Algorithms and Data Structures

Graphs: Introduction and First Algorithms

Ulf Leser

This Course

• Introduction	2
• Abstract Data Types	1
• Complexity analysis	1
• Styles of algorithms	1
• Lists, stacks, queues	2
• Sorting (lists)	3
• Searching (in lists, PQs, SOL)	5
• Hashing (to manage lists)	2
• Trees (to manage lists)	4
• Graphs (no lists!)	5
• Sum	**21/26**

Content of this Lecture

- Graphs
- Definitions
- Representing Graphs
- Traversing Graphs
- Connected Components
- Shortest Paths

Graphs

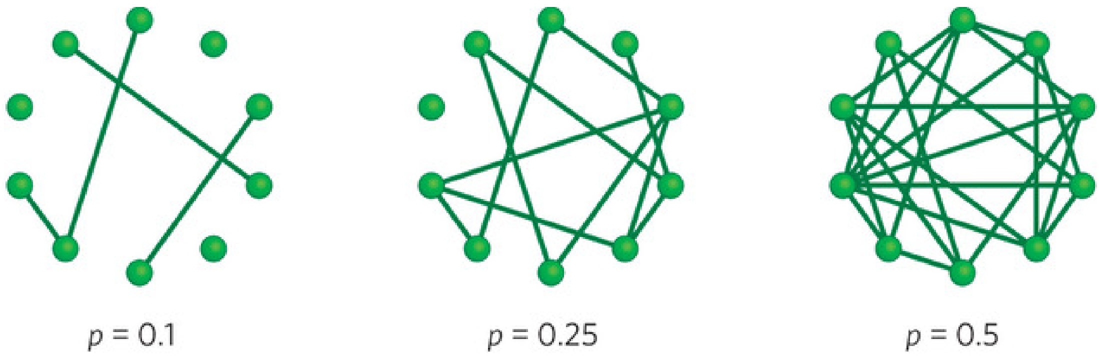
- There are objects and there are relations between objects
- Directed trees can represent hierarchical relations
  - Relations that are asymmetric, cycle-free, binary
  - Examples: parent_of, subclass_of, smaller_than, ...
- Undirected trees can represent cycle-free, binary relations
- This excludes 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)

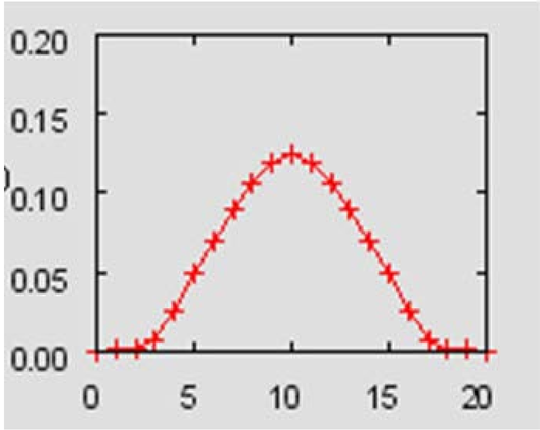
Types of Graphs

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

Exemplary Graphs

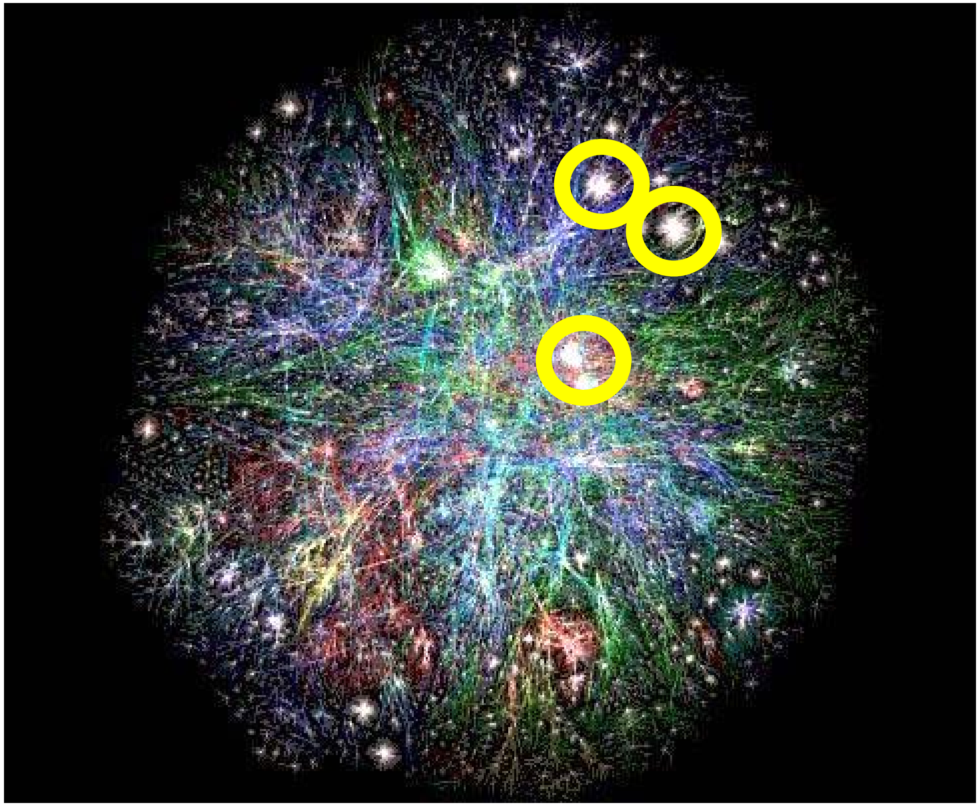
- Classical theoretical model: Random Graphs
  - Create every possible edge with a fixed probability p





