Week 07

Things to Note ...

- Fun quiz next week
- Mid-term exam in week 9 (problem sets 3-8 are a good preparation)

In This Lecture ...

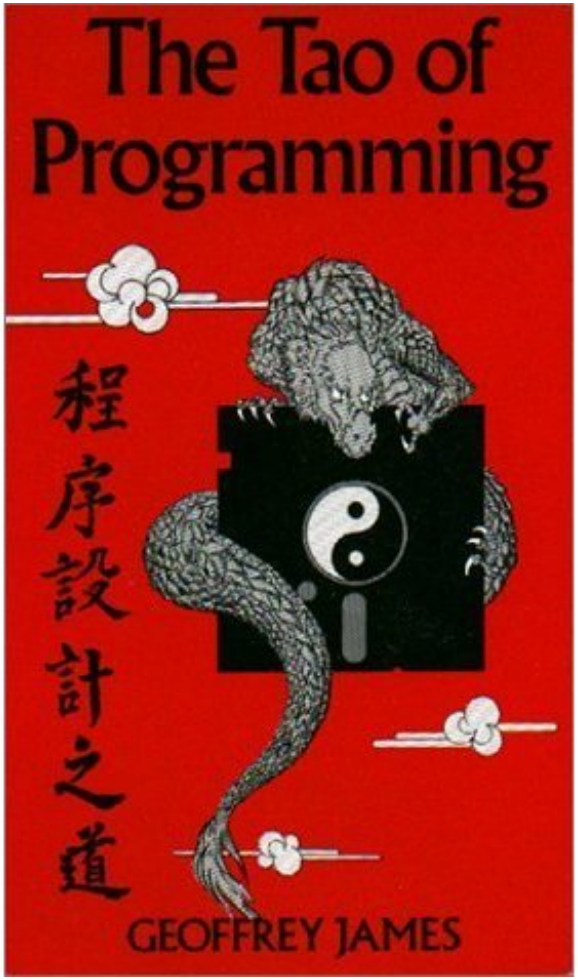
- Graph algorithms: graph search ([S] 17.7,18.1-18.3,18.7,19.1-19.3)

Coming Up ...

- More graph algorithms: spanning trees, minimal paths ([S] Ch.20-21)

The Tao of Programming

Next in a series of advices from the Tao of Programming …



The Tao of Programming (cont)

Book 3
Chapter 3.2

There once was a Master Programmer who wrote unstructured programs.
A novice programmer, seeking to imitate him, also began to write unstructured programs.
When the novice asked the Master to evaluate his progress, the Master criticized him for writing unstructured programs, saying,

"What is appropriate for the Master is not appropriate for the novice. You must understand Tao before transcending structure."

Graph Algorithms

Problems on Graphs

What kind of problems do we want to solve on/via graphs?

- is the graph fully-connected?
- can we remove an edge and keep it fully-connected?
- is one vertex reachable starting from some other vertex?
- what is the cheapest cost path from *v* to *w*?
- which vertices are reachable from *v*? (transitive closure)
- is there a cycle that passes through all vertices? (circuit)
- is there a tree that links all vertices? (spanning tree)
- what is the minimum spanning tree?
- …
- can a graph be drawn in a plane with no crossing edges? (planar graphs)
- are two graphs "equivalent"? (isomorphism)

Graph Algorithms

In this course we examine algorithms for

- connectivity (simple graphs)
- path finding (simple/directed graphs)
- minimum spanning trees (weighted graphs)
- shortest path (weighted graphs)

and look at generic methods for traversing graphs.

We begin with one of the simplest graph traversals …

Graph Traversal

Finding a Path

Questions on paths:

- is there a path between two given vertices (*src*,*dest*)?
- what is the sequence of vertices from *src* to *dest*?

