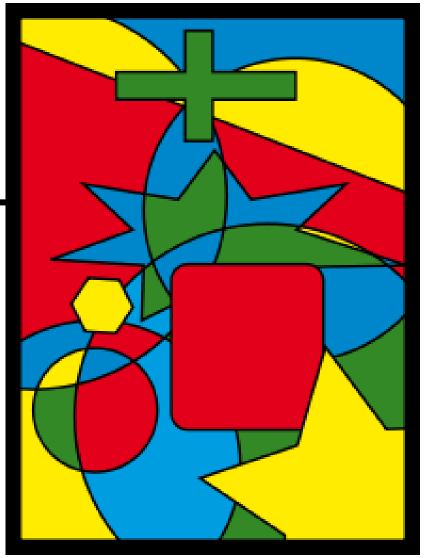
- How many colors do one need to color a map such that never two colors meet at a border?

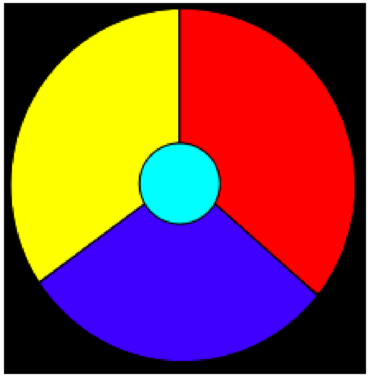


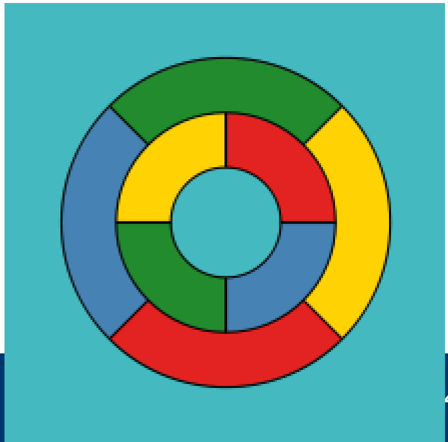
[http://www.wikipedia.de]

- Chromatic number: Number of colors sufficient to color a graph such that no adjacent nodes have the same color
- Every planar graph has chromatic number of at most 4

History [Wikipedia.de]



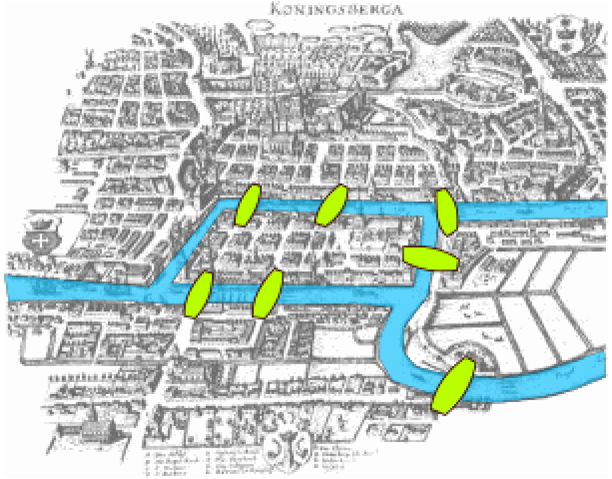




- This is not simple to proof
- It is easy to see that one sometimes needs at least four colors
- It is easy to show that one may need arbitrary many colors for general graphs
- First conjecture which until today was proven only by computers
  - Falls into many, many subcases – try all of them with a program

Königsberger Brückenproblem

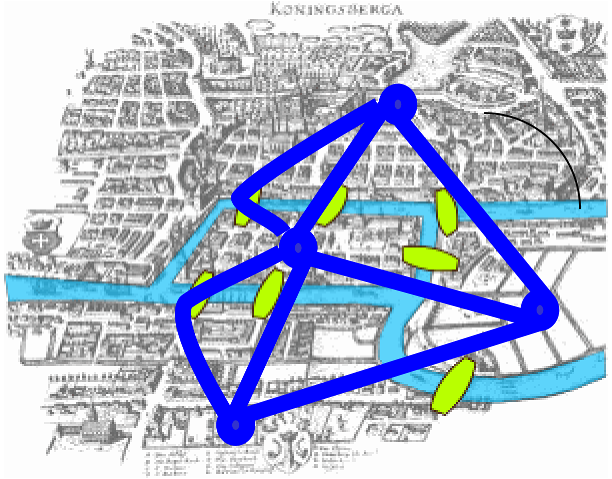
- Given a city with rivers and bridges: Is there a cycle-free path crossing every bridge exactly once?
  - Euler-Path



