Algorithms and Data Structures

Graphs: Introduction and First Algorithms

Marius Kloft

This Course

• Introduction	2
• Complexity analysis	1
• Abstract Data Types	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!)	4
• The End	1
• Sum	**21/26**

Content of this Lecture

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

Graphs

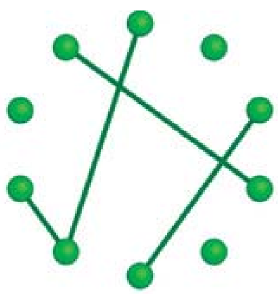
- Directed trees represent hierarchical relations
  - Directed trees represent relations that are
    - Asymmetric: parent_of, subclass_of, smaller_than, …
    - Cycle-free
    - Binary
  - Undirected trees: Symmetric relations, but still a hierarchy
- This excludes many real-life relations
  - friend_of, similar_to, reachable_by, html_linked_to, …
- Graphs can represent all binary relationships
  - Symmetric: Undirected graphs, asymmetric: Directed graphs
- 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
- Here: Only binary, simple, finite graphs

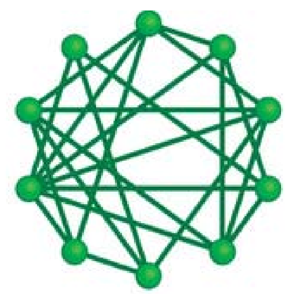
Exemplary Graphs

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

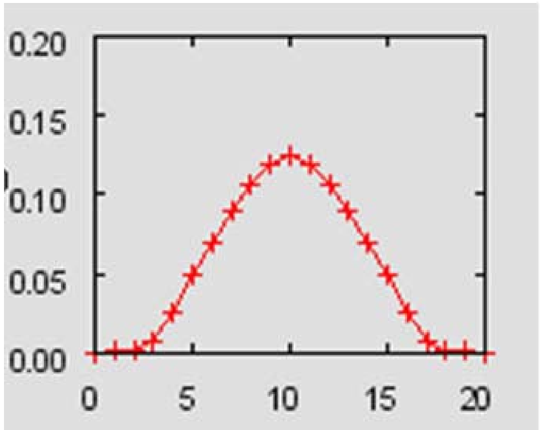


$p = 0.1$

$p = 0.25$

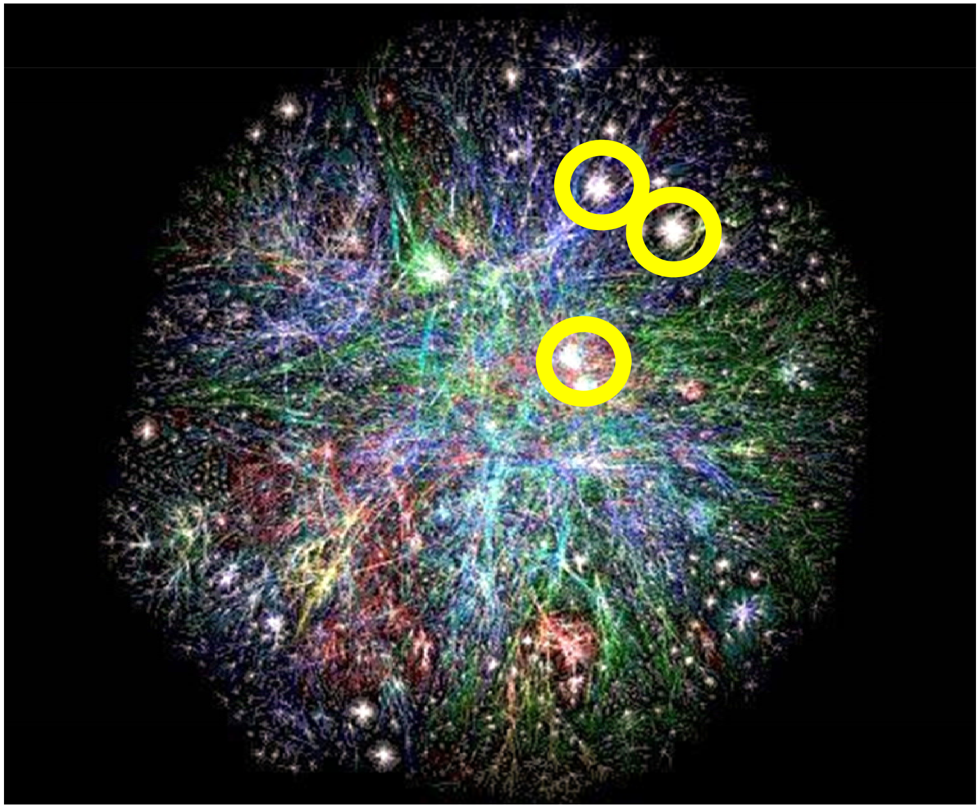


$p = 0.5$



  - For a graph with n nodes, this creates a graph where the degree of every node has expected value p*n, and the degree distribution follows a Poisson distribution