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

Web Graph



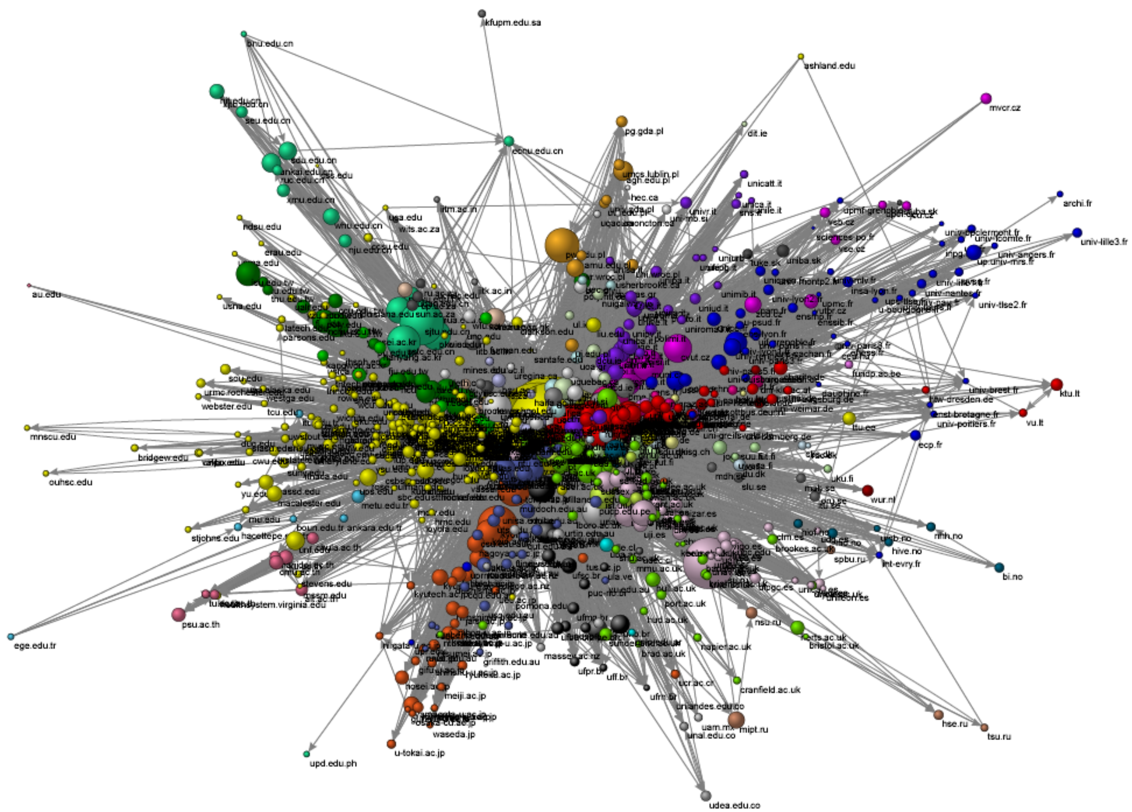
Note the strong local clustering

This is not a random graph

- Graph layout is difficult

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

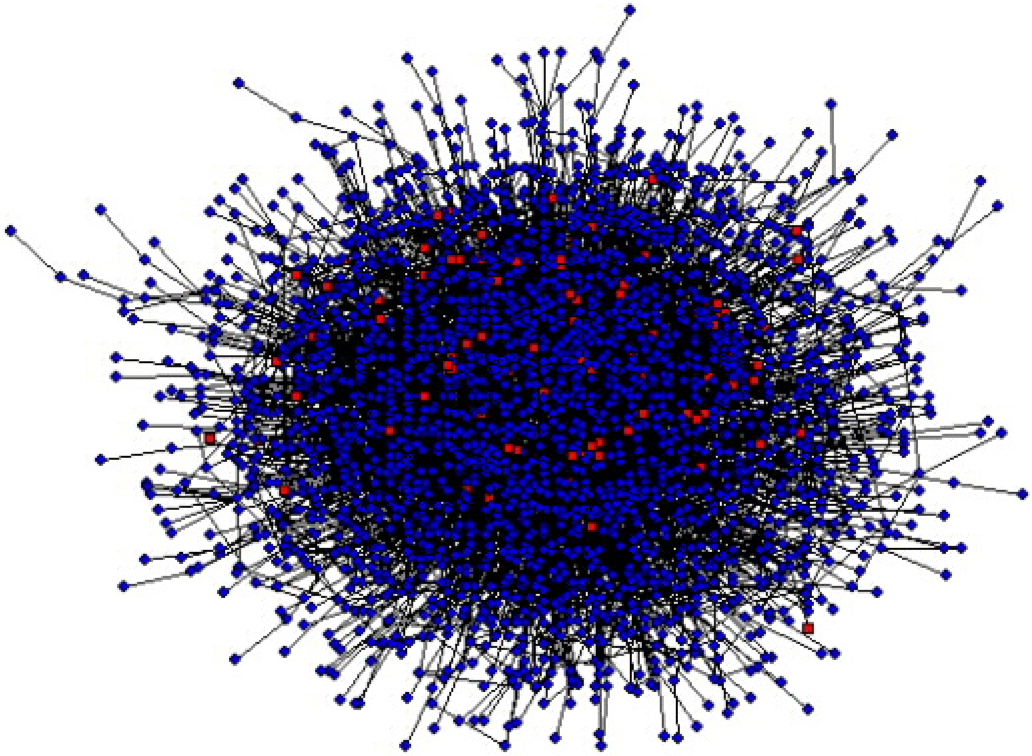
Universities Linking to Universities



- Small-World Property

[http://internetlab.cindoc.csic.es/cv/11/world_map/map.html]

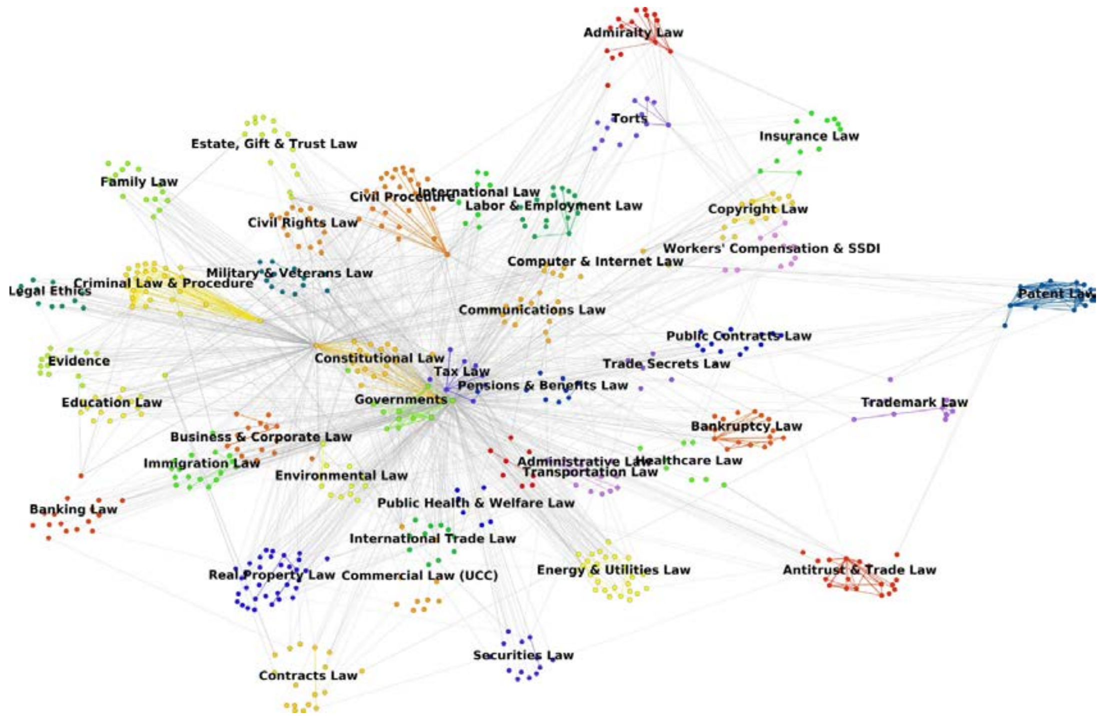
Human Protein-Protein-Interaction Network