Approach to solving problem:

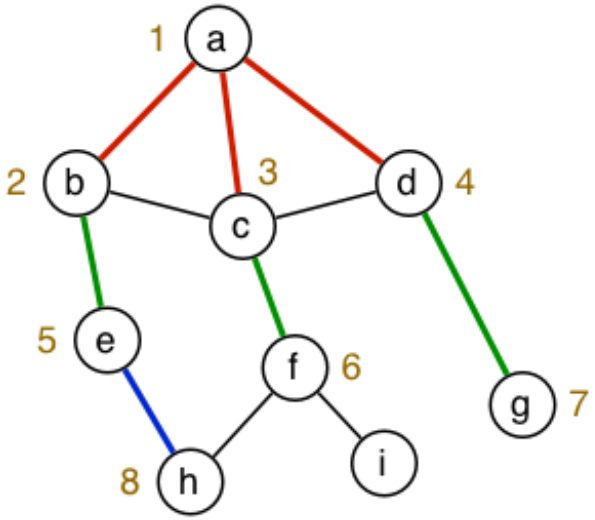
- examine vertices adjacent to *src*
- if any of them is *dest*, then done
- otherwise try vertices two edges from *v*
- repeat looking further and further from *v*

Two strategies for graph traversal/search: depth-first, breadth-first

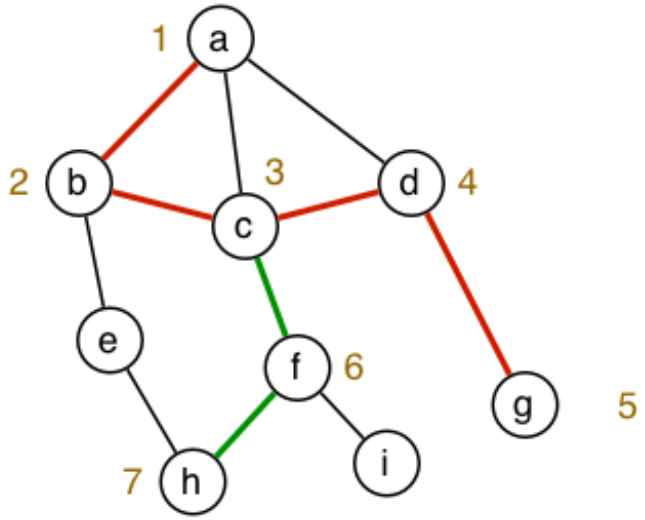
- DFS follows one path to completion before considering others
- BFS "fans-out" from the starting vertex ("spreading" subgraph)

Finding a Path (cont)

Comparison of BFS/DFS search for checking if there is a path from *a* to *h* …



Breadth-first Search



Depth-first Search

Both approaches ignore some edges by remembering previously visited vertices.

Finding a Path (cont)

Depth-first …

- favour following path rather than neighbours
- can be implemented recursively or iteratively (via stack)
- full traversal produces a depth-first spanning tree

Breadth-first …

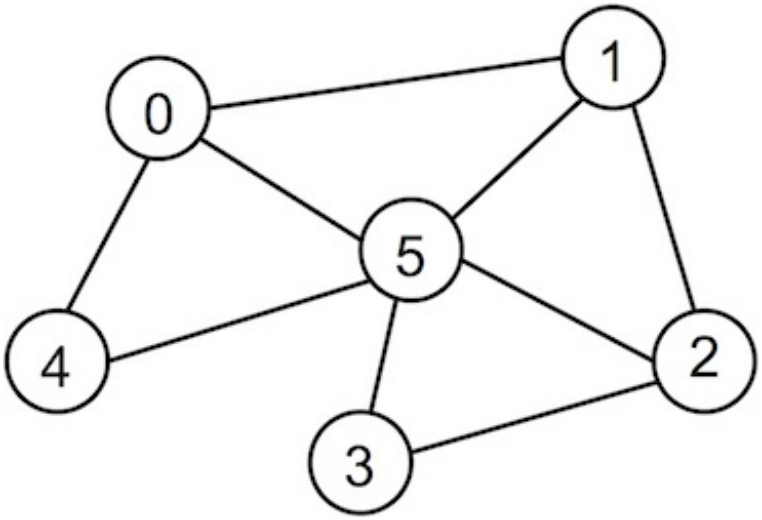
- favour neighbours rather than path following
- can be implemented iteratively (via queue)
- full traversal produces a breadth-first spanning tree