Web Graph



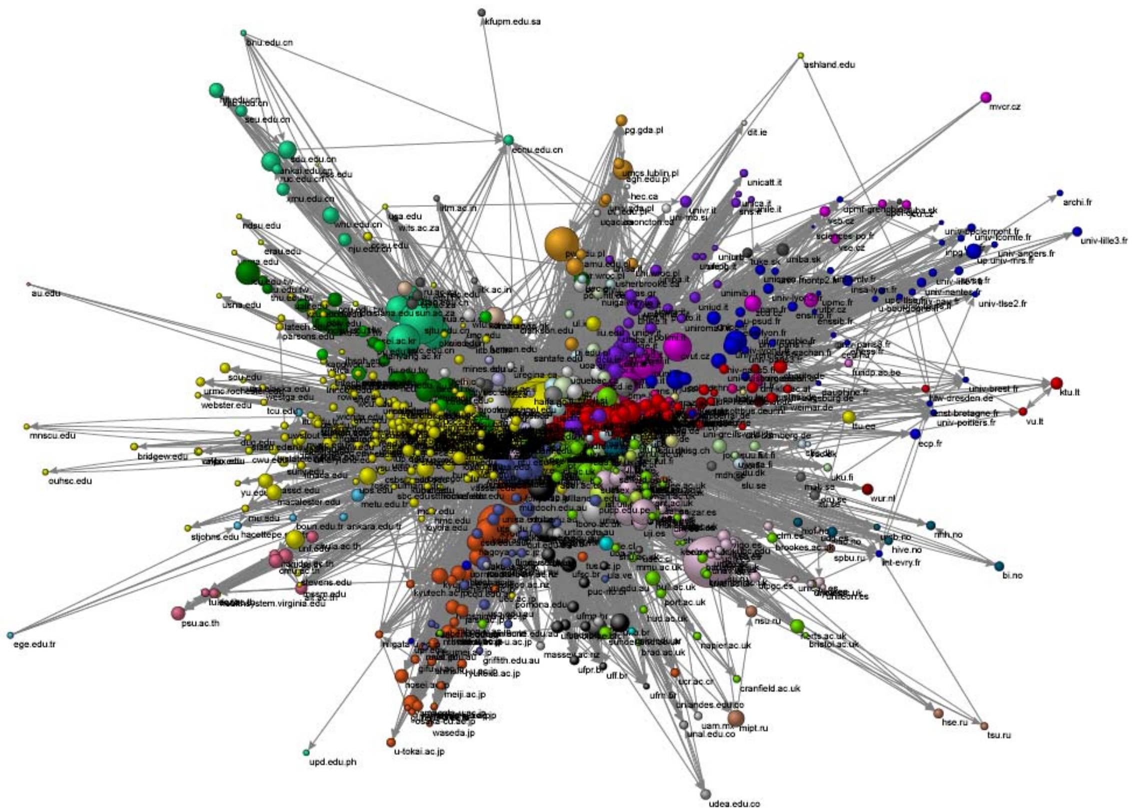
Note the strong local clustering

This is not a random graph

- Graph layout is difficult

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

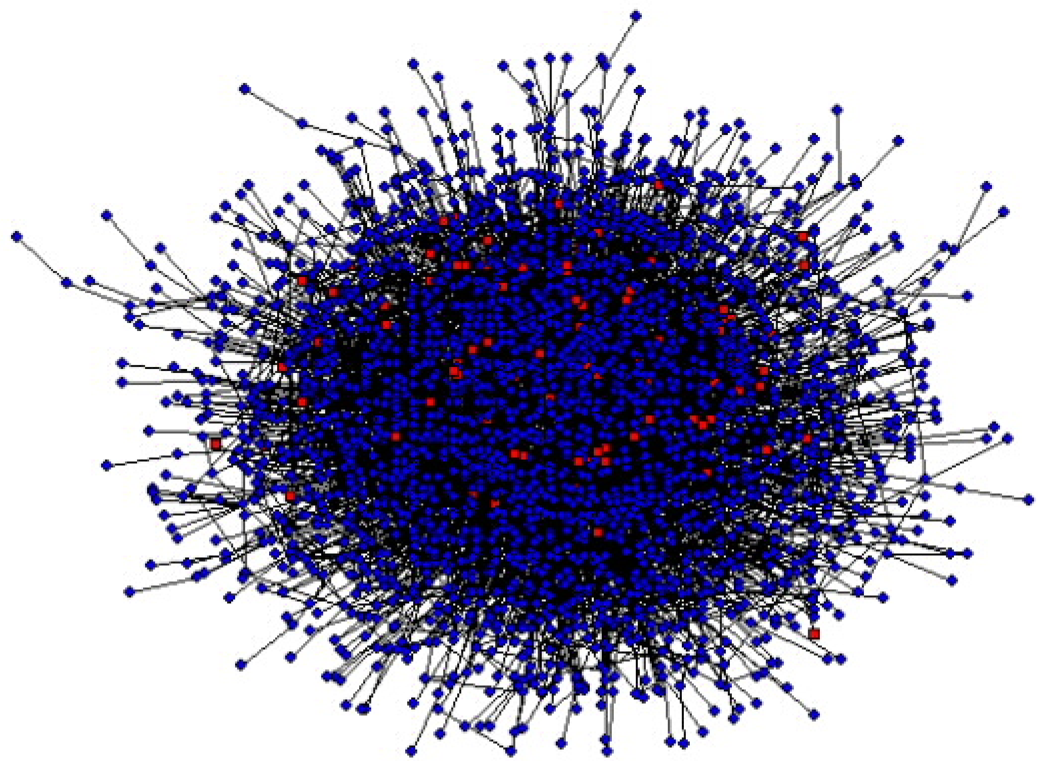
Universities Linking to Universities



- Small-World Property

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

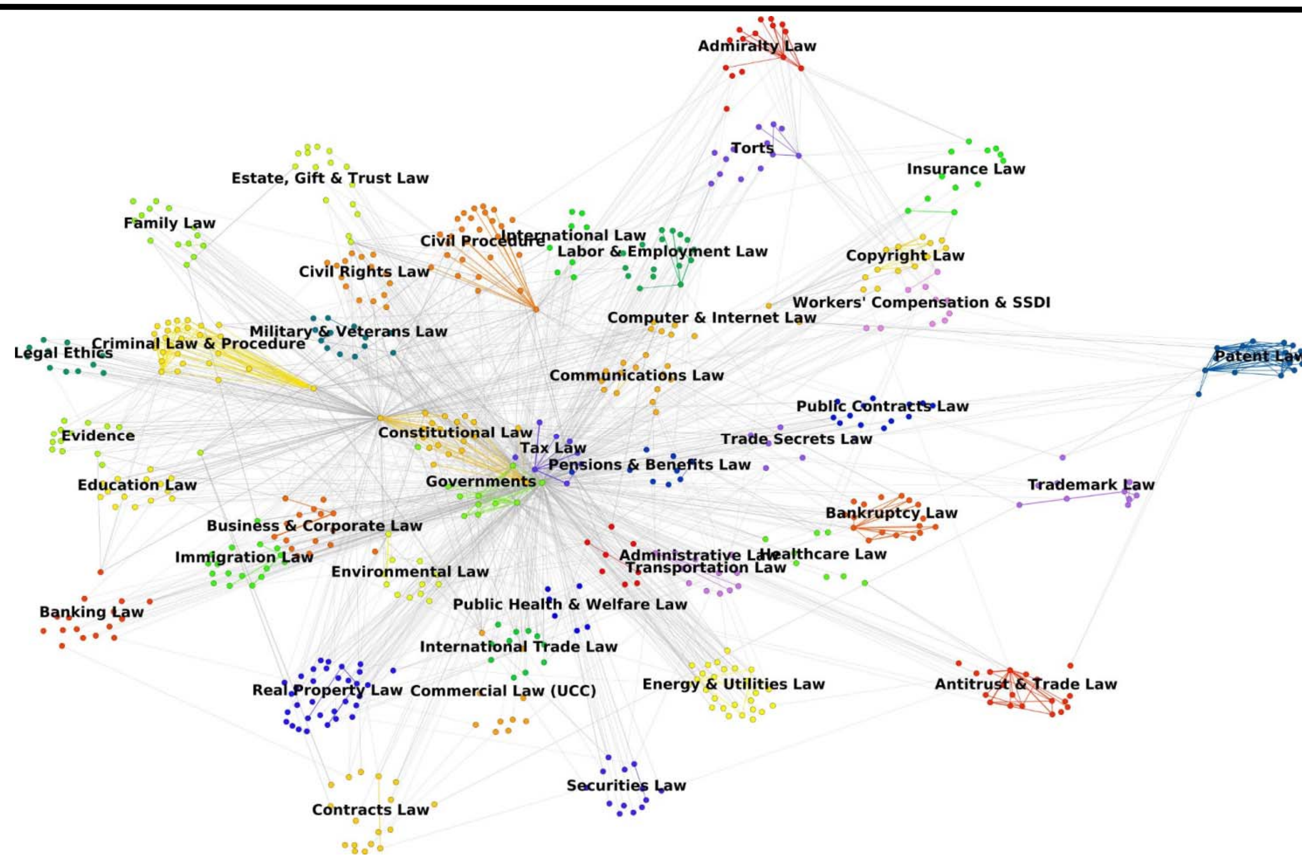
Human Protein-Protein-Interaction Network



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

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

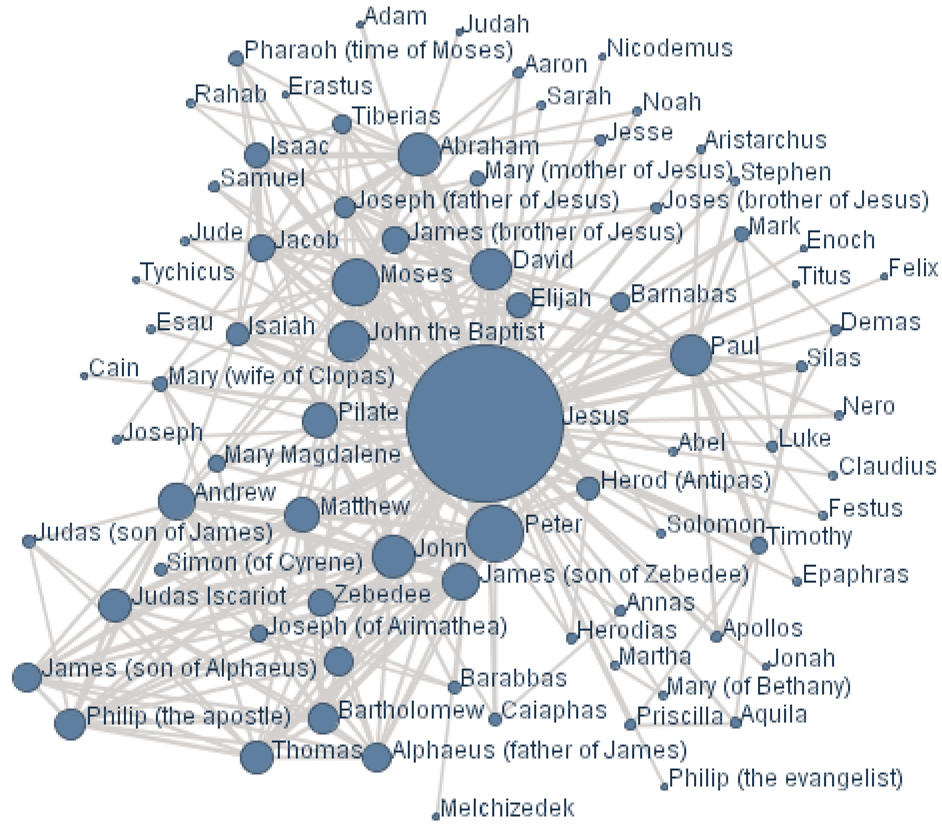
Word Co-Occurrence



- Words that are close have similar meaning
- Words cluster into topics

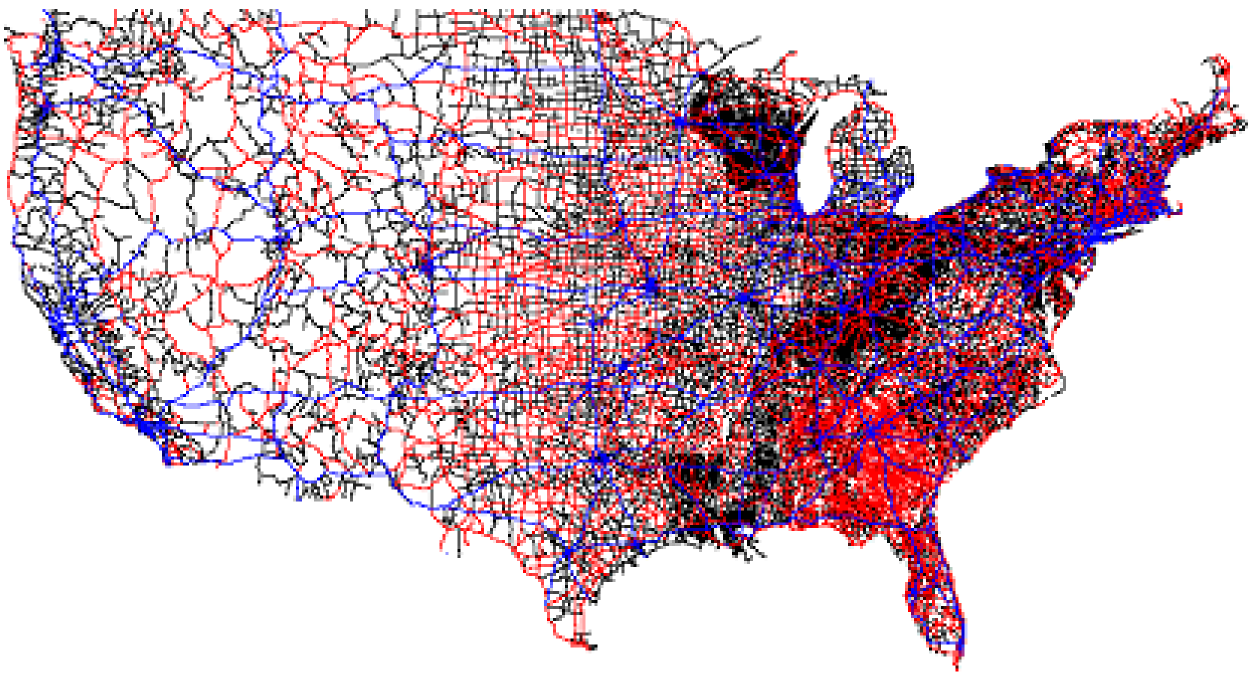
[http://www.michaelbommarito.com/blog/]

Social Networks



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

Road Network



- Specific property: Planar graphs

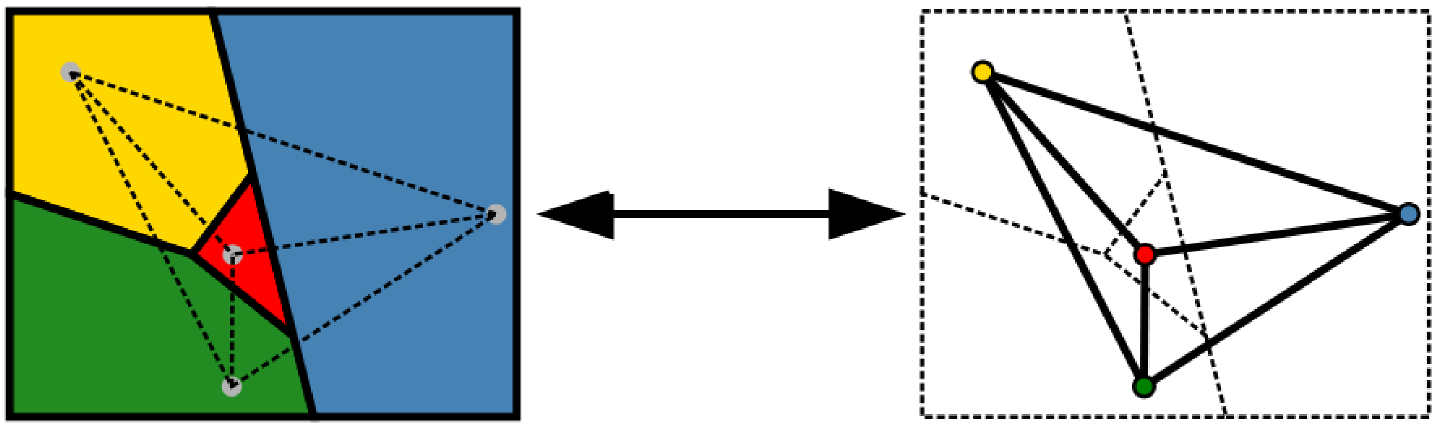
[Sanders, P. &Schultes, D. (2005).Highway Hierarchies Hasten Exact Shortest Path Queries. In *13th European Symposium on Algorithms (ESA), 568-579.*]

More Examples

- Graphs are also a wonderful abstraction

Coloring Problem

- How many colors do we need such that no two neighboring regions in a map / adjacent nodes in a graph share the same color?