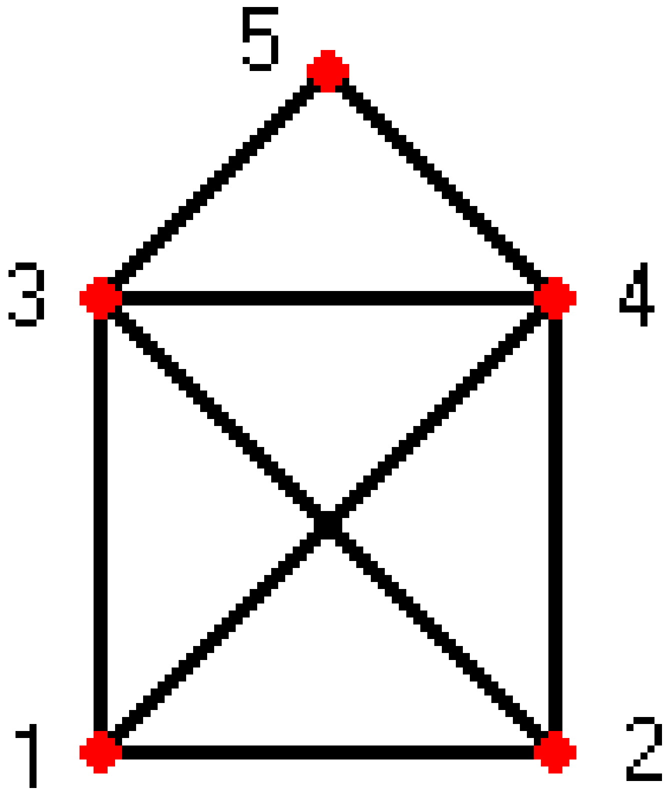
Source: Wikipedia.de

Königsberger Brückenproblem

- Given a city with rivers and bridges: Is there a cycle-free path crossing every bridge exactly once?
  - A graph has an Euler-Path iff at contains 0 or 2 nodes with odd degree
- Hamiltonian path
  - … visits each vertex exactly once
  - NP complete



Recall?



Content of this Lecture

- Graphs
- Definitions
- Representing Graphs
- Traversing Graphs
- Connected Components

Recall from Trees

- Definition

  A graph $G=(V, E)$ consists of a set of vertices (nodes) V and a set of edges ($E \subseteq VxV$).

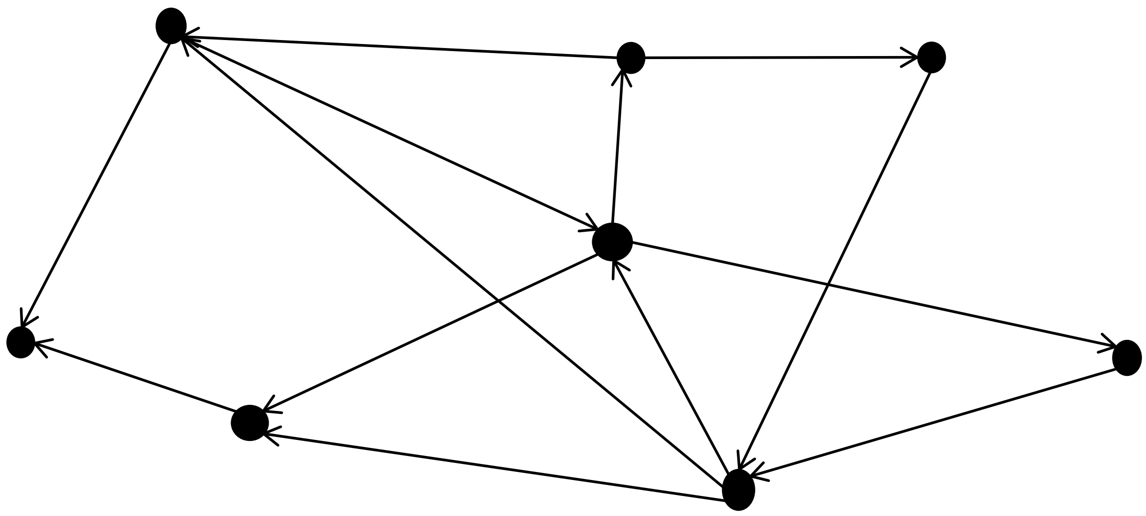
  - *A sequence of edges $e_1, e_2, .., e_n$ is called a path iff $\forall 1 \leq i < n$: $e_i=(v', v)$ and $e_{i+1}=(v, v``)$; the length of this path is n*
  - *A path $(v_1,v_2), (v_2,v_3), \ldots, (v_{n-1},v_n)$ is acyclic iff all v_i are different*
  - *G is acyclic, if no path in G contains a cycle; otherwise it is cyclic*
  - *A graph is connected if every pair of vertices is connected by at least one path*
  - *G is called undirected, if $\forall (v,v') \in E \Rightarrow (v',v) \in E$. Otherwise it is called directed.*