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

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

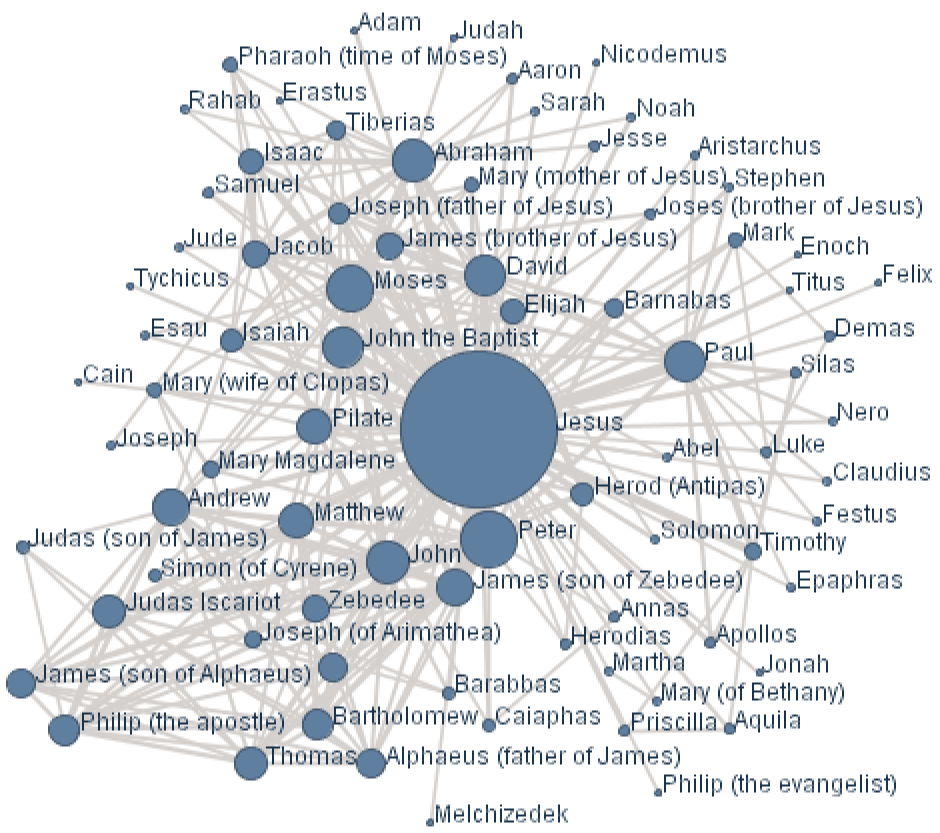
Word Co-Occurrence



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

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

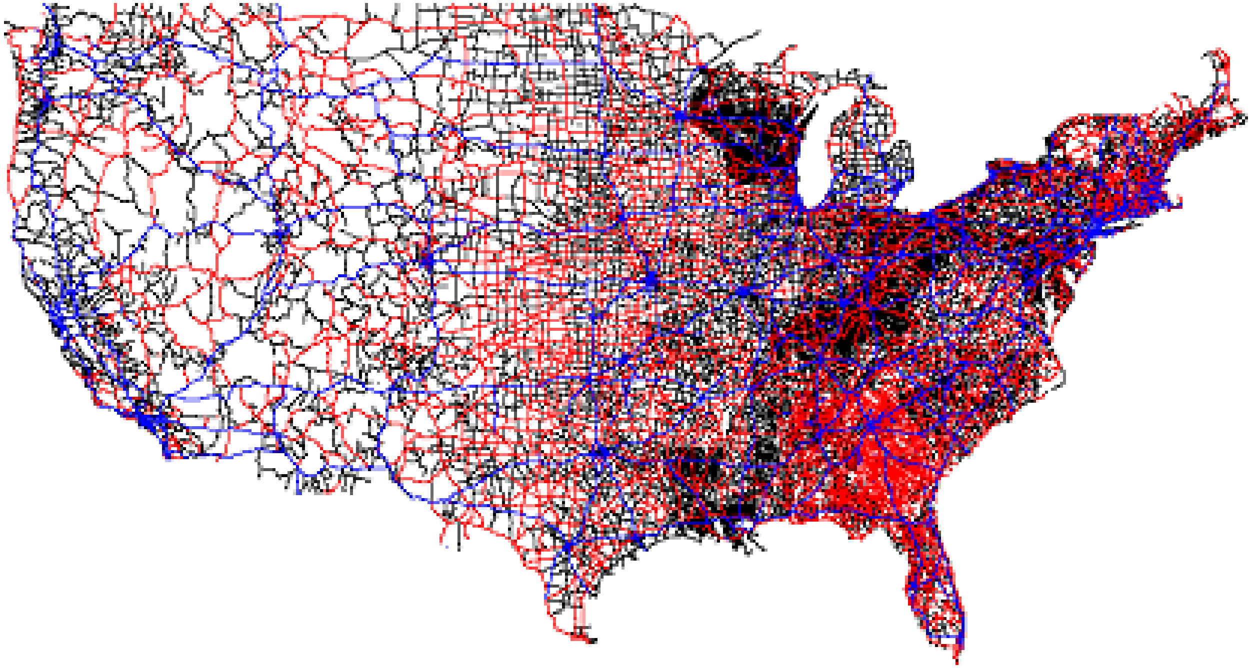
Social Networks



- Six degrees of separation

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

Road Network



- Specific property: Planar graphs

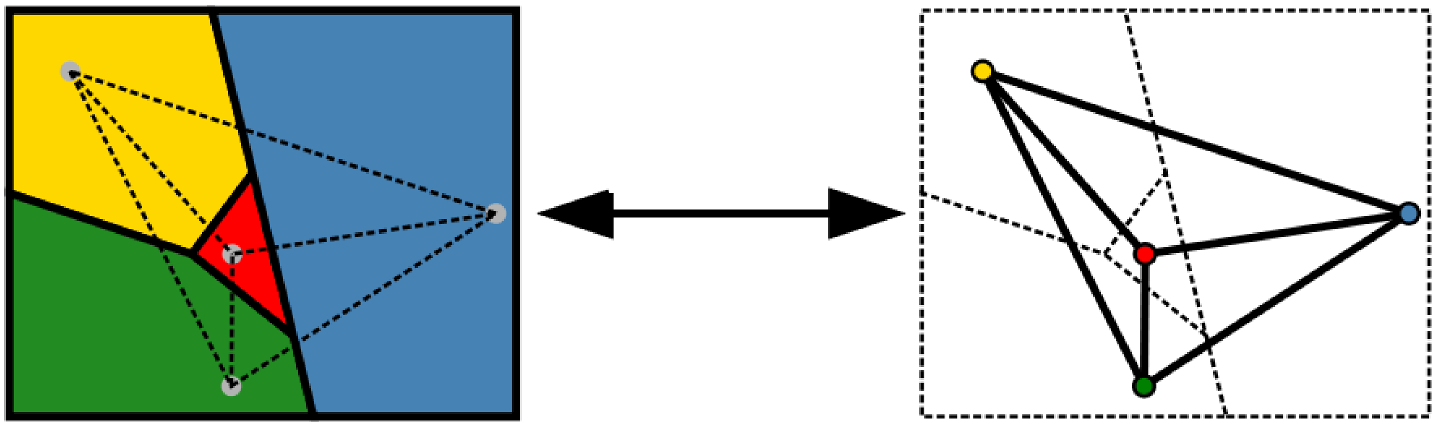
[Sanders, P. &Schultes, D. (2005).Highway Hierarchies Hasten Exact Shortest Path Queries. In *13th European Symposium on Algorithms (ESA), 568-579.*]

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?

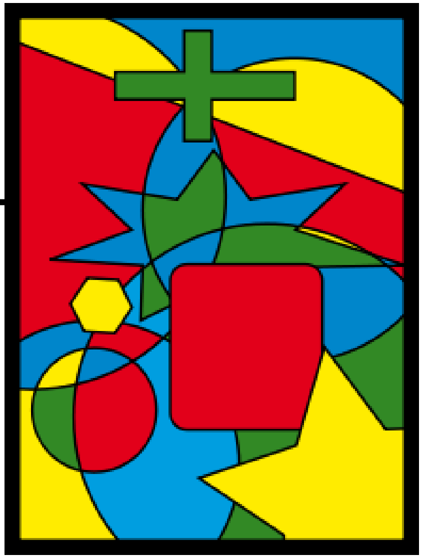


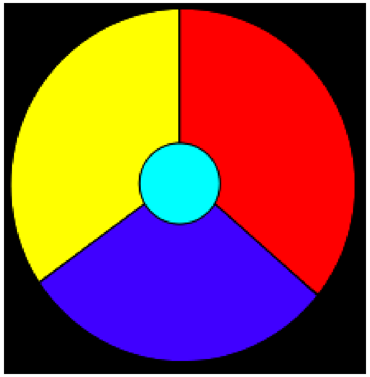
[http://www.wikipedia.de]

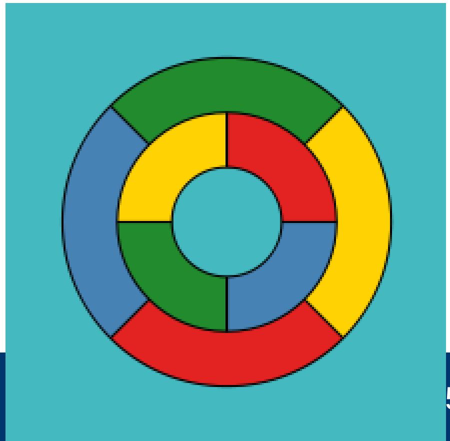
- Chromatic number: Number of colors sufficient to color a graph such that no adjacent nodes have the same color
- Every planar graph has chromatic number of at most 4