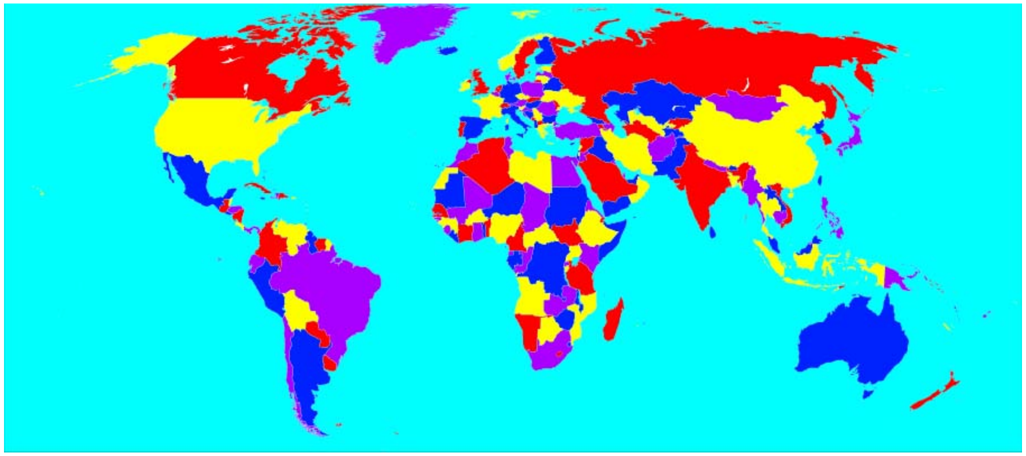


[http://www.wikipedia.de]

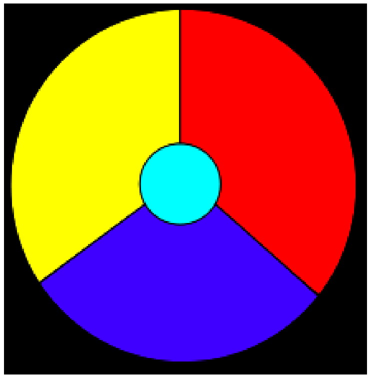
- Chromatic number: Number of colors sufficient to color a graph such that no adjacent nodes have the same color
- Every planar graph has chromatic number of at most 4

Every Map (Planar Graph) Can Be Colored With 4 Colors

- This is not simple to prove

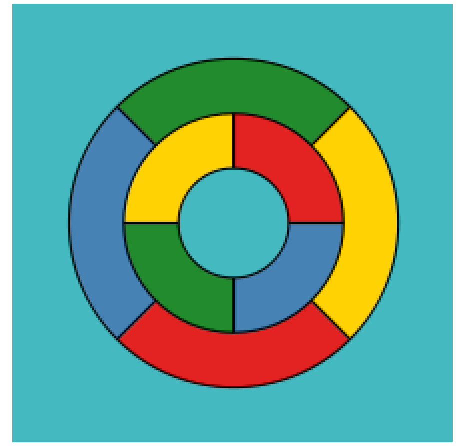


- It is easy to see that one sometimes needs at least four colors

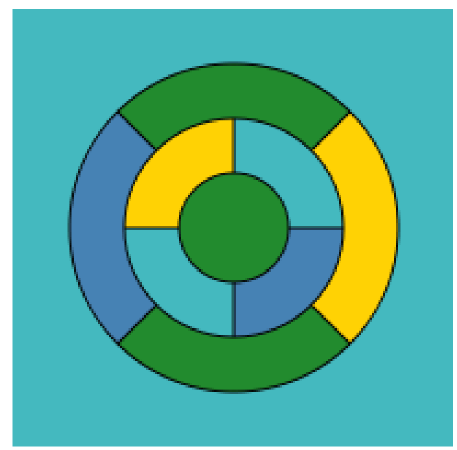


Every Planar Graph Can Be Colored With 4 Colors

- But don't we sometimes need 5 or more colors?



- Quiz: can we color this graph with <5 colors?
  - Yes



Every Planar Graph Can Be Colored With 4 Colors

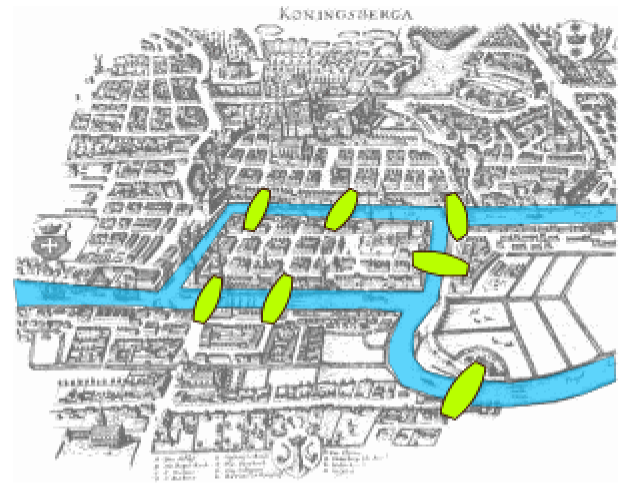
Remark:

- This was the first conjecture which until today was proven only by computers
  - Falls into many, many subcases – try all of them with a program

Appel & Haken, 1976

Seven Bridges of Königsberg (Euler, 1736)

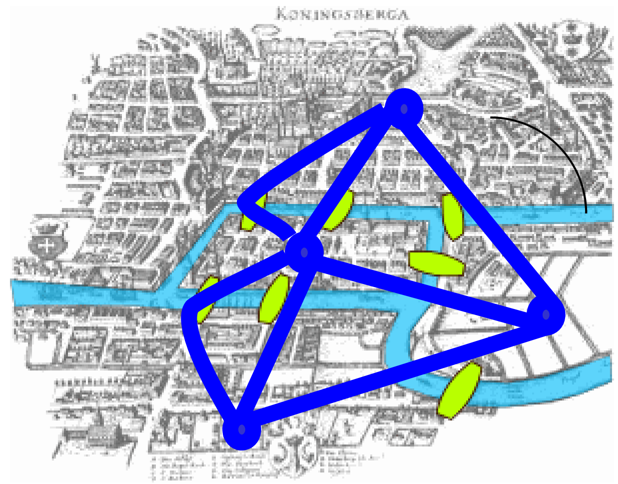
- Given a city with rivers and bridges: Is there a cycle-free path crossing every bridge exactly once?
  - Euler-Path



Source: Wikipedia.de

Königsberger Brückenproblem

- Given a city with rivers and bridges: Is there a cycle-free path crossing every bridge exactly once?
  - Euler-Path (simple to check)
- Hamiltonian path
  - … visits each vertex exactly once
  - NP complete to check



Content of this Lecture

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

Recall from Trees

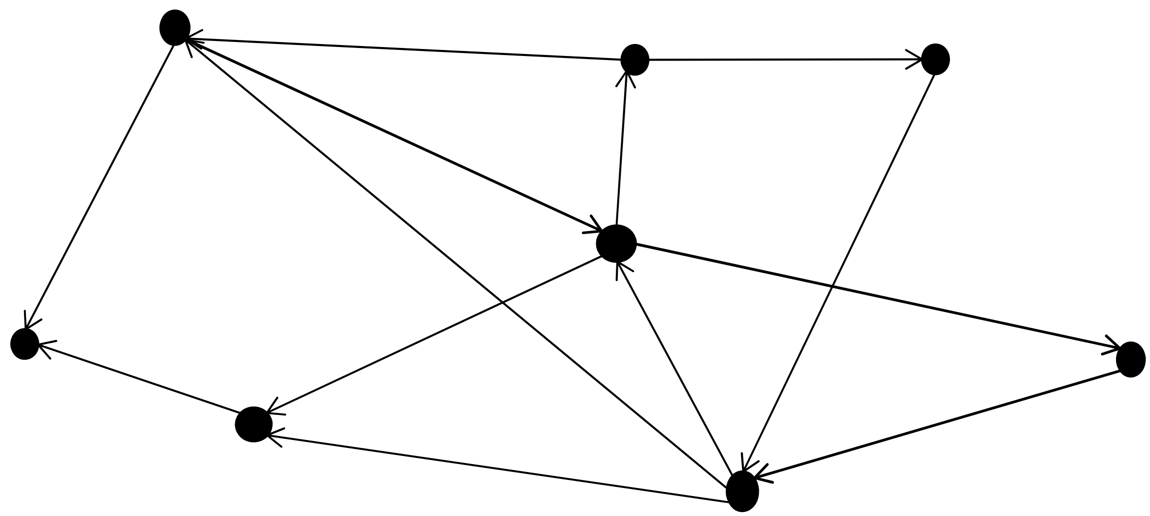
- Definition
  A graph $G=(V, E)$ consists of a set of vertices (nodes) V and a set of edges ($E \subseteq V \times V$).
  - *A sequence of edges $e_1, e_2, .., e_n$ is called a path iff $\forall 1 \leq i < n$: $e_i=(v', v)$ and $e_{i+1}=(v, v'')$; the length of this path is n*
  - *A path $(v_1,v_2),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