More Definitions

- Definition
  Let G=(V, E) be a directed graph. Let $v \in V$
  - *The outdegree out(v) is the number of edges with v as start point*
  - *The indegree in(v) is the number of edges with v as end point*
  - *G is edge-labeled, if there is a function $w:E \rightarrow L$ that assigns an element of a set of labels L to every edge*
  - *If L are numbers (real, int, …), G is called weighted*
- Remarks
  - Labels / weights max be assigned to edges or nodes (or both)
  - Indegree and outdegree are identical for undirected graphs and called degree (number of neighbors)

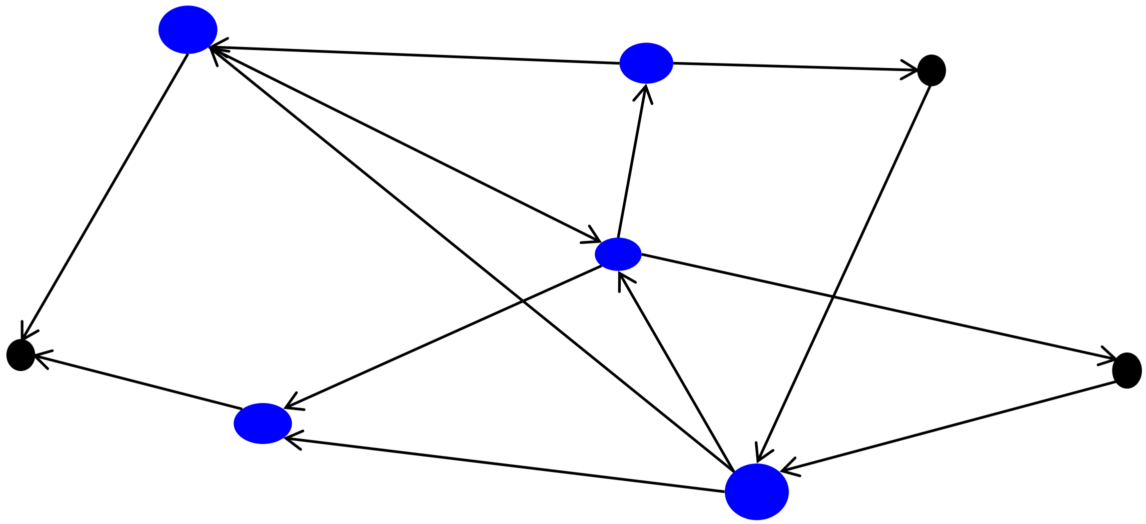
Some More Definitions

- Definition. *Let G=(V, E) be a directed graph.*
  - *Any G'=(V', E') is called a subgraph of G, if $V' \subseteq V$ and $E' \subseteq E$ and $\forall (v_1,v_2) \in E'$: $v_1,v_2 \in V'$*
  - *For any $V' \subseteq V$, the graph $(V', E \cap (V' \times V'))$ is called the induced subgraph of G (induced by V')*