Exercise #1: Traversal-induced Spanning Trees

A spanning tree of a graph

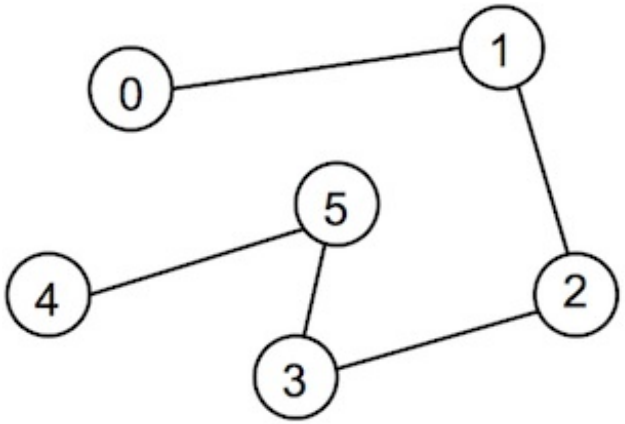
- includes all vertices
- uses a subset of edges, without cycles

Show the DFS and BFS spanning trees for the graph below, starting with 0:

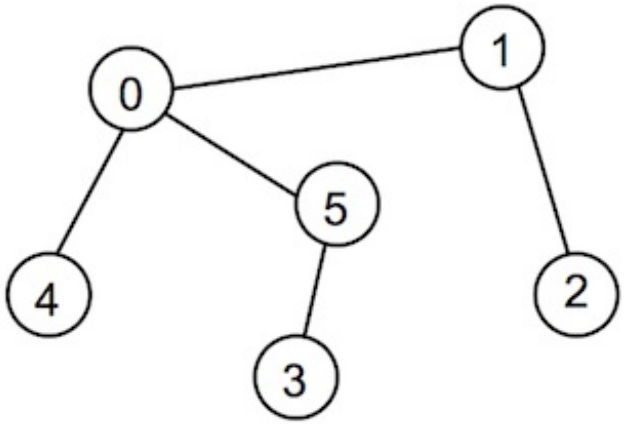


Consider neighbours in ascending order

Answer:



DFS spanning tree



BFS spanning tree

Depth-first Search

Depth-first search can be described recursively as

`depthFirst(G,v):`

1. mark **`v`** as visited
2. for each **`(v,w)`**$\in$*edges*(G) do
   if **`w`** has not been visitied then
   `depthFirst(w)`

The recursion induces backtracking

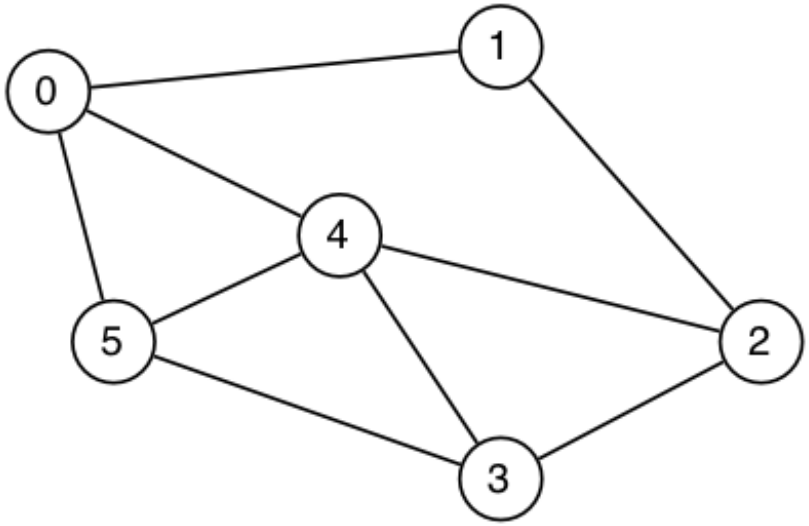
Depth-first Search (cont)