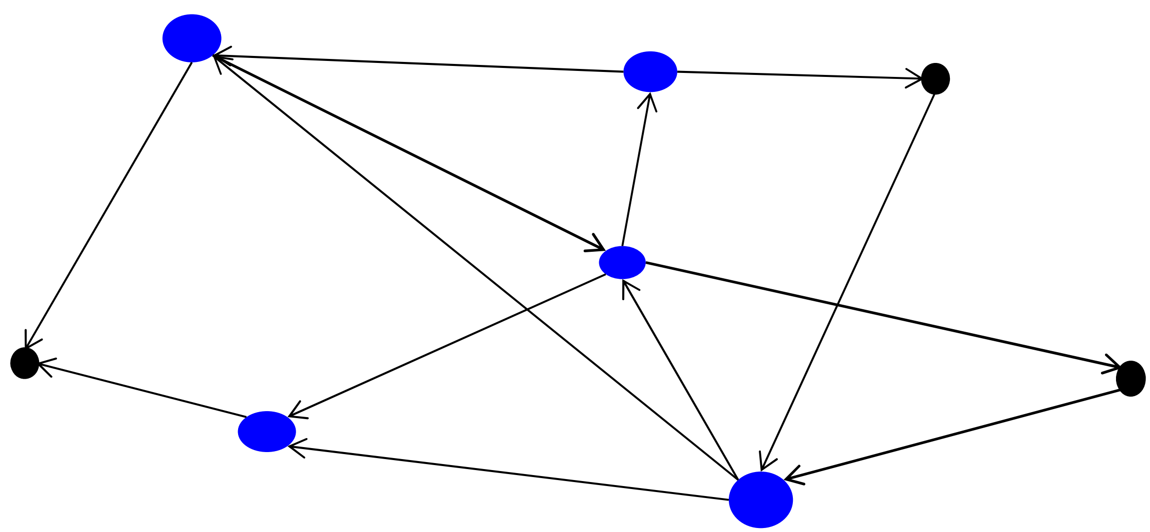
Some More Definitions

- 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')*



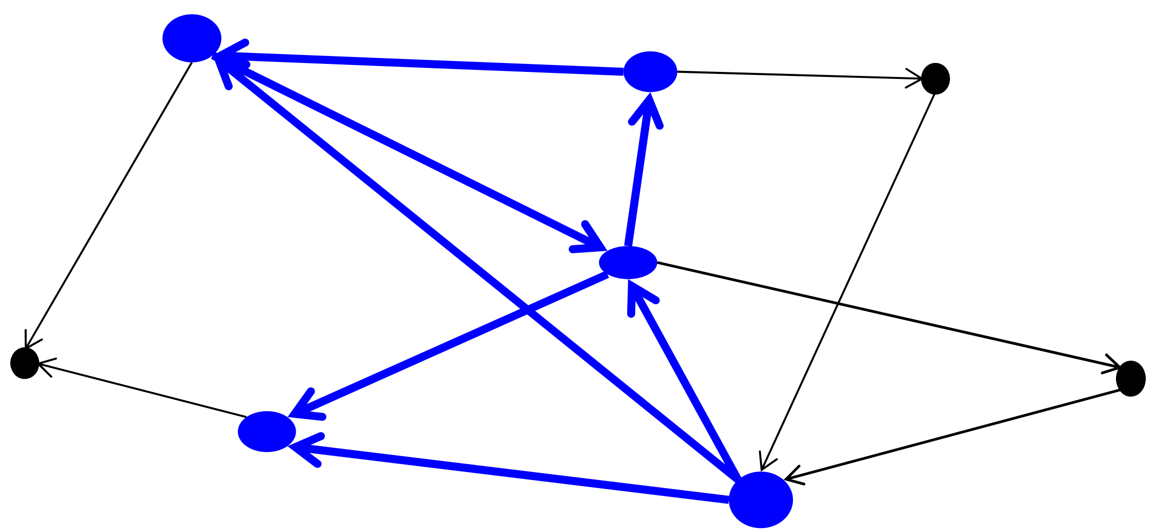
Some More Definitions

- Definition. *Let G=(V, E) be a directed graph.*
  - *Any G′=(V′, E′) is called a subgraph of G, if V′⊆V and E′⊆E and for all $(v_1,v_2)\in E'$: $v_1,v_2 \in V'$*
  - *For any V′⊆V, the graph (V′, E∩(V′×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′)*



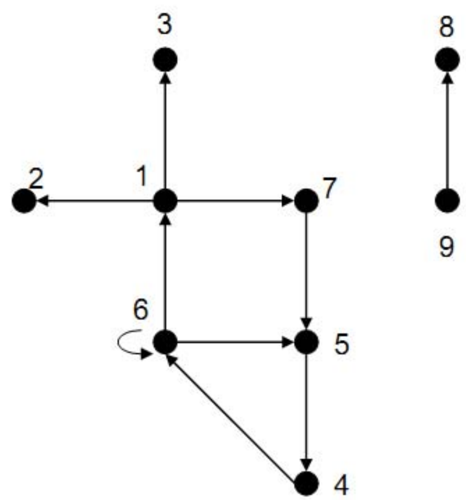
Data Structures

- 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
- Appropriateness depends on the specific problem one wants to study and the nature of the graphs
  - Shortest paths, transitive hull, cliques, spanning trees, …
  - Random, sparse/dense, scale-free, planar, bipartite, …

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 $(v_i, v_j) \in E$*



	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

[OW93]

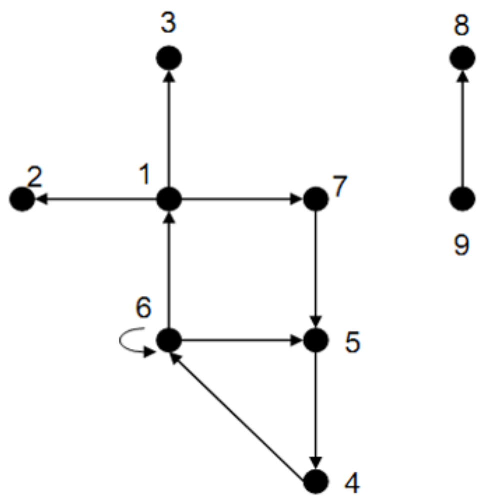
Adjacency Matrix

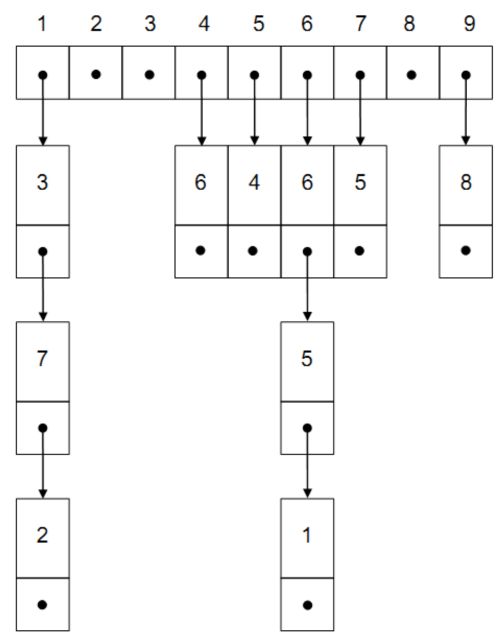
- Remarks:
  - Allows to test existence of an edge in O(1)
  - Requires O(|V|) to obtain all in-coming (outgoing) edges of a node
  - For large graphs, M is too large to be of practical use
  - If G is sparse (much less edges than $|V|^2$), M wastes a lot of space
  - If G is dense, M is a very compact 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 $\Omega(|V|^2)$

	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

- Definition
  Let G=(V, E). The adjacency list L_G for G is a list containing all nodes of G. The entry representing $v_i \in V$ also contains a list of all edges outgoing (or incoming or both) from v_i.

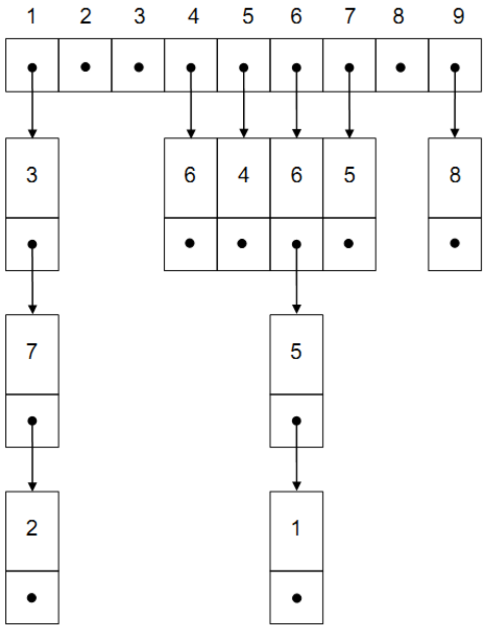




[OW93]

Adjacency List

- 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(k)) – 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)



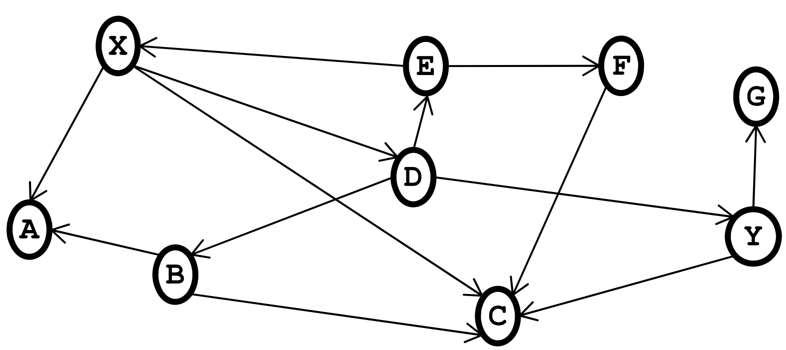
Comparison

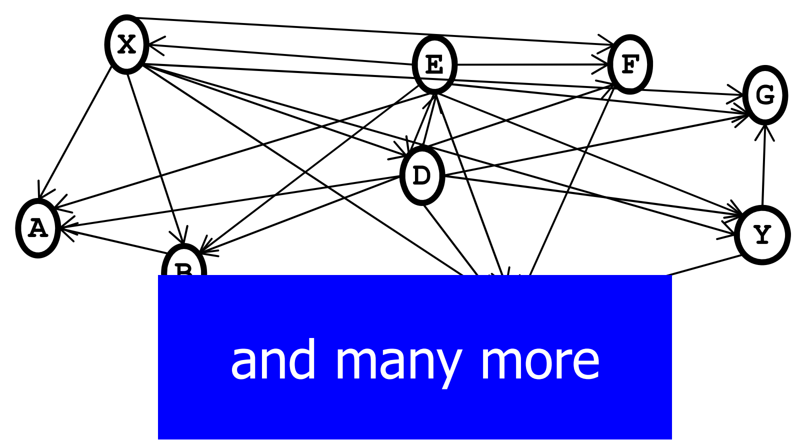
	Matrix	**Lists**
Test an edge for given v	O(1)	O(log(k))
All outgoing edges of v	O(n)	O(k)
Space	$O(n^2)$	O(n+m)