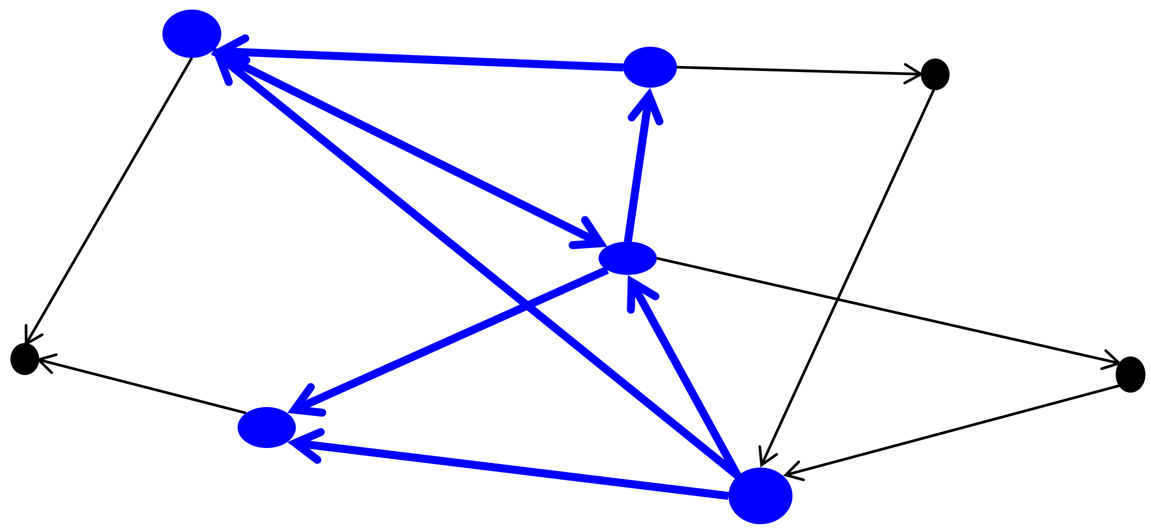
Some More Definitions

- Definition. *Let G=(V, E) be a directed graph.*
  - *Any G'=(V', E') is called a subgraph of G, if $V' \subseteq V$ and $E' \subseteq E$ and $\forall (v_1,v_2) \in E'$: $v_1,v_2 \in V'$*
  - *For any $V' \subseteq V$, the graph $(V', E \cap (V' \times V'))$ is called the induced subgraph of G (induced by V')*



Some More Definitions

- Definition. *Let G=(V, E) be a directed graph.*
  - *Any G'=(V', E') is called a subgraph of G, if $V' \subseteq V$ and $E' \subseteq E$ and $\forall (v_1,v_2) \in E'$: $v_1,v_2 \in V'$*
  - *For any $V' \subseteq V$, the graph $(V', E \cap (V' \times V'))$ is called the induced subgraph of G (induced by V')*



Content of this Lecture

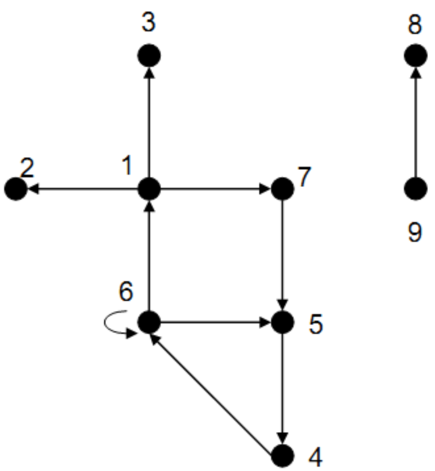
- Graphs
- Definitions
- Representing Graphs
- Traversing Graphs
- Connected Components

Data Structures

- From an abstract point of view, a graph is a list of nodes and a list of (weighted, directed) edges
- Two fundamental implementations
  - Adjacency matrix
  - Adjacency lists
- As usual, the representation determines the complexity of primitive operations
  - E.g. find node, find edge, find neighbors, …
- Suitability depends on the specific problem under study and the nature of the graphs
  - Shortest paths, transitive hull, cliques, spanning trees, …
  - Random, sparse/dense, scale-free, planar, …

Example [OW93]