Recursive DFS path checking

```
hasPath(G,src,dest):
|   Input  graph G, vertices src,dest
|   Output true if there is a path from src to dest in G,
|          false otherwise
|
|   return dfsPathCheck(G,src,dest)

dfsPathCheck(G,v,dest):
|   mark v as visited
|   for all (v,w)∈edges(G) do
|   |   if w=dest then          // found edge to dest
|   |      return true
|   |   else if w has not been visited then
|   |      if dfsPathCheck(G,w,dest) then
|   |         return true      // found path via w to dest
|   |      end if
|   |   end if
|   end for
|   return false               // no path from v to dest
```

Exercise #2: Depth-first Traversal (i)

Trace the execution of **`dfsPathCheck(G,0,5)`** on:



Consider neighbours in ascending order

Answer:

0 - 1 - 2 - 3 - 4 - 5

Depth-first Search (cont)

Cost analysis:

- each vertex visited at most once ⇒ cost = *O(V)*
- visit all edges incident on visited vertices ⇒ cost = *O(E)*
  - assuming an adjacency list representation

Time complexity of DFS: *O(V+E)* (adjacency list representation)

Depth-first Search (cont)

Knowing whether a path exists can be useful

Knowing what the path is even more useful
⇒ record the previously visited node as we search through the graph
(so that we can then trace path through graph)

Make use of global variable:

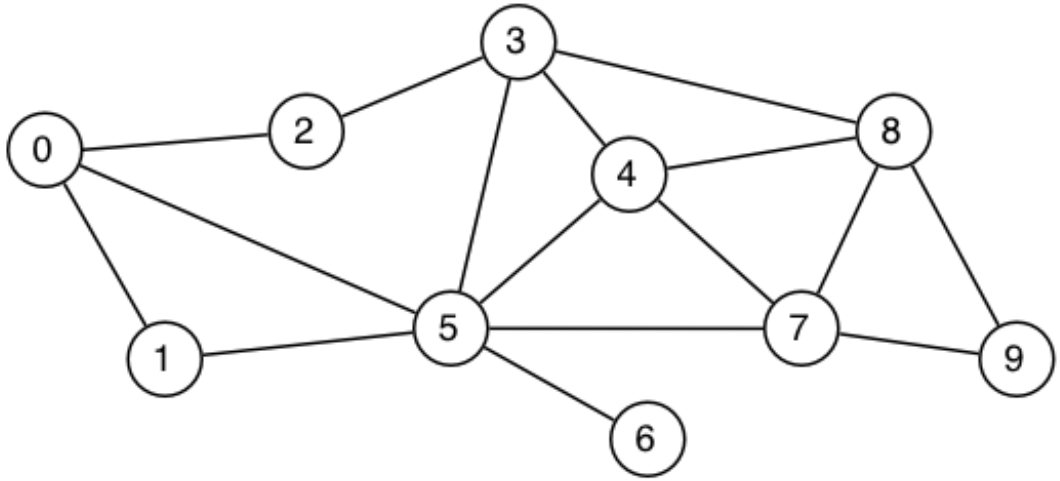
- **`visited[]`** … array to store previously visited node, for each node being visited

Depth-first Search (cont)

```
visited[] // store previously visited node, for each vertex 0..nV-1

findPath(G,src,dest):
|   Input graph G, vertices src,dest
|
|   for all vertices v∈G do
|      visited[v]=-1
|   end for
|   visited[src]=src                        // starting node of the path
|   if dfsPathCheck(G,src,dest) then        // show path in dest..src order
|   |  v=dest
|   |  while v≠src do
|   |     print v"-"
|   |     v=visited[v]
|   |  end while
|   |  print src
|   end if

dfsPathCheck(G,v,dest):
|   for all (v,w)∈edges(G) do
|   |  if visited[w]=-1 then
|   |     visited[w]=v
|   |     if w=dest then                    // found edge from v to dest
|   |        return true
|   |     else if dfsPathCheck(G,w,dest) then
|   |        return true                    // found path via w to dest
|   |     end if
|   |  end if
|   end for
|   return false                            // no path from v to dest
```

Exercise #3: Depth-first Traversal (ii)

Show the DFS order in which we visit vertices in this graph when searching for a path from 0 to 6:



visited	?	?	?	?	?	?	?	?	?	?
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Consider neighbours in ascending order

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

Path: 6-5-1-0

Depth-first Search (cont)

DFS can also be described non-recursively (via a stack):

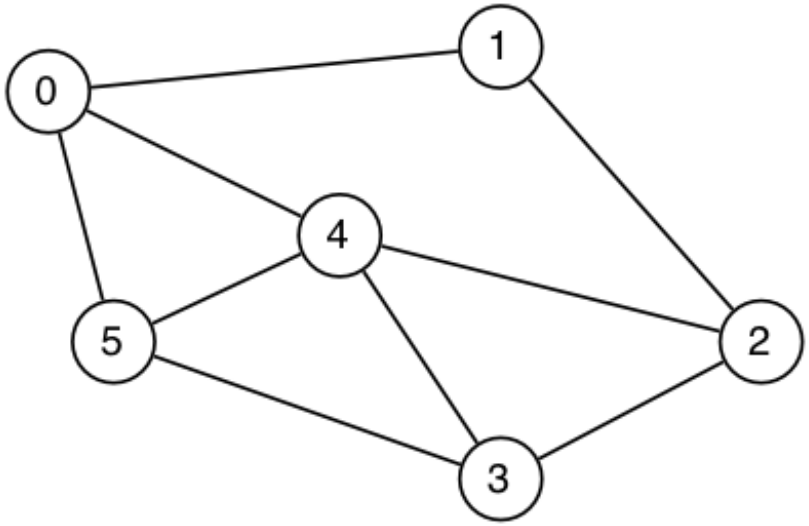
```
visited[] // array of visiting orders, indexed by vertex 0..nV-1

findPathDFS(G,src,dest):
|   Input graph G, vertices src,dest
|
|   for all vertices v∈G do
|      visited[v]=-1
|   end for
|   found=false
|   visited[src]=src
|   push src onto new stack s
|   while ¬found ∧ s is not empty do
|   |   pop v from s
|   |   for each (v,w)∈edges(G) do
|   |   |   if visited[w]=-1 then
|   |   |      visited[w]=v
|   |   |      if w=dest then
|   |   |         found=true
|   |   |      else
|   |   |         push w onto s
|   |   |      end if
|   |   |   end if
|   |   end for
|   end while
|   if found then
|      display path in dest..src order
|   end if
```

Uses standard stack operations (push, pop, check if empty)

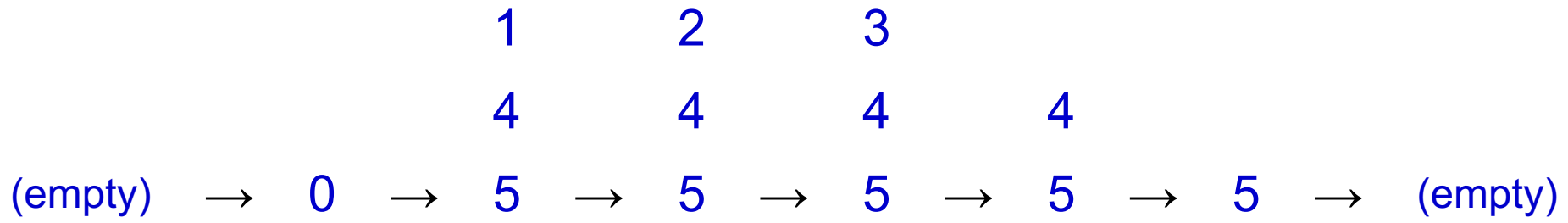
Time complexity is the same: *O(V+E)* (each vertex added to stack once, each element in vertex's adjacency list visited once)

Exercise #4: Depth-first Traversal (iii)

Show how the stack evolves when executing **`findPathDFS(g,0,5)`** on:



Push neighbours in descending order … so they get popped in ascending order


1
2
3
4
4
4
4
(empty)
0
5
5
5
5
5
(empty)

Breadth-first Search

Basic approach to breadth-first search (BFS):

- visit and mark current vertex
- visit all neighbours of current vertex
- then consider neighbours of neighbours

Notes:

- tricky to describe recursively
- a minor variation on non-recursive DFS search works
  ⇒ switch the *stack* for a *queue*

Breadth-first Search (cont)

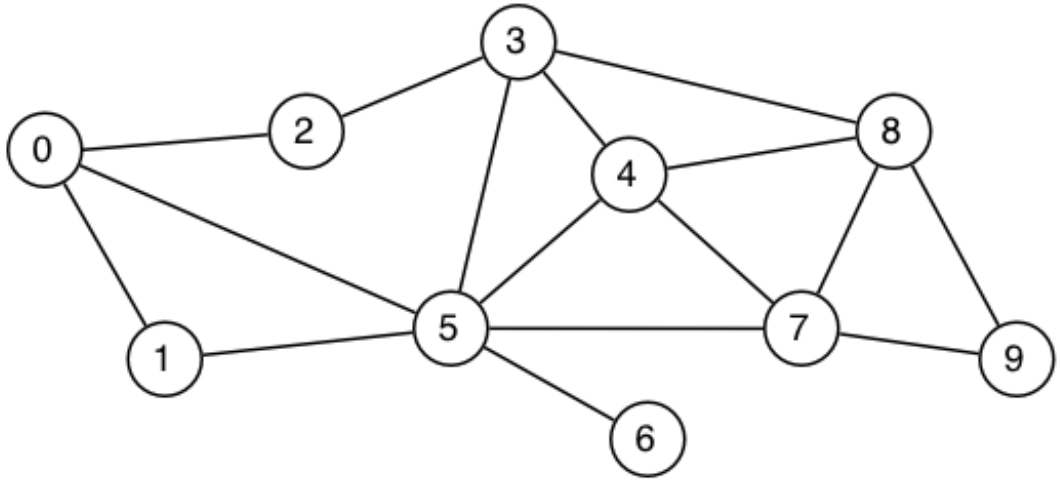
BFS algorithm (records visiting order):

```
visited[] // array of visiting orders, indexed by vertex 0..nV-1

findPathBFS(G,src,dest):
|   Input  graph G, vertices src,dest
|
|   for all vertices v∈G do
|      visited[v]=-1
|   end for
|   found=false
|   visited[src]=src
|   enqueue src into new queue q
|   while ¬found ∧ q is not empty do
|   |   dequeue v from q
|   |   for each neighbour w of v do
|   |   |   if visited[w]=-1 then
|   |   |      visited[w]=v
|   |   |      if w=dest then
|   |   |         found=true
|   |   |      else
|   |   |         enqueue w into q
|   |   |      end if
|   |   |   end if
|   |   end for
|   end while
|   if found then
|      display path in dest..src order
|   end if
```

Uses standard queue operations (enqueue, dequeue, check if empty)

Exercise #5: Breadth-first Traversal

Show the BFS order in which we visit vertices in this graph when searching for a path from 0 to 6:



visited	?	?	?	?	?	?	?	?	?	?
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Consider neighbours in ascending order

0	0	0	2	5	0	5	5	-1	-1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Path: 6-5-0

Breadth-first Search (cont)

Time complexity of BFS: *O(V+E)* (same as DFS)

BFS finds a "shortest" path

- based on minimum # edges between *src* and *dest*.
- stops with first-found path, if there are multiple ones

In many applications, edges are weighted and we want path

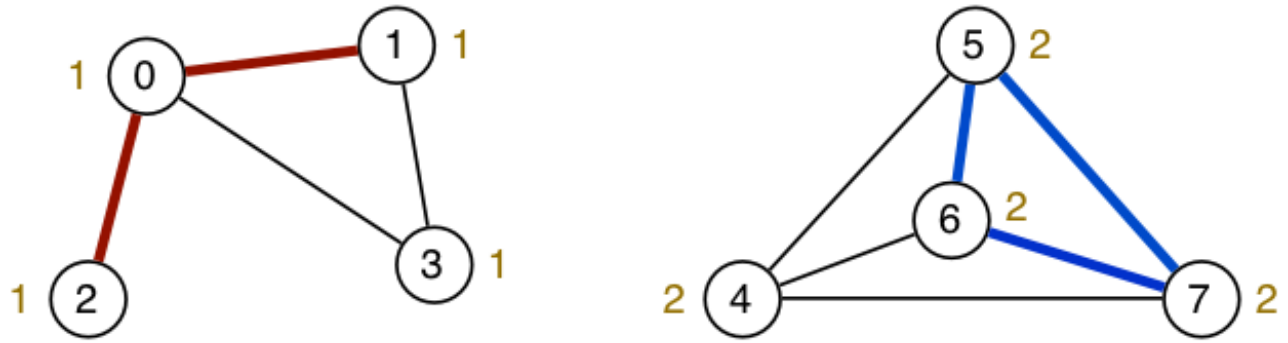
- based on minimum sum-of-weights along path *src* .. *dest*

We discuss weighted/directed graphs later.

Other DFS Examples

Other problems to solve via DFS graph search

- checking for the existence of a cycle
- determining which connected component each vertex is in



Graph with two connected components, a path and a cycle

Exercise #6: Buggy Cycle Check

A graph has a cycle if

- it has a path of length > 1
- with start vertex *src* = end vertex *dest*
- and without using any edge more than once

We are not required to give the path, just indicate its presence.

The following DFS cycle check has two bugs. Find them.

```
hasCycle(G):
|   Input  graph G
|   Output true if G has a cycle, false otherwise
|
|   choose any vertex v∈G
|   return dfsCycleCheck(G,v)

dfsCycleCheck(G,v):
|   mark v as visited
|   for each (v,w)∈edges(G) do
|   |   if w has been visited then    // found cycle
|   |      return true
|   |   else if dfsCycleCheck(G,w) then
|   |      return true
|   end for
|   return false                      // no cycle at v
```

1. Only one connected component is checked.
2. The loop

```
for each (v,w)∈edges(G) do
```

should exclude the neighbour of v from which you just came, so as to prevent a single edge *w-v* from being classified as a cycle.

Computing Connected Components

Problems:

- how many connected subgraphs are there?
- are two vertices in the same connected subgraph?

Both of the above can be solved if we can

- build an array, one element for each vertex V
- indicating which connected component V is in
- **`componentOf[]`** … array [0..nV-1] of component IDs

Computing Connected Components (cont)

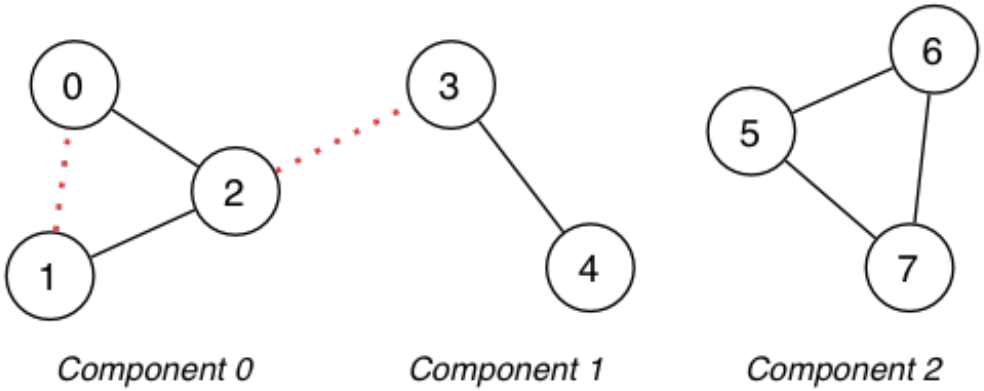
Algorithm to assign vertices to connected components:

```
components(G):
|   Input graph G
|
|   for all vertices v∈G do
|      componentOf[v]=-1
|   end for
|   compID=0
|   for all vertices v∈G do
|   |   if componentOf[v]=-1 then
|   |      dfsComponents(G,v,compID)
|   |      compID=compID+1
|   |   end if
|   end for

dfsComponents(G,v,id):
|   componentOf[v]=id
|   for all vertices w adjacent to v do
|      if componentOf[w]=-1 then
|         dfsComponents(G,w,id)
|      end if
|   end for
```

Exercise #7: Connected components

Trace the execution of the algorithm

1. on the graph shown below
2. on the same graph but with the dotted edges added



Consider neighbours in ascending order

1.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
-1	-1	-1	-1	-1	-1	-1	-1
0	-1	-1	-1	-1	-1	-1	-1
0	-1	0	-1	-1	-1	-1	-1
0	0	0	-1	-1	-1	-1	-1
0	0	0	1	-1	-1	-1	-1
…							
0	0	0	1	1	2	2	2

2.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
-1	-1	-1	-1	-1	-1	-1	-1
0	-1	-1	-1	-1	-1	-1	-1
0	0	-1	-1	-1	-1	-1	-1
0	0	0	-1	-1	-1	-1	-1
…							
0	0	0	0	0	1	1	1

Computing Connected Components (cont)

Consider an application where connectivity is critical

- we frequently ask questions of the kind above
- but we cannot afford to run **`components()`** each time

Add a new fields to the **`GraphRep`** structure:

```
typedef struct GraphRep *Graph;

struct GraphRep {
    ...
    int nC;   // # connected components
    int *cc;  // which component each vertex is contained in
    ...       // i.e. array [0..nV-1] of 0..nC-1
}
```

Computing Connected Components (cont)

With this structure, the above tasks become trivial:

```
// How many connected subgraphs are there?
int nConnected(Graph g) {
   return g->nC;
}
// Are two vertices in the same connected subgraph?
bool inSameComponent(Graph g, Vertex v, Vertex w) {
   return (g->cc[v] == g->cc[w]);
}
```

Consider maintenance of such a graph representation:

- initially, **`nC = nV`** (because no edges)
- adding an edge may reduce **`nC`**
- removing an edge may increase **`nC`**
- **`cc[]`** can simplify path checking
  (ensure **`v,w`** are in same component before starting search)

Additional maintenance cost amortised by reduced cost for **`nConnected()`** and **`inSameComponent()`**

Hamiltonian and Euler Paths

Hamiltonian Path and Circuit

Hamiltonian path problem:

- find a simple path connecting two vertices *v,w* in graph *G*
- such that the path includes each vertex exactly once

If *v* = *w*, then we have a Hamiltonian circuit

Simple to state, but difficult to solve (*NP*-complete)

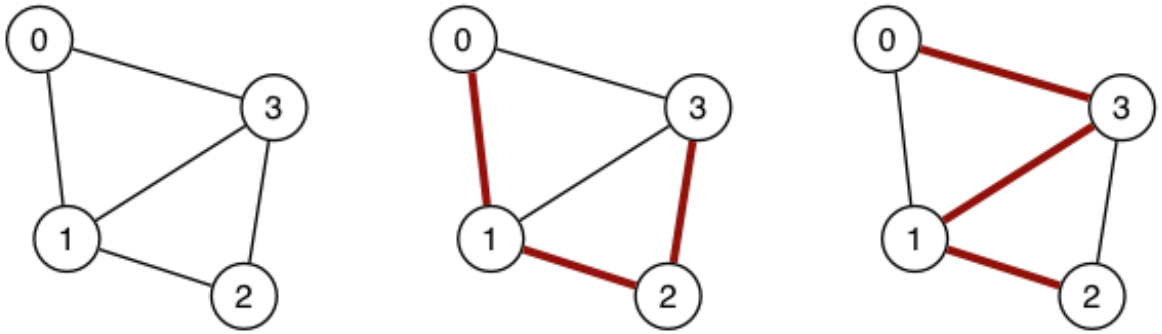
Many real-world applications require you to visit all vertices of a graph:

- Travelling salesman
- Bus routes
- …

Problem named after Irish mathematician, physicist and astronomer Sir William Rowan