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





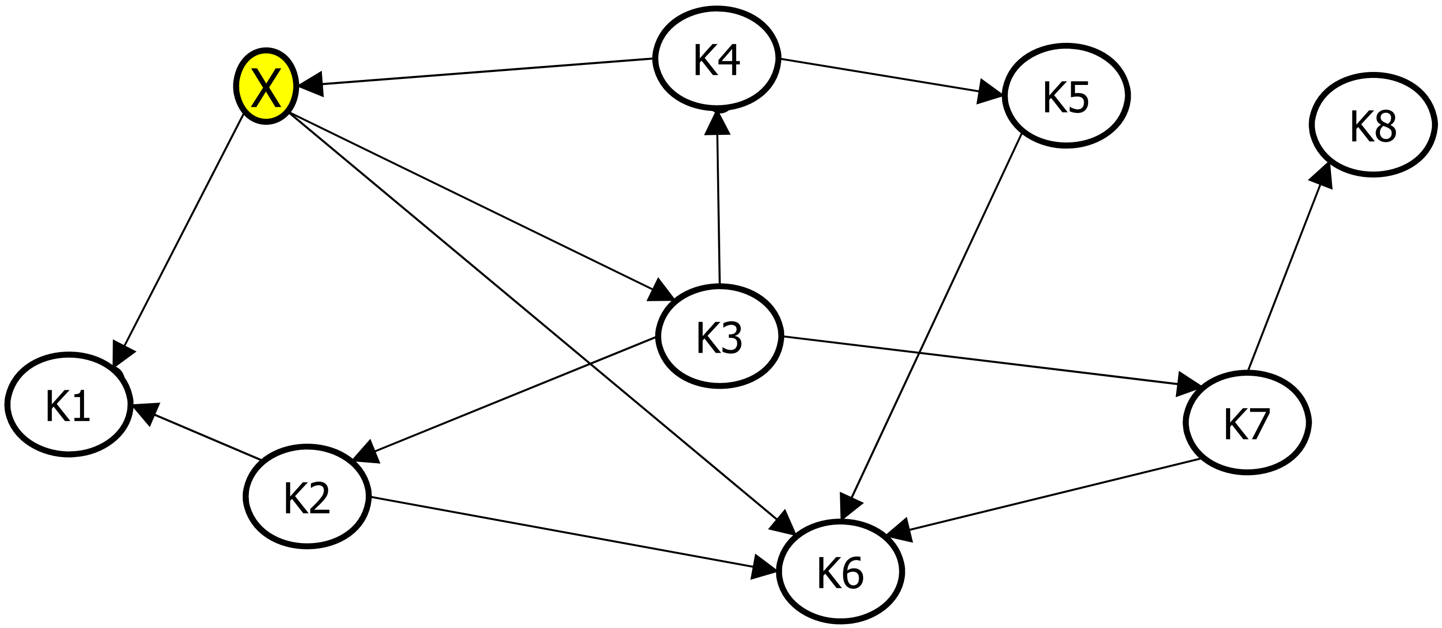
Content of this Lecture

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

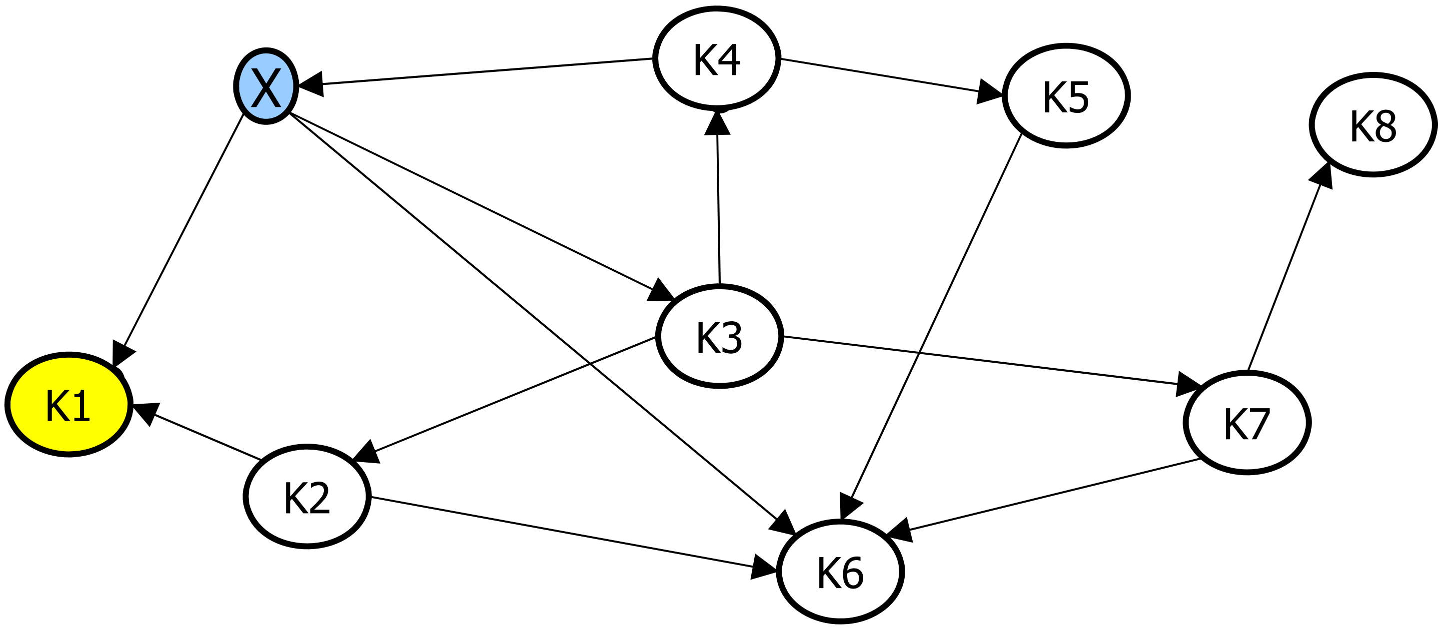
Graph Traversal

- One thing we often do with graphs is traversing them
  - "Traversal" means visiting every node exactly once
    - Not necessarily on one consecutive path (Hamiltonian path)
- Two popular orders of traversal
  - Depth-first: Using a stack
  - Breadth-first: Using a queue
  - The scheme is identical to that in tree traversal (lecture 6)