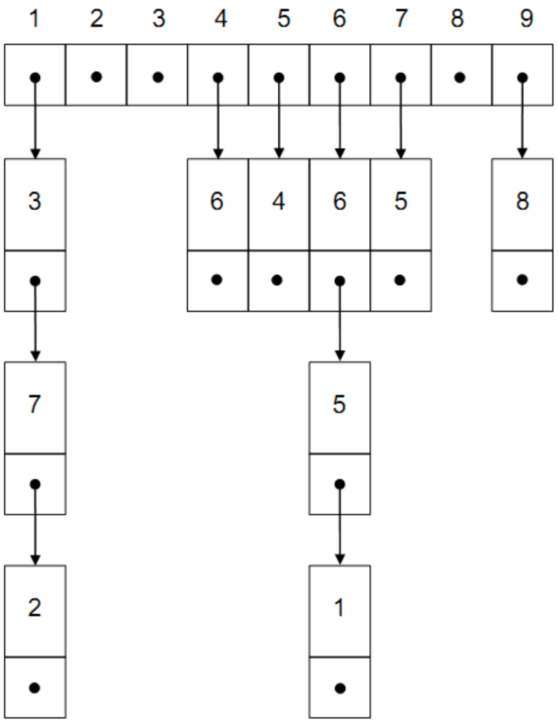
Graph



Adjacency Matrix

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0
5	0	0	0	1	0	0	0	0	0
6	1	0	0	0	1	1	0	0	0
7	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	1	0

Adjacency List



Adjacency Matrix

- Definition
  Let G=(V, E) be a simple graph. The adjacency matrix M_G for G is a two-dimensional matrix of size |V||V|, where $M[i,j]=1$ iff $(v_i, v_j) \in E$*
- Remarks
  - Allows to test existence of a given edge in O(1)
  - Requires O(|V|) to obtain all incoming (outgoing) edges of a node
  - For large graphs, M is too large to be of practical use
  - If G is sparse (much less edges than $|V|^2$), M wastes a lot of space
  - If G is dense, M is a very compact representation (1 bit / edge)
  - In labeled graphs, M[i,j] contains the label
  - Since M must be initialized with zero's, without further tricks all algorithms working on adjacency matrices are in $\Omega(|V|^2)$

Adjacency List

- Definition
  Let G=(V, E). The adjacency list L_G for G is a list of all nodes v_i of G. The entry representing $v_i \in V$ is a list of all edges outgoing (or incoming or both) from v_i.
- Remarks (assume a fixed node v)
  - Let k be the maximal outdegree of G. Then, accessing an edge outgoing from v is O(log(k)) (if list is sorted; or use hashing)
  - Obtaining a list of all outgoing edges from v is in O(k)
    - If only outgoing edges are stored, obtaining a list of all incoming edges is O(|V|*log(|E|)) – we need to search all lists
    - Therefore, usually outgoing and incoming edges are stored, which doubles space consumption
  - If G is sparse, L is a compact representation
  - If G is dense, L is wasteful (many pointers, many IDs)

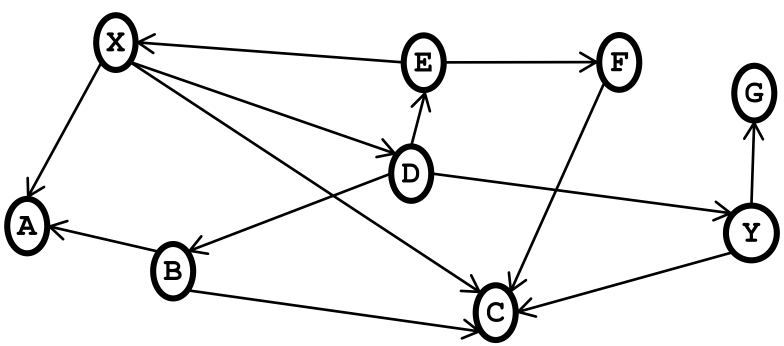
Comparison

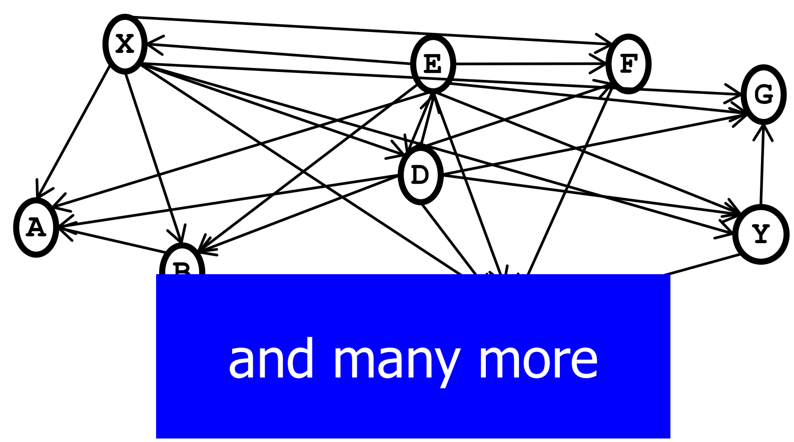
	Matrix	**Lists**
Test if a given edge exists	O(1)	O(log(k))
Find all outgoing edges of a given v	O(n)	O(k)
Space of G	$O(n^2)$	O(n+m)