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

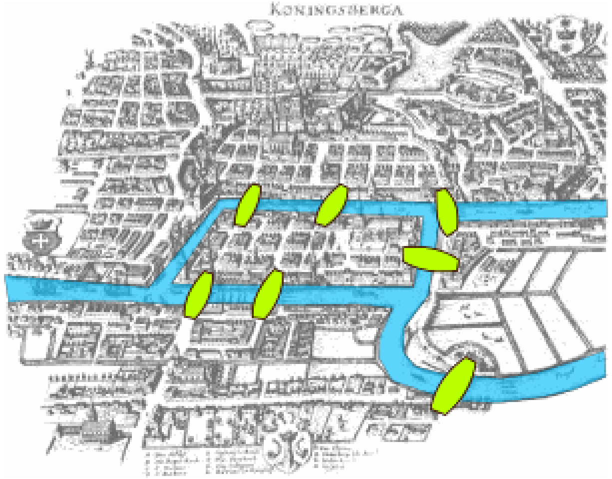






Königsberger Brückenproblem

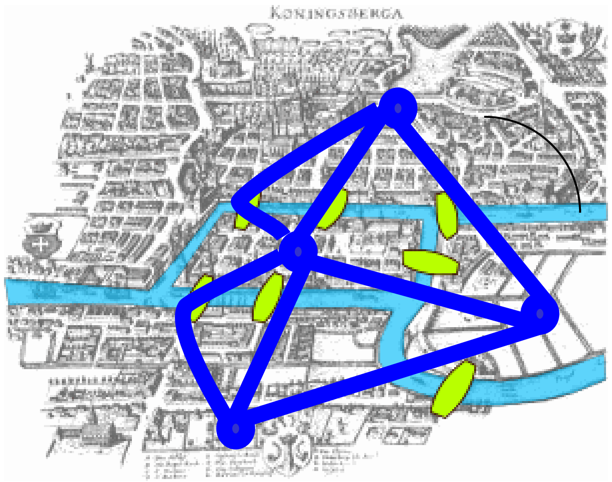
- Given a city with rivers and bridges: Is there a cycle-free path crossing every bridge exactly once?
  - Euler-Path



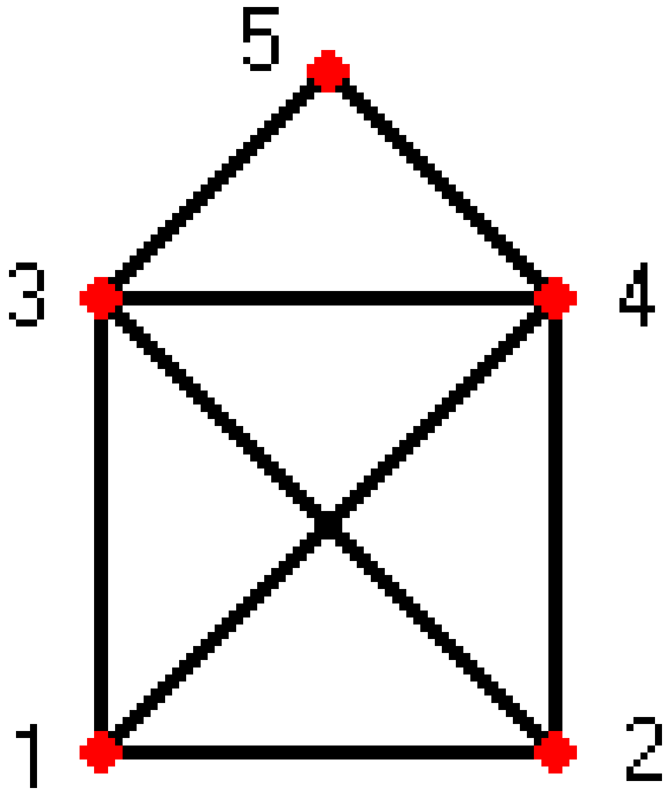
Source: Wikipedia.de

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 edges 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
- Shortest Paths

Recall from Trees

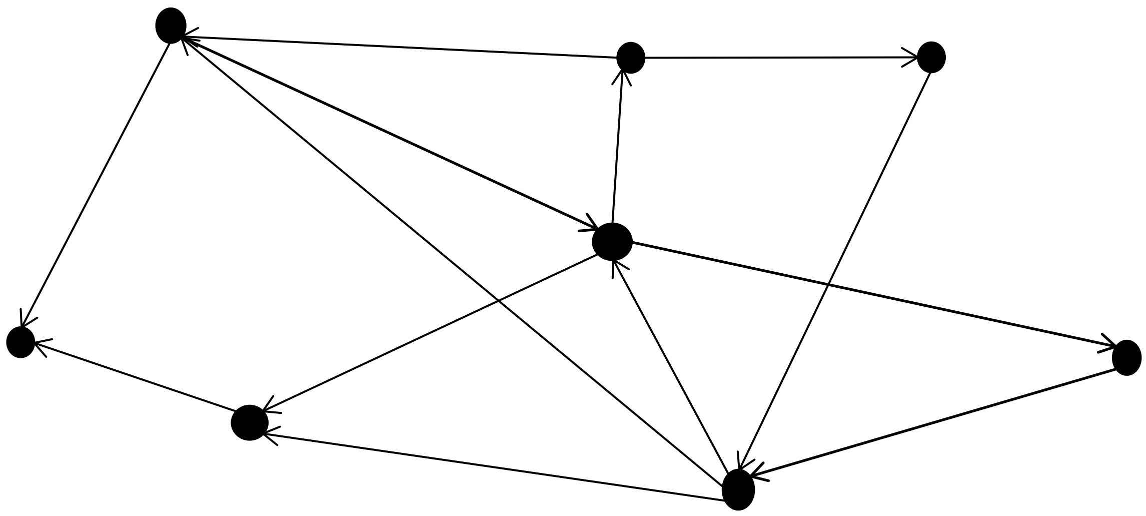
- Definition
  A graph $G=(V, E)$ consists of a set of vertices (nodes) V and a set of edges ($E \subseteq V x V$).
  - *A sequence of edges $e_1, e_2, .., e_n$ is called a path iff $\forall 1 \leq i<n$: $e_i=(v', v)$ and $e_{i+1}=(v, v``)$; the length of this path is n*
  - *A path $(v_1,v_2), (v_2,v_3), ..., (v_{n-1},v_n)$ 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*
- Definition
  A graph (tree) 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*
  – *A labeled graph with $L=\mathbb{N}$ is called weighted*
- Remarks
  – Weights can as well be reals; often we only allow positive weights
  – Labels / weights max be assigned to edges or nodes (or both)
  – Indegree and outdegree are identical for undirected graphs

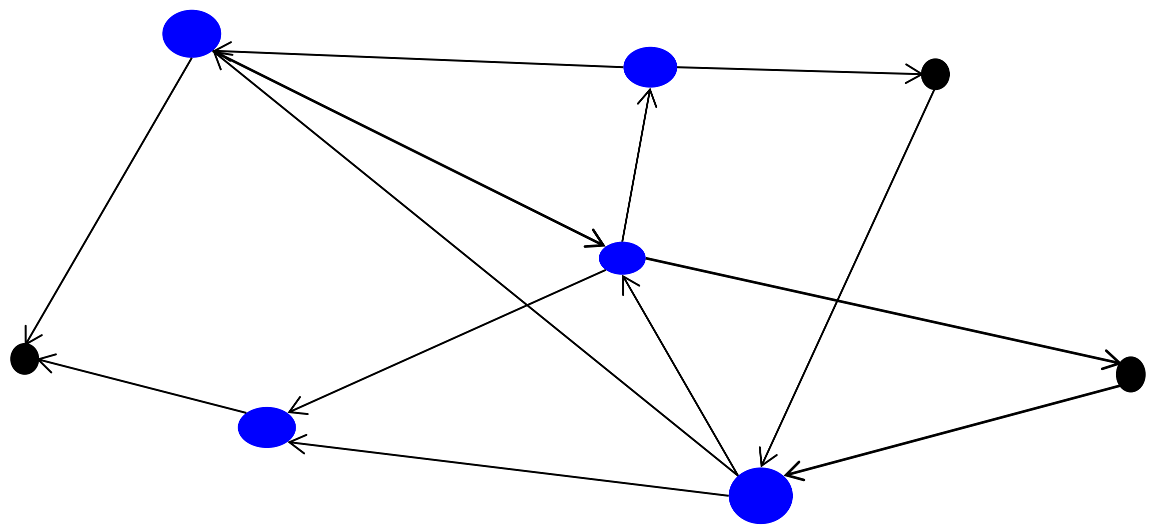
Some More Definitions

- 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 for all $(v_1,v_2) \in E'$: $v_1,v_2 \in V'$*
  - *For any $V' \subseteq V$, the graph $(V', E \cap (V' \times V'))$ is called the induced subgraph of G (induced by V')*



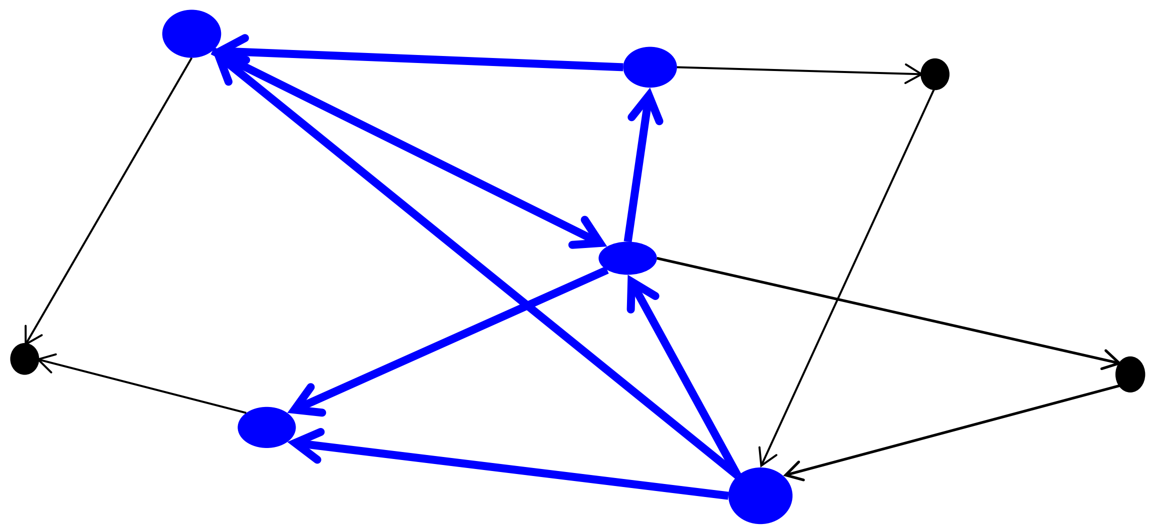
Some More Definitions

- 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 for all $(v_1,v_2) \in E'$: $v_1,v_2 \in V'$*
  - *For any $V' \subseteq V$, the graph $(V', E \cap (V' \times V'))$ is called the induced subgraph of G (induced by V')*



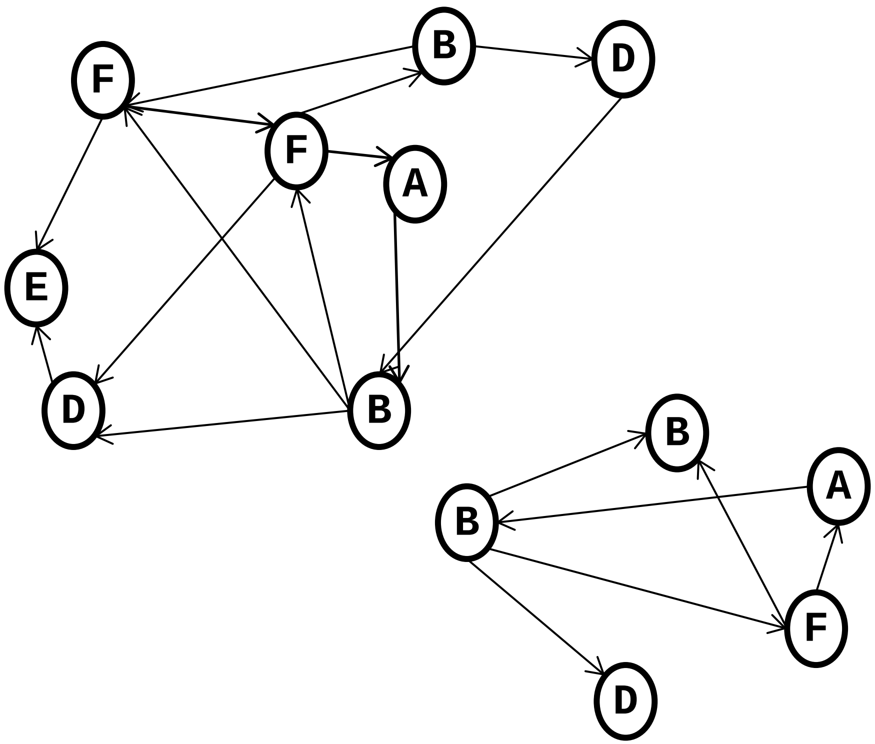
Some More Definitions

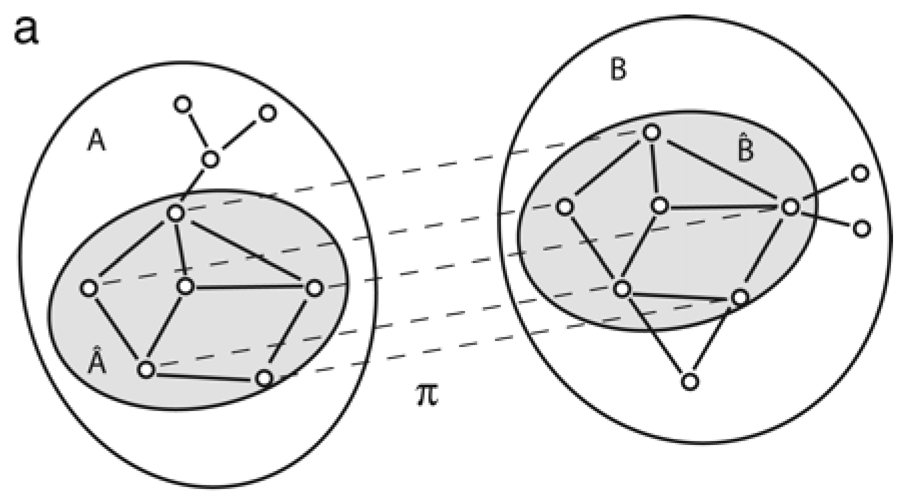
- Definition. *Let G=(V, E) be a directed graph.*
  - *Any G'=(V', E') is called a subgraph of G, if $V' \subseteq V$ and $E' \subseteq E$ and for all $(v_1,v_2) \in E'$: $v_1,v_2 \in V'$*
  - *For any $V' \subseteq V$, the graph $(V', E \cap (V' \times V'))$ is called the induced subgraph of G (induced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



Famous Problem

- Subgraph isomorphism problem: Given a graph $G_1=(V_1,E_1)$ and a graph $G_2=(V_2,E_2)$: Is there an isomorphism $f:V_1 \rightarrow V_2$ such that $f(G_1)$ is a subgraph of G_2?





Content of this Lecture

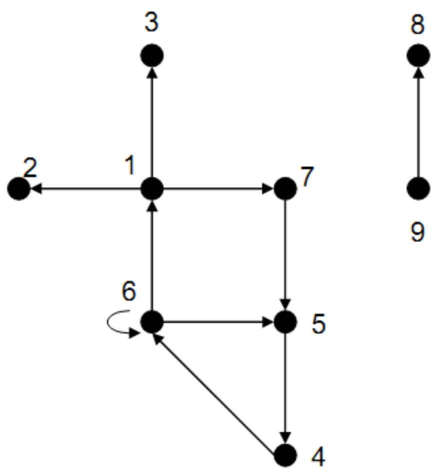
- Graphs
- Definitions
- Representing Graphs
- Traversing Graphs
- Connected Components
- Shortest Paths

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 which primitive operations take how long
- 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]

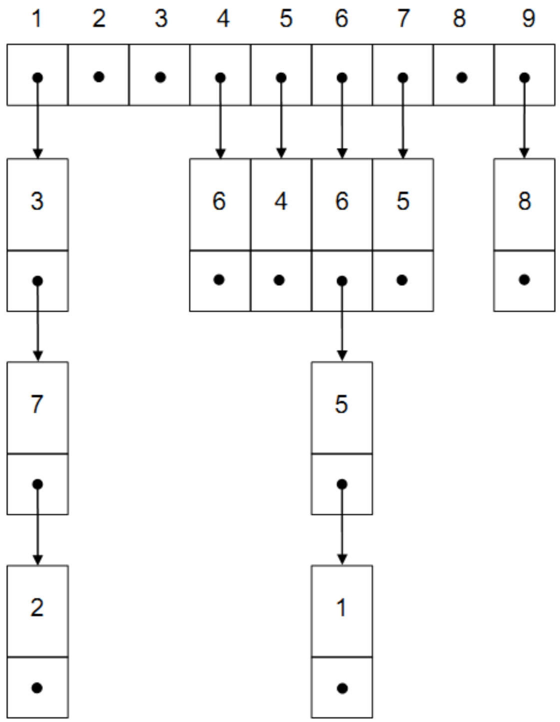
Graph



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 weighted graphs, M[i,j] contains the weight
  - 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)

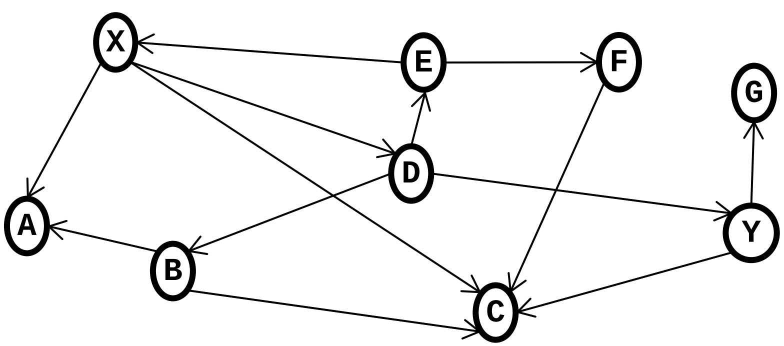
Comparison

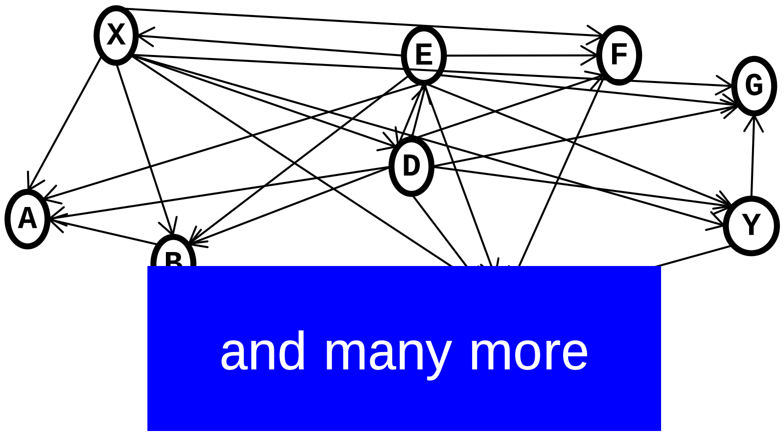
	Matrix	Lists
Test if a given edge exists	O(1)	O(log(k))
Find all outgoing edges of a given v	O(n)	O(k)
Space of G	$O(n^2)$	O(n+m)

- With n=\|V\|, m=\|E\|
- We assume a node-indexed array
  - L is an array and nodes are unique 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
- Shortest Paths

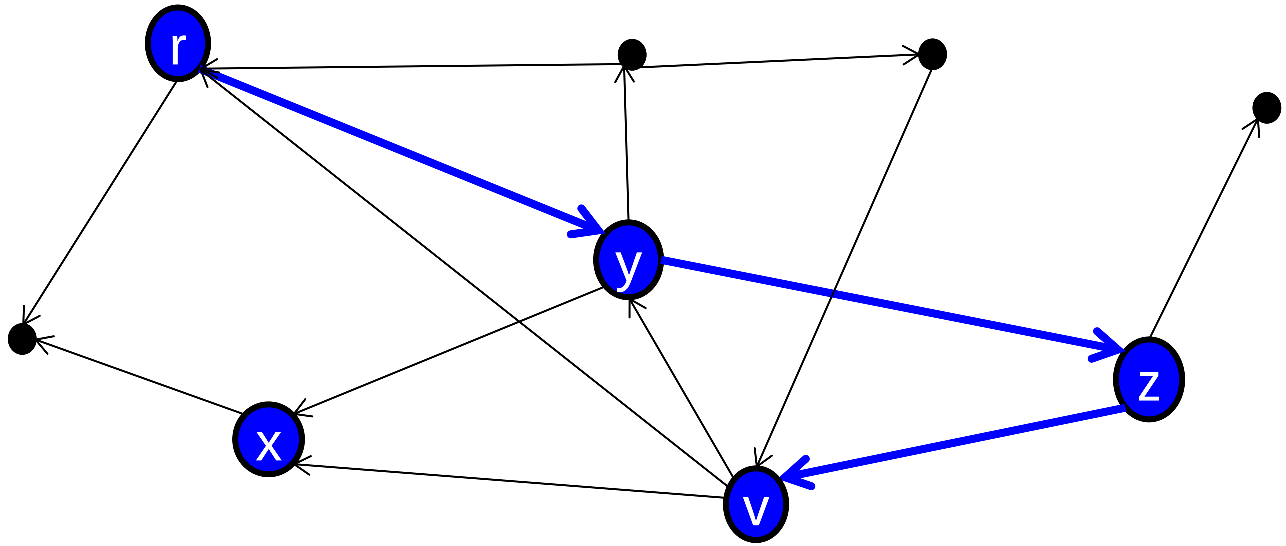
Graph Traversal

- 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

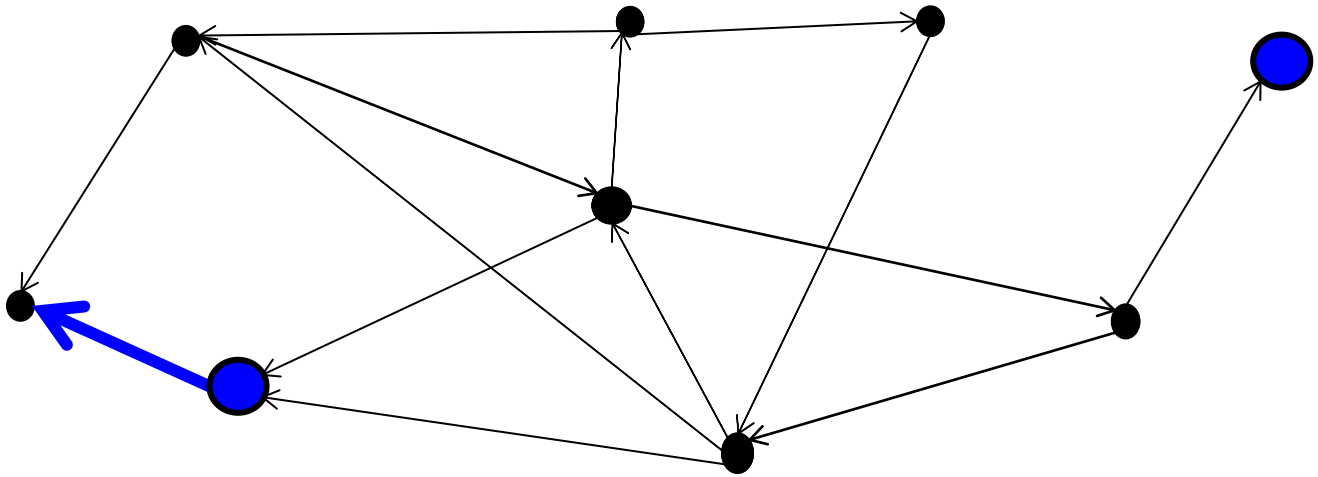
- Any naïve traversal will visit nodes more than once
  - If there is at least one node with more than one incoming edge
- Any naïve traversal will run into infinite loops
  - If the graphs contains at least one cycle (is cyclic)
- Breaking cycles / avoiding multiple visits
  - Assume we started the 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

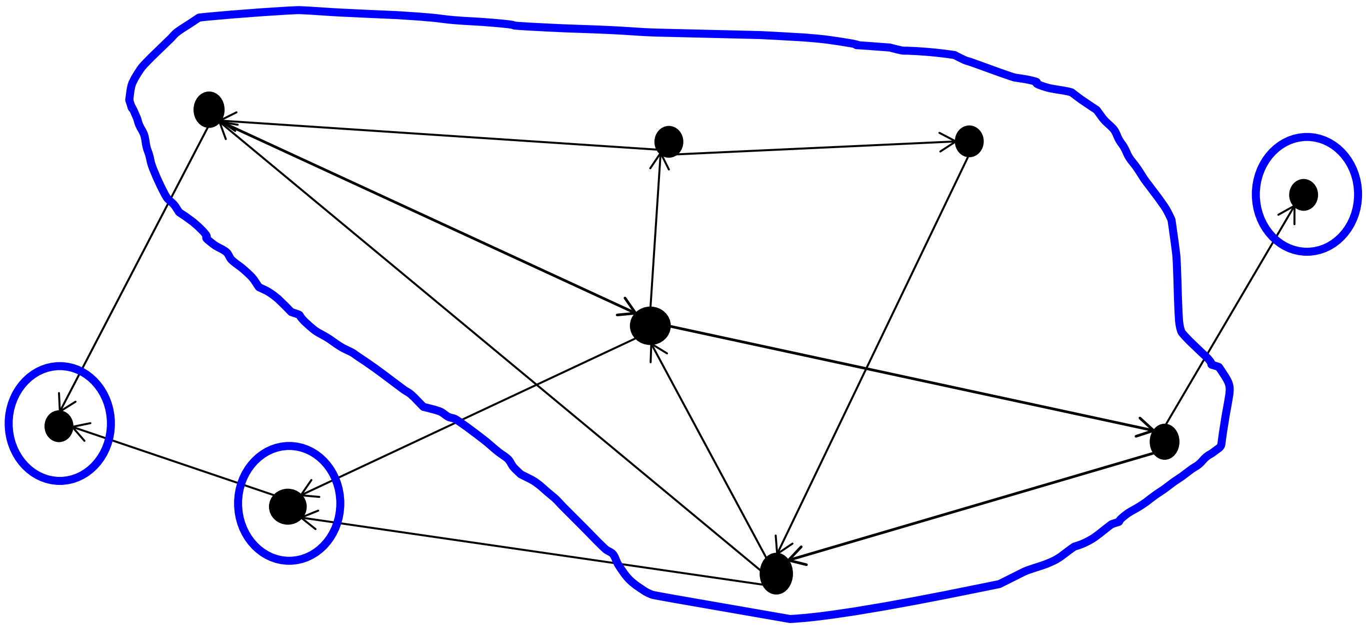
Where do we Start?



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 ((V,E) graph) {
  U := V;       # Unseen nodes
  S := ∅;       # Seen nodes
  while U≠∅ do
    v := any_node_from( U);
    traverse( v, S, U);
  end while;
}
```

Called once for every connected component

```
func void traverse (v node,
                    S,U list)
{
  t := new Stack();
  t.put( v);
  while not t.isEmpty() do
    n := t.getNext();
    print n;   # Do something
    U := U \ {n};
    S := S ∪ {n};
    c := n.outgoingNodes();
    foreach x in c do
      if x∈U then
        t.put( 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
- S and U can be implemented as bit-array of size |V|, allowing O(1) operations
  - Setting, removing, testing nodes
- Altogether: O(n+m)

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

Content of this Lecture

- Graphs
- Definitions
- Representing Graphs
- Traversing Graphs
- Connected Components
- Shortest Paths

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 traverse(v) are added to this 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

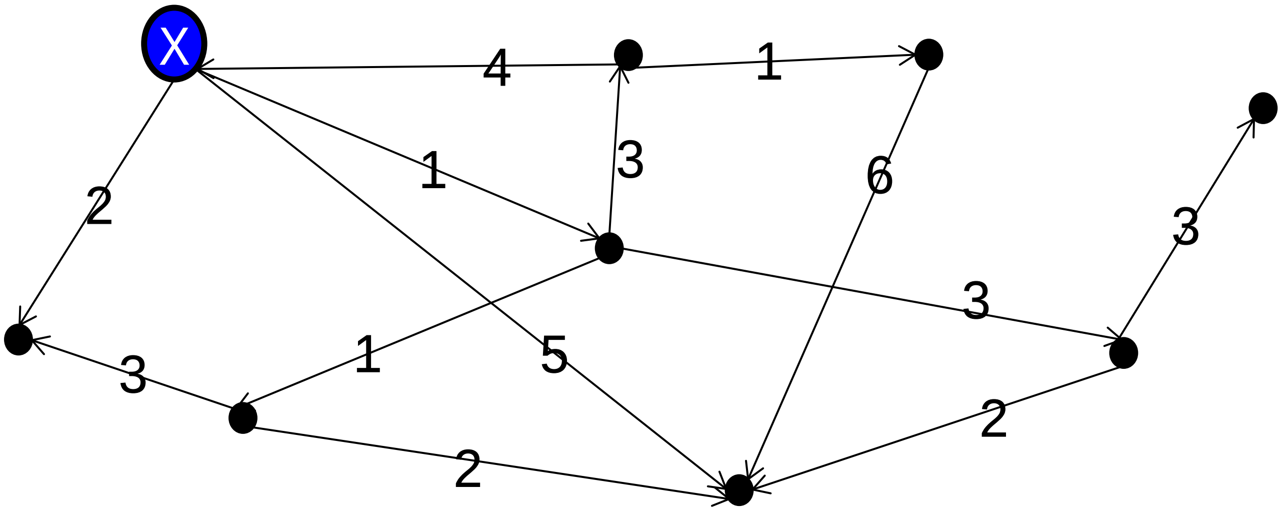
Content of this Lecture

- Graphs
- Definitions
- Representing Graphs
- Traversing Graphs
- Connected Components
- Shortest Paths
  - Single-Source-Shortest-Paths: Dijkstra’s Algorithm
  - Shortest Path between two given nodes
  - Other

Distance in Graphs

- Definition
  Let G=(V, E) be a graph. The distance d(u,v) between any two nodes u,v∈V for u≠v is defined as
  - *G un-weighted: The length of the shortest path from u to v, or ∞ if no path from u to v exists*
  - *G weighted: The minimal aggregated edge weight of all non-cyclic paths from u to v, or ∞ if no path from u to v exists*
  - *If u=v, d(u,v)=0*
- Remark
  - Distance in un-weighted graphs is the same as distance in weighted graphs with unit costs
  - Beware of negative cycles in directed graphs

Single-Source Shortest Paths in a Graph



- Task: Find the distance between X and all other nodes
- Only positive edge weights allowed
  - Bellman-Ford algorithm solves the general case

Algorithmic Idea

- Enumerate paths by iteratively extending already found shortest paths by all possible extensions
  - All edges outgoing from the end node of a short path
- These extensions
  - ... either lead to a node which we didn't reach before – then we found a path, but cannot yet be sure it is the shortest
  - ... or lead to a node which we already reached but we are not yet sure of we found the shortest path to it – update current best distance
  - ... or lead to a node which we already reached and for which we also surely found a shortest path already – these can be ignored
- Eventually, we enumerate nodes by their distance

Algorithm

```
G = (V, E);
x : start_node;    # x∈V
A : array_of_distances_from_x;
∀i: A[i]:= ∞;
L := V;        # organized as PQ
A[x] := 0;
while L≠∅
  k := L.get_closest_node();
  L := L \ k;
  forall (k,f,w)∈E do
    if f∈L then
      new_dist := A[k]+w;
      if new_dist < A[f] then
        A[f] := new_dist;
        update( L);
      end if;
    end if;
  end for;
end while;
```

- We enumerate nodes by length of their shortest paths
  - In the first loop, we pick x and update distances (A) to all adjacent nodes
  - When we pick a node k, we already have computed its distance to x in A
  - We adapt the current best distances to all neighbors of k we haven't picked yet
- Once we picked all nodes, we are done

Dijkstra's Algorithm – Single Operations

```
1.  G = (V, E);
2.  x : start_node;     # x∈V
3.  A : array_of_distances_from_x;
4.  ∀i: A[i]:= ∞;
5.  L := V;       # organized as PQ
6.  A[x] := 0;
7.  while L≠∅
8.    k := L.get_closest_node();
9.    L := L \ k;
10.   forall (k,f,w)∈E do
11.     if f∈L then
12.       new_dist := A[k]+w;
13.       if new_dist < A[f] then
14.         A[f] := new_dist;
15.         update( L);
16.       end if;
17.     end if;
18.   end for;
19. end while;
```

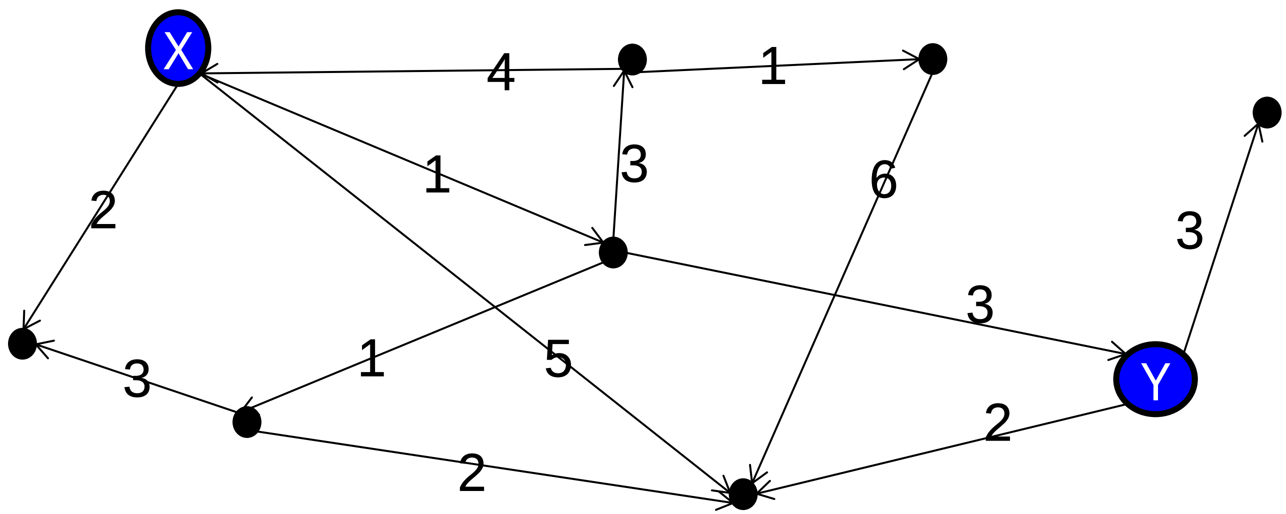
- Assume a heap-based PQ L
  - L holds at most all nodes (n)
  - L4: O(n)
  - L5: O(n) (build PQ)
  - L8: O(1) (getMin)
  - L9: O(log(n)) (deleteMin)
  - L10: O(m) (with adjacency list)
  - L11: O(1)
    - Requires additional array LA of size |V| storing membership of nodes in L
  - L15: O(log(n)) (updatePQ)
    - Store in LA pointers to nodes in L; then remove/insert node

Dijkstra's Algorithm - Loops

```
1.  G = (V, E);
2.  x : start_node;    # x∈V
3.  A : array_of_distances;
4.  ∀i: A[i]:= ∞;
5.  L := V;       # organized as PQ
6.  A[x] := 0;
7.  while L≠∅
8.    k := L.get_closest_node();
9.    L := L \ k;
10.   forall (k,f,w)∈E do
11.     if f∈L then
12.       new_dist := A[k]+w;
13.       if new_dist < A[f] then
14.         A[f] := new_dist;
15.         update( L);
16.       end if;
17.     end if;
18.   end for;
19. end while;
```

- Central costs
  - L9: O(log(n)) (deleteMin)
  - L15: O(log(n)) (del+ins)
- Loops
  - Lines 7-18: O(n)
  - Line 10-17: All edges exactly once
  - Together: O(m+n)
- Altogether: O((n+m)*log(n))
  - With Fibonacci heaps: Amortized costs are O(n*log(n)+m))
  - Also possible in $O(n^2)$; this is better in dense graphs ($m \sim n^2$)

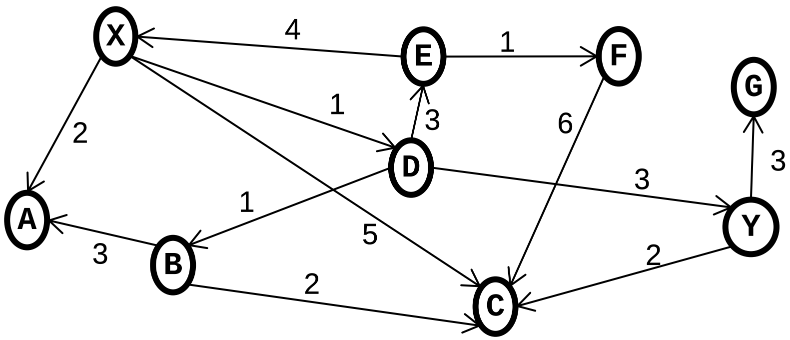
Single-Source, Single-Target



- Task: Find the distance between X and only Y
  - There is no way to be WC-faster than Dijkstra in general graphs
    - We can stop as soon as Y appears at the min position of the PQ
    - We can visit edges in order of increasing weight (might help)
    - Worst-case complexity unchanged
- Things are different in planar graphs (navigators!)

Faster SS-ST Algorithms

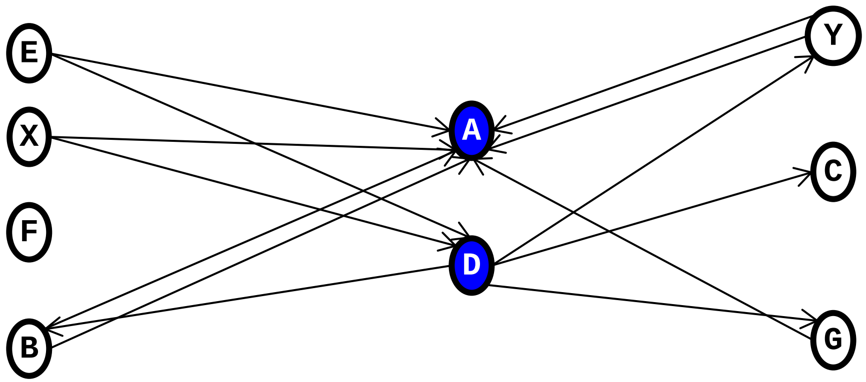
- Trick 1: Pre-compute all distances
  - Transitive closure with distances
  - Requires $O(|V|^2)$ space: Prohibitive for large graphs
  - How? See next lecture



→	A	B	C	D	E	F	G	X	Y
A	0	-	-	-	-	-	-	-	-
B	3	0	2	-	-	-	-	-	-
C	-	-	0	-	-	-	-	-	-
D	4	1	3	0	3	4	6	7	3
E	6	6	7	5	0	1	11	4	8
F	-	-	6	-	-	0	-	-	-
G	-	-	-	-	-	-	0	-	-
X	2	2	4	1	4	5	7	0	4
Y	-	-	2	-	-	-	3	-	0

Faster SS-ST Algorithms

- Trick 2: Two-hop cover with distances
  - Find a (hopefully small) set S of nodes such that
    - For every pair of nodes v_1,v_2, at least one shortest path from v_1 to v_2 goes through a node $s \in S$
    - Thus, the distance between v_1,v_2 is $\min\{ d(v_1,s)+d(s,v_2) \mid s \in S\}$
    - S is called a 2-hop cover
  - Problem: Finding a minimal S is NP-complete
    - And S need not be small



More Distances

- Graphs with negative edge weights
  - Shortest paths (in terms of weights) may be very long (edges)
  - Bellman-Ford algorithm is in $O(n^2*m)$
- All-pairs shortest paths
  - Only positive edge weights: Use Dijkstra n times
  - With negative edge weights: Floyd-Warshall in $O(n^3)$
    - See next lecture
- Reachability
  - Simple in undirected graphs: Compute all connected components
  - In digraphs: Use graph traversal or a special graph indexing method

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 disjoined, 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.