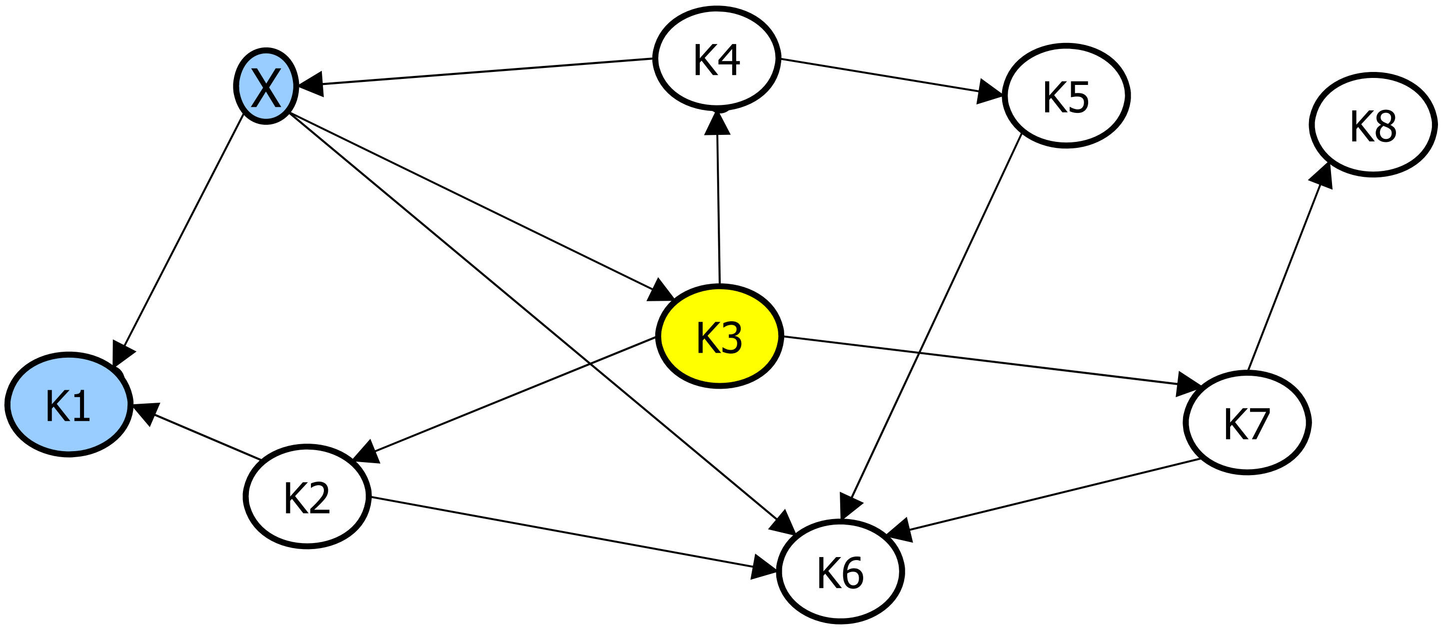
Example: Breadth-first Traversal



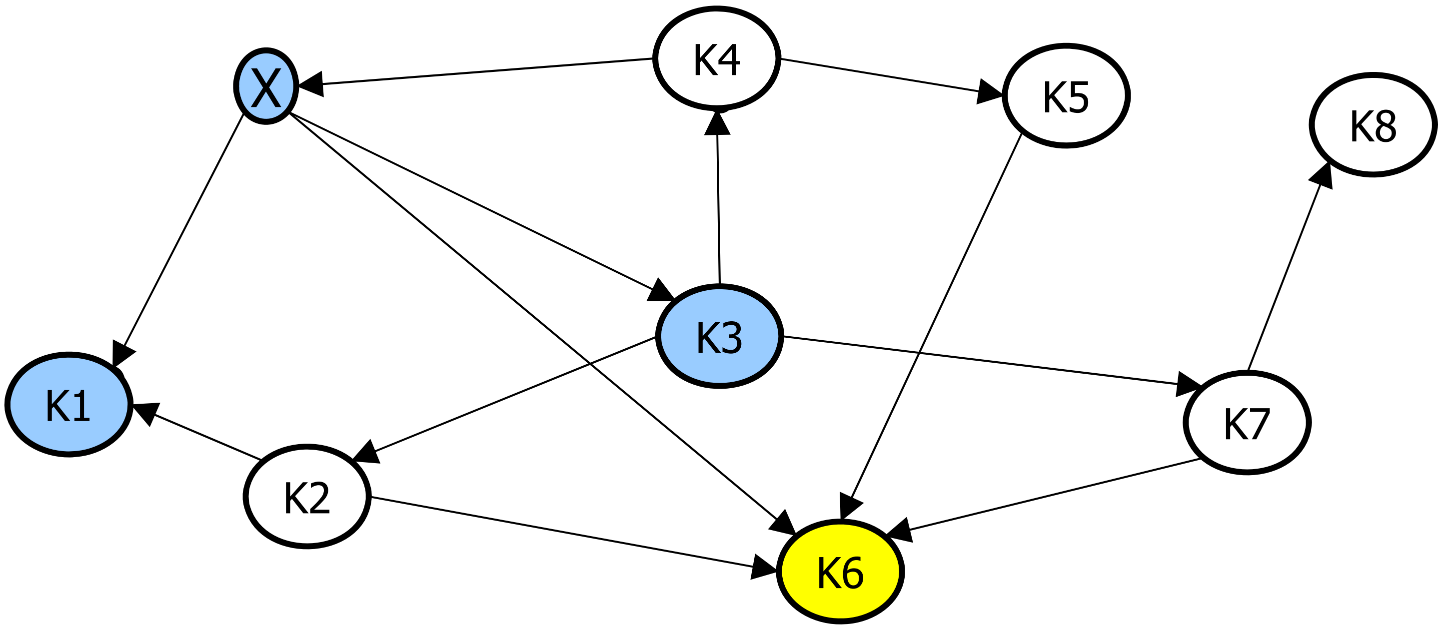
Example: Breadth-first Traversal



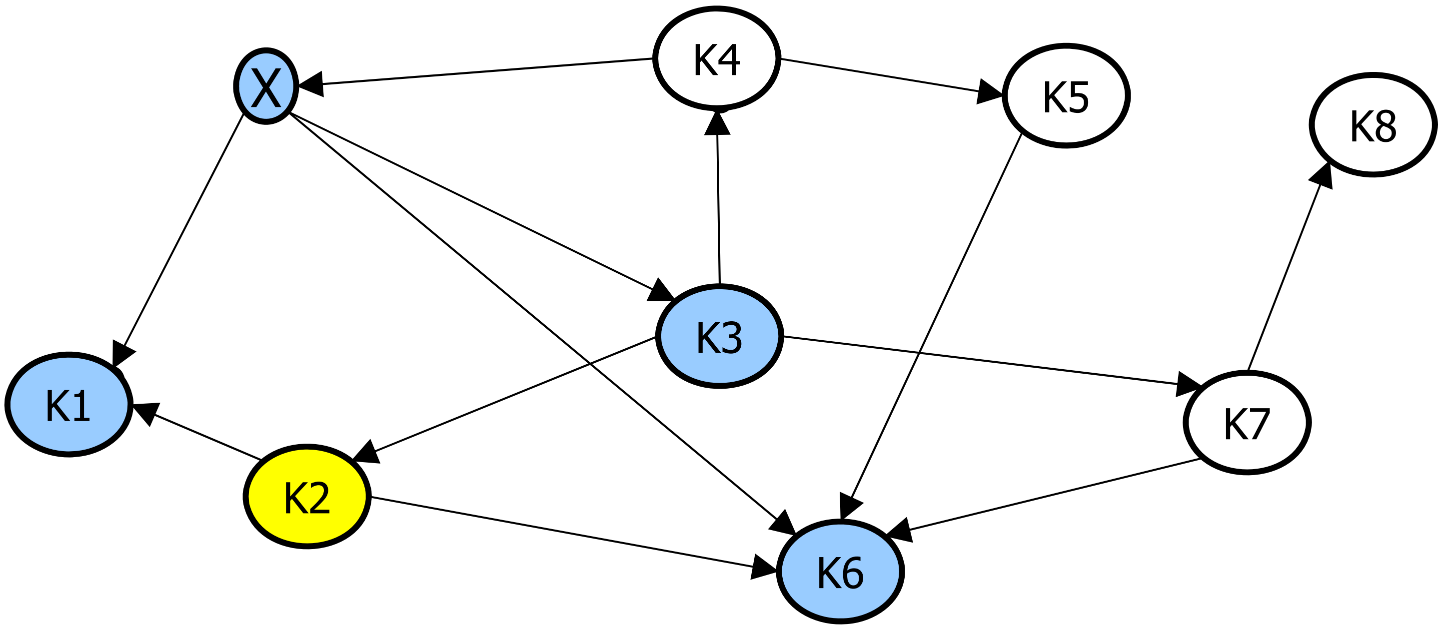
Example: Breadth-first Traversal



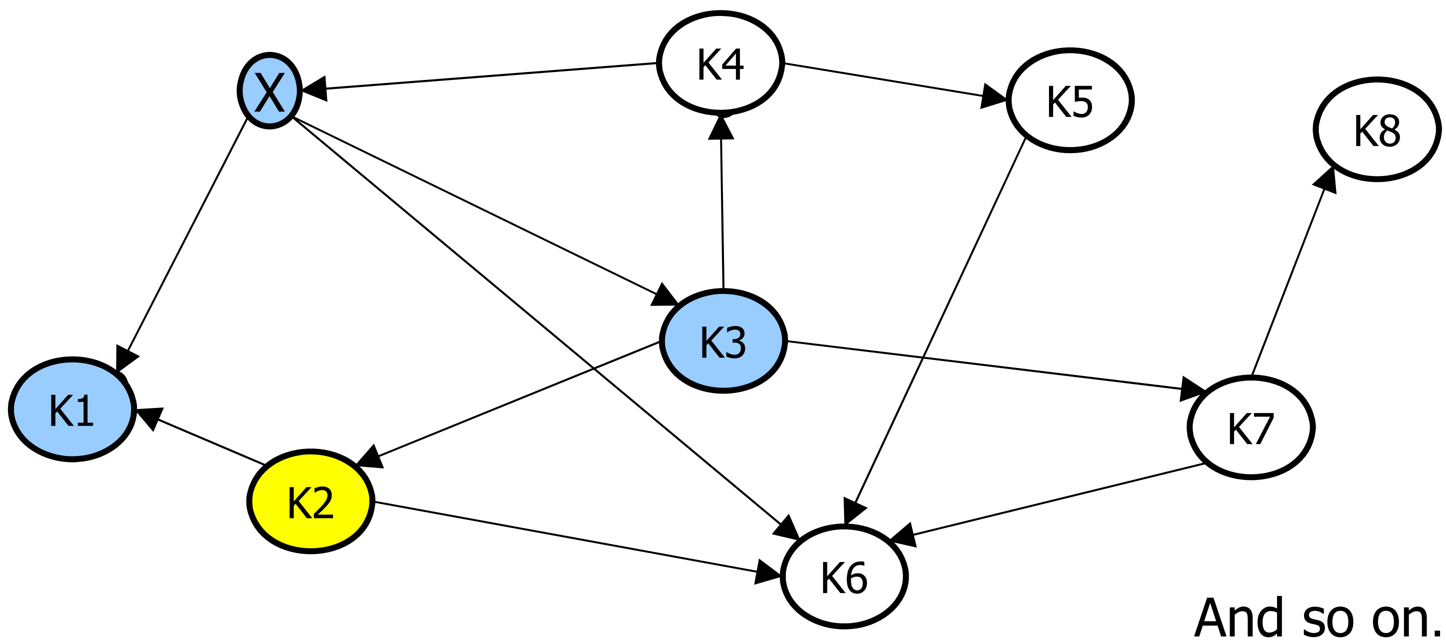
Example: Breadth-first Traversal



Example: Breadth-first Traversal

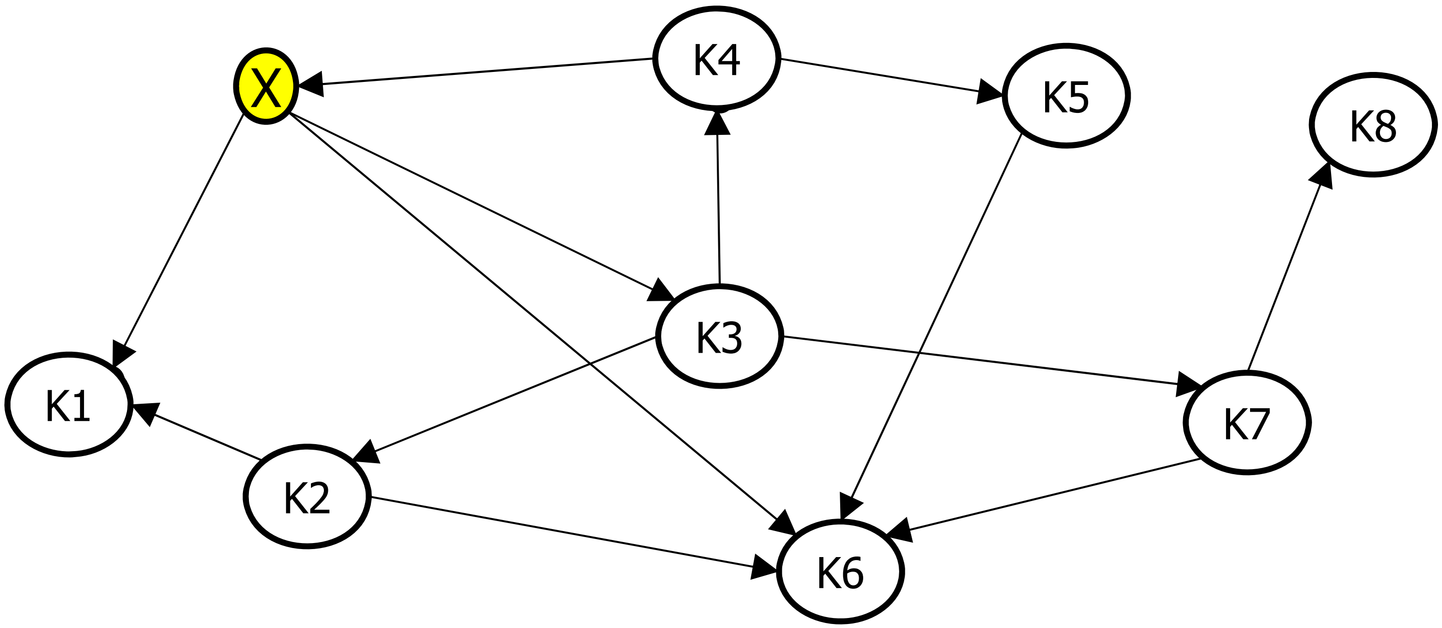


Example: Breadth-first Traversal

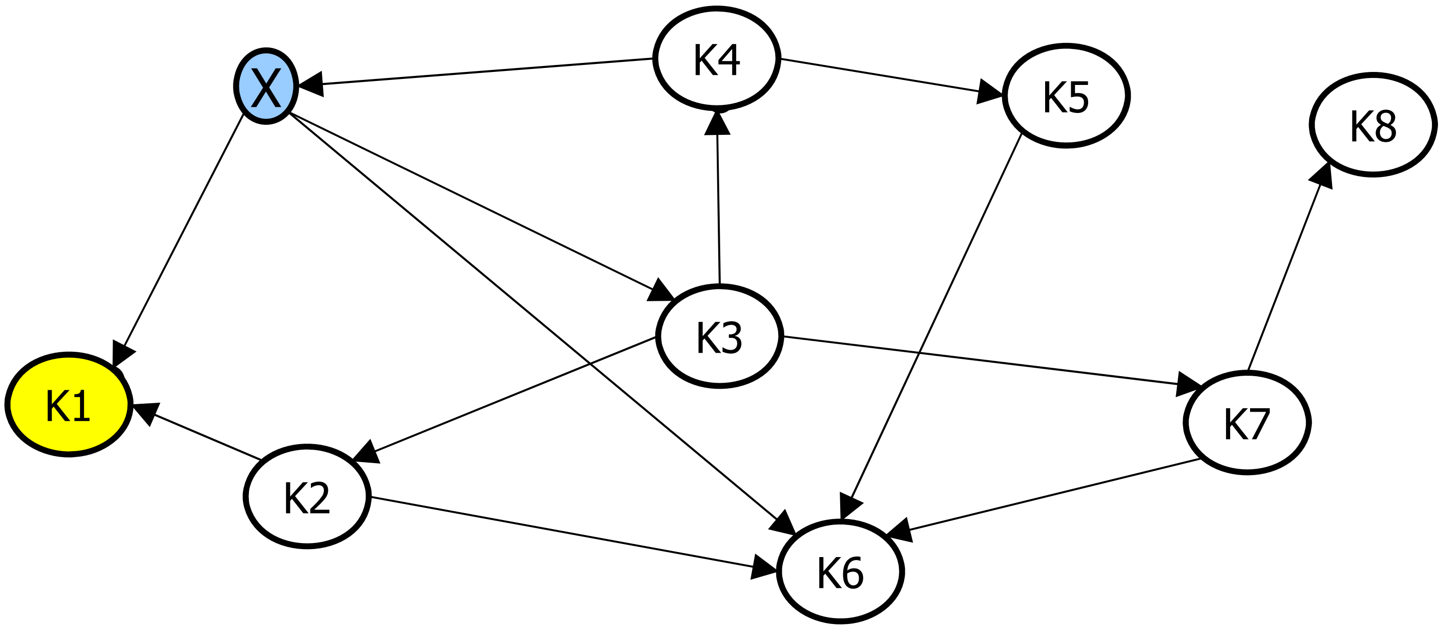


And so on…

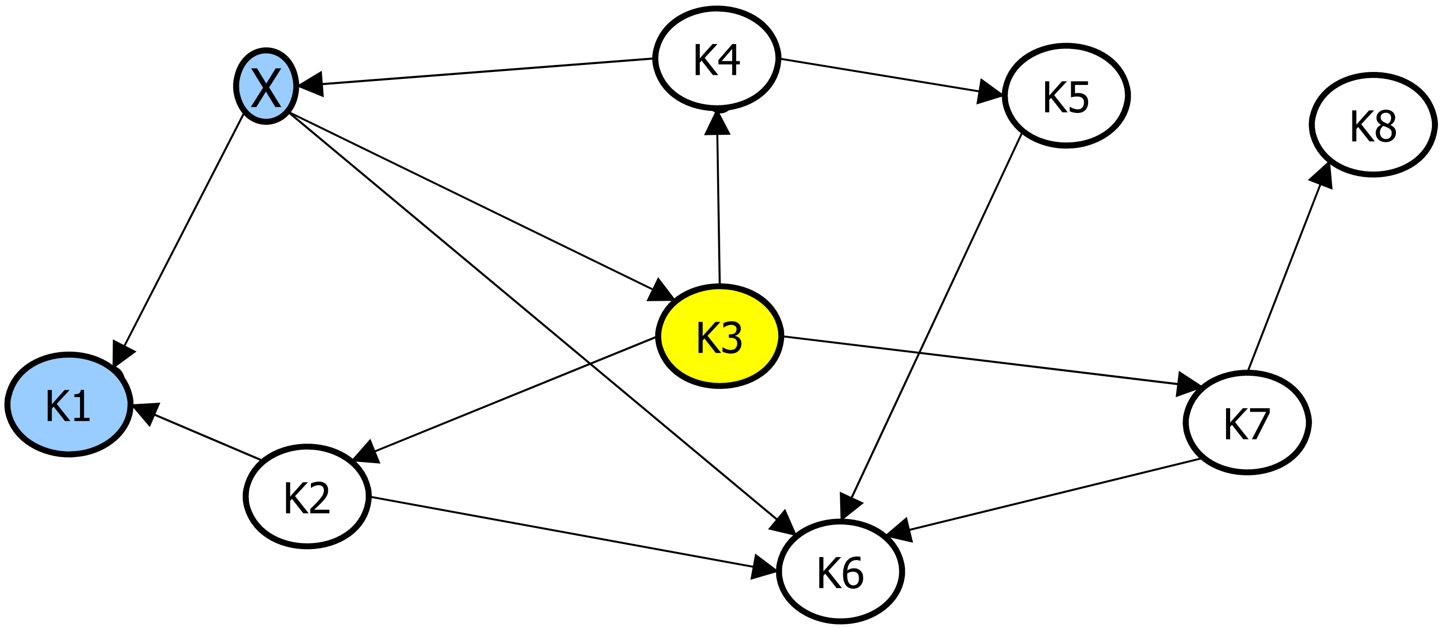
Example: Depth-first Traversal



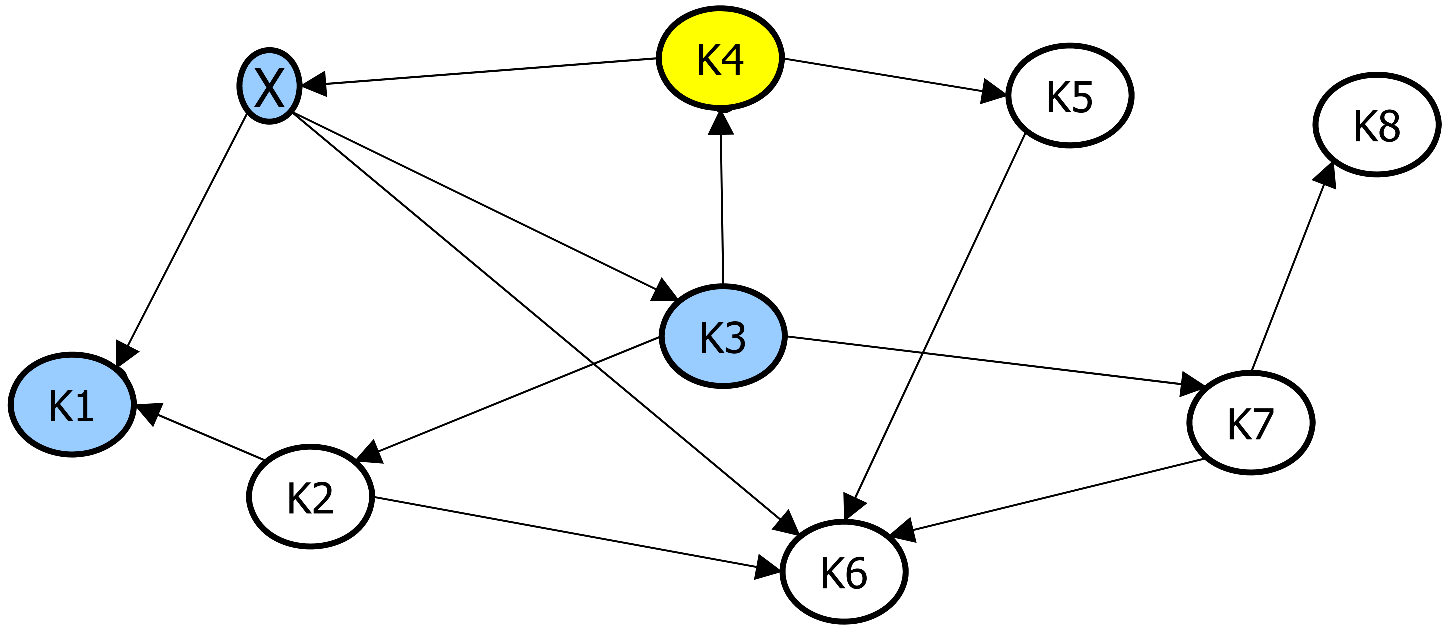
Example: Depth-first Traversal



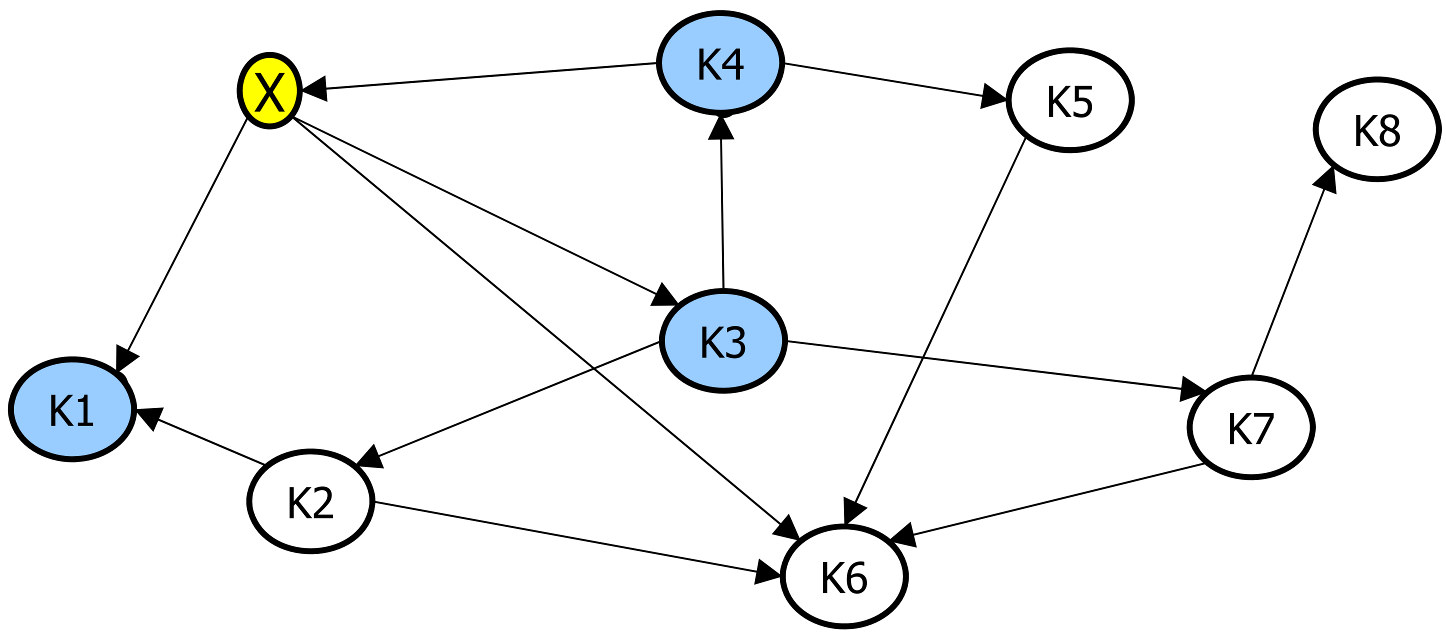
Example: Depth-first Traversal



Example: Depth-first Traversal



Example: Depth-first Traversal



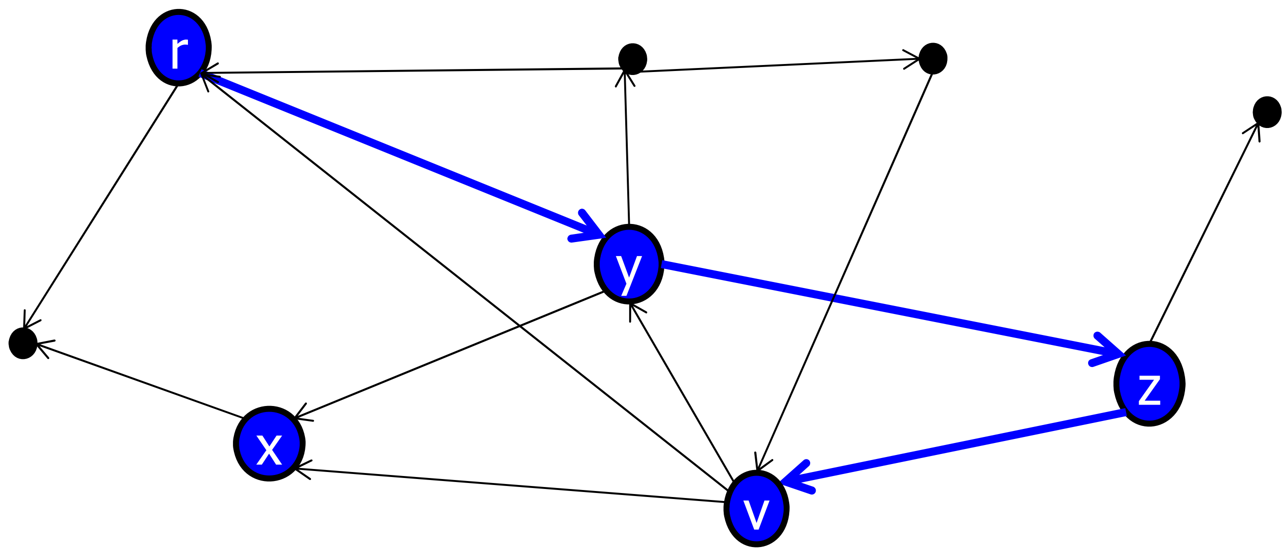
- Problem:
  - We have to take care of cycles
  - No root – where should we start?

Breaking Cycles

- Naïve traversal will usually visit nodes more than once