- With n=\|V\|, m=\|E\|
- We assume a node-indexed array
  - L is an array and nodes are uniquely numbered
  - We find the list for node v in O(1)
  - Otherwise, L has additional costs for finding v

Transitive Closure

- Definition
  Let G=(V,E) be a digraph and $v_i, v_j \in V$. The ***transitive closure*** *of G is a graph G'=(V, E') where $(v_i, v_j) \in E'$ iff G contains a path from v_i to v_j.*
- TC usually is dense and represented as adjacency matrix
- Compact encoding of reachability information





Content of this Lecture

- Graphs
- Definitions
- Representing Graphs
- Traversing Graphs
- Connected Components

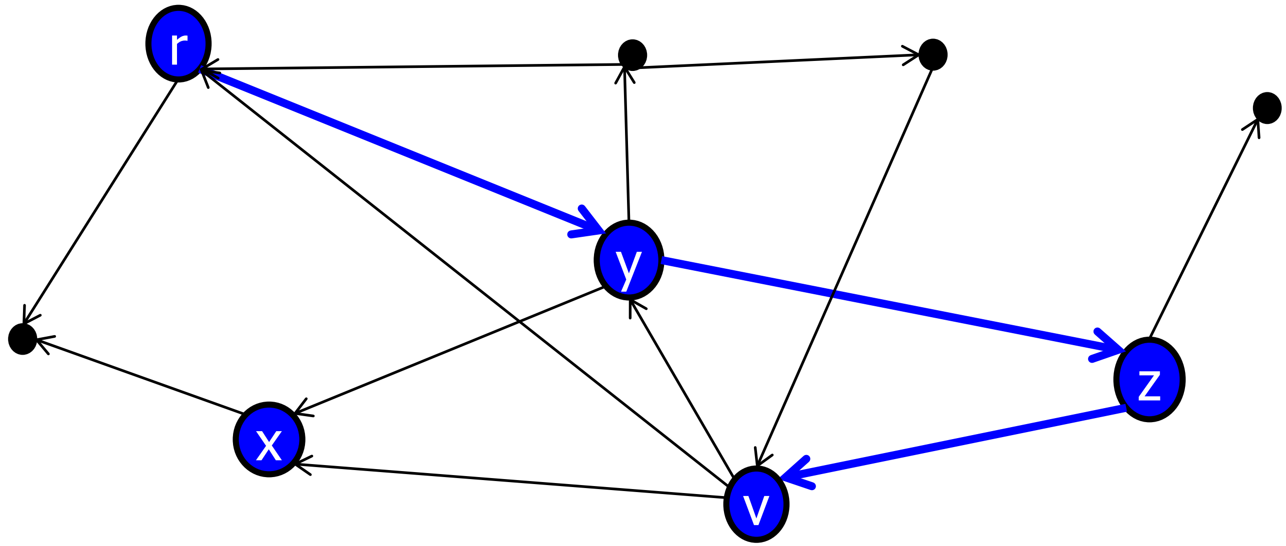
Graph Traversal

- One thing we often do with graphs is traversal
- "Traversal" means: Visit every node exactly once in a sequence determined by the graph's topology
  - Not necessarily on one consecutive path (Hamiltonian path)
- Two popular orders
  - Depth-first: Using a stack
  - Breadth-first: Using a queue
  - The scheme is identical to that in tree traversal
- Difference
  - We have to take care of cycles
  - No root – where should we start?

Breaking Cycles

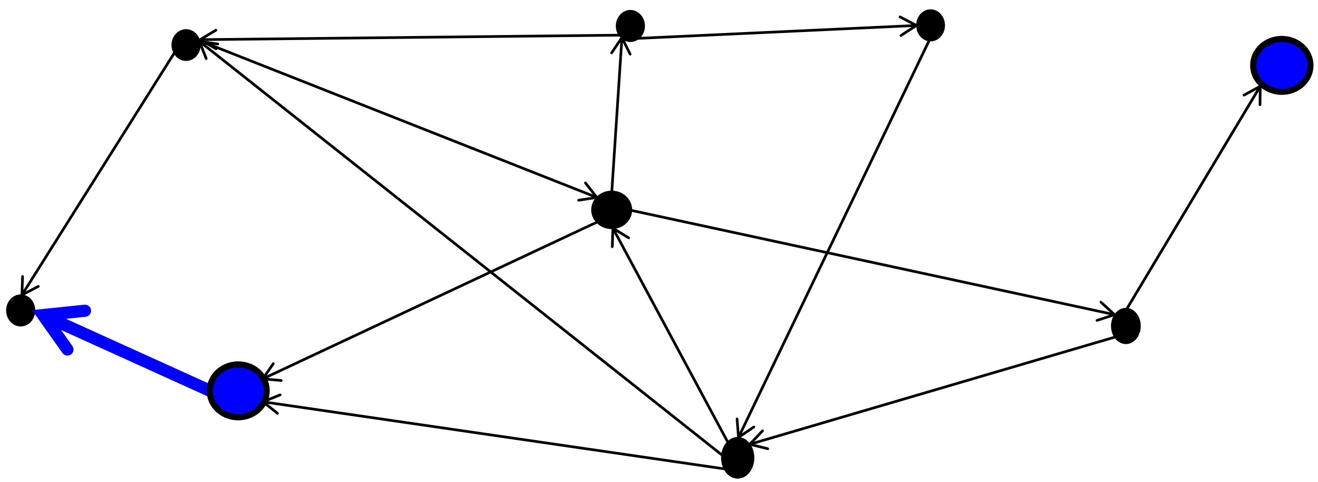
- Any naïve traversal will visit nodes more than once
  - If there is at least one node with more than one incoming edge
- Any naïve traversal will run into infinite loops
  - If the graphs contains at least one cycle (is cyclic)
- Breaking cycles / avoiding multiple visits
  - Assume we started the traversal at a node r
  - During traversal, we keep a list S of already visited nodes
  - Assume we are in v and aim to proceed to v' using e=(v, v')∈E
  - If v'∈S, v' was visited before and we are about to run into a cycle or visit v' twice
  - In this case, e is ignored

Example



- Started at r and went S={r, y, z, v}
- Testing (v,y): y∈S, drop
- Testing (v, r): r∈S, drop
- Testing (v, x): x∉S, proceed

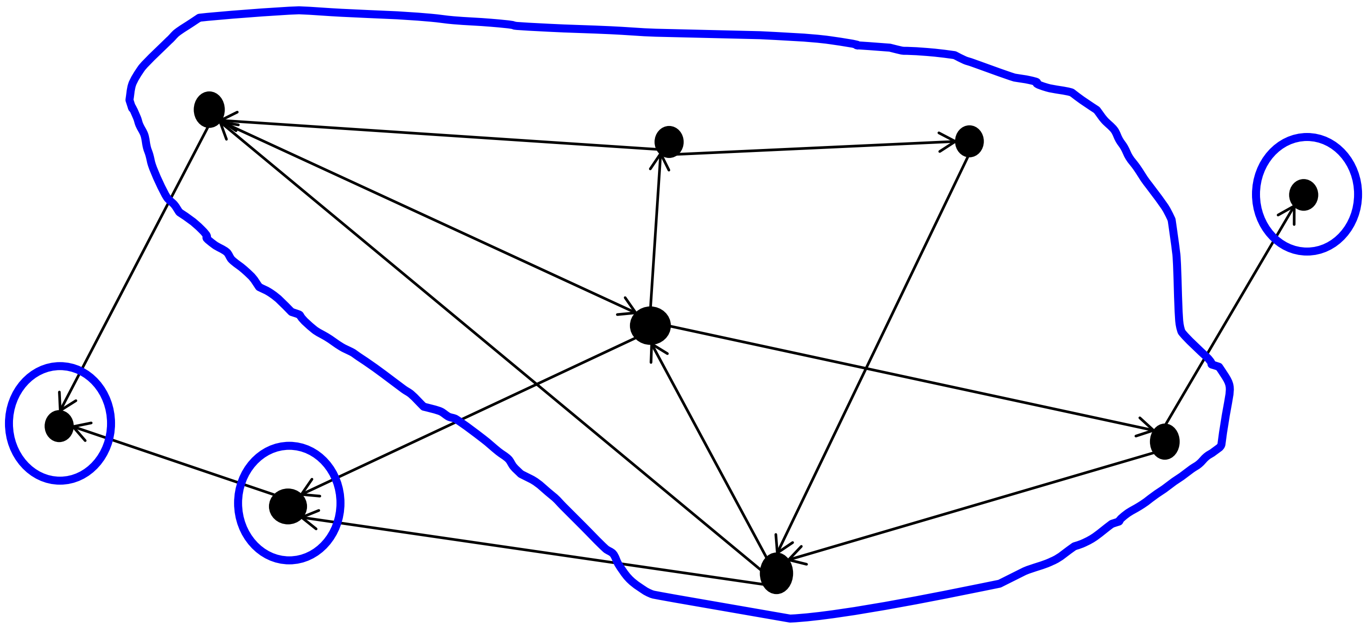
Where do we Start?



Where do we Start?

- Definition
  Let G=(V, E). Let V′⊆V and G′ be the subgraph of G induced by V′
  - *G′ is called connected if it contains a path between any pair v,v′∈V′*
  - *G′ is called maximally connected, if no subgraph induced by a superset of V′ is connected*
  - *If G is undirected, any maximal connected subgraph of G is called a connected component of G*
  - *If G is directed, any maximal connected subgraph of G is called a strongly connected component of G*

Example



Where do we Start?

- If a undirected graph falls into several connected components, we cannot reach all nodes by a single traversal, no matter which node we use as start point
- If a digraph falls into several strongly connected components, we might not reach all nodes by a single traversal
- Remedy: If the traversal gets stuck, we restart at unseen nodes until all nodes have been traversed

Depth-First Traversal on Directed Graphs

```
func void DFS (G=(V,E)) {
  U := V;       # Unseen nodes
  while U≠∅ do
    v := getNextUnseen( U);
    traverse( G, v, U);
  end while;
}
```

Called once for every connected component

```
func void traverse (G, v node,
                    U set) {
  t := new Stack();
  t.put( v);
  U := U \ {v};
  while not t.isEmpty() do
    n := t.pop();
    print n;
    c := n.outgoingNodes();
    foreach x in c do
      if x∈U then
        U := U \ {x};
        t.push( x);
      end if;
    end for;
  end while;
}
```

Analysis

- We put every node exactly once on the stack
  - Once visited, never visited again
- We look at every edge exactly once
  - Outgoing edges of a visited node are never considered again
- U can be implemented as bit-array of size |V|, allowing O(1) operations
  - Add, remove, getNextUnseen
- Altogether: O(n+m)

```
func void traverse (G, v node,
                    U set) {
  t := new Stack();
  t.put( v);
  U := U \ {v};
  while not t.isEmpty() do
    n := t.pop();
    print n;
    c := n.outgoingNodes();
    foreach x in c do
      if x∈U then
        U := U \ {x};
        t.push( x);
      end if;
    end for;
  end while;
}
```

Content of this Lecture

- Graphs
- Definitions
- Representing Graphs
- Traversing Graphs
- Connected Components

In Undirected Graphs

- In an undirected graph, whenever there is a path from r to v and from v to v', then there is also a path from v' to r
  - Simply go the path r $\rightarrow$ v $\rightarrow$ v' backwards
- Thus, DFS (and BFS) traversal can be used to find all connected components of a undirected graph G
  - Whenever you call traverse(v), create a new component
  - All nodes visited during one call of traverse(v) form one connected component
- Obviously in O(n+m)

In Digraphs

- The problem is considerably more complicated for digraphs
  - Previous conjecture does not hold
- Still: Tarjan‘s or Kosaraju’s algorithm find all strongly connected components in O(n + m)
  - See next lecture

Possible Examination Questions

- Let G be an undirected graph and S,T be two connected components of G. Proof that S and T must be disjoint, i.e., cannot share a node.
- Let G be an undirected graph with n vertices and m edges, $m<=n^2$. What is the minimal and what is the maximal number of connected components G can have?
- Let G be a positively edge-weighted digraph G. Design an algorithm which finds the longest acyclic path in G. Analyze the complexity of your algorithm.
- An Euler path through an undirected graph G is a cycle-free path from any start to any end node that hits every node of G (exactly once). Give an algorithm which tests for an input graph G whether it contains an Euler path.