  - If there is at least one node with more than one incoming edge
- Naïve traversal might run into infinite loops
  - If the graph contains at least one cycle (is cyclic)

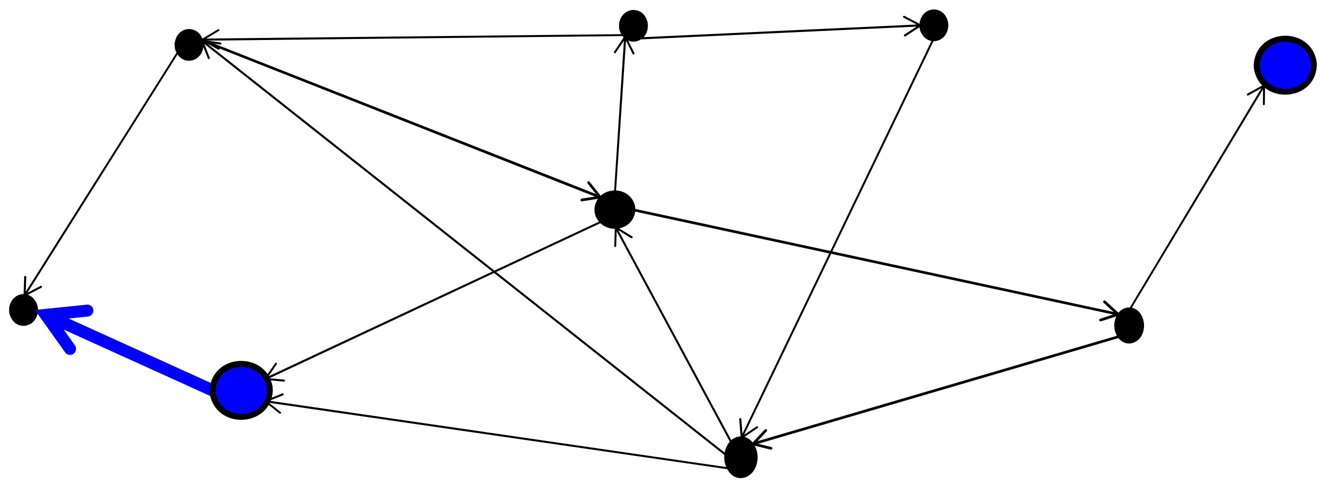
- Breaking cycles / avoiding multiple visits
  - Assume we started the traversal at a node r
  - During traversal, we keep a list S of already visited nodes
  - Assume we are in v and aim to proceed to v' using e=(v, v')∈E
  - If v'∈S, v' was visited before and we are about to run into a cycle
  - In this case, e is ignored

Example



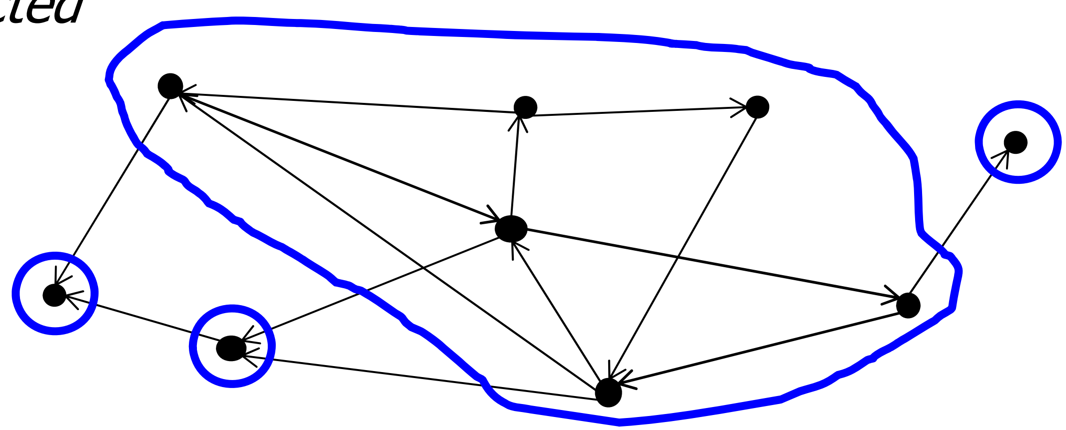
- Started at r and went S={r, y, z, v}
- Testing (v,y): y∈S, drop
- Testing (v, r): r∈S, drop
- Testing (v, x): x∉S, proceed

Where do we Start?



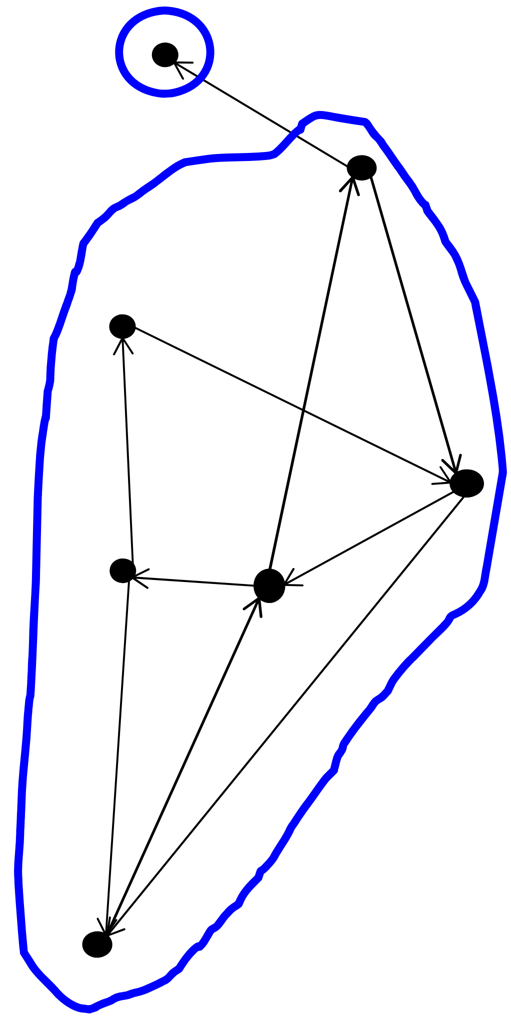
Where do we Start?

- Definition
  Let G=(V, E) and let G′ be the subgraph of G induced by some V′⊆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*
  - *Any maximal connected subgraph of G is called a connected component of G, if G is undirected, and a strongly connected component, if G is directed*



Where do we Start?

- If an 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 directed graph 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 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)
{
  s := new Stack();
  s.put( v);
  while not s.isEmpty() do
    n := s.get();
    print n;   # Do something
    U := U \ {n};
    S := S ∪ {n};
    c := n.outgoingNodes();
    foreach x in c do
      if x∈U then
        s.put( x);
      end if;
    end for;
  end while;
}
```

Analysis

- We have every node exactly once on the stack
  - Once visited, never visited again
- We look at every edge exactly once
  - Outgoing edges of every visited node are never considered again
- Altogether: O(n+m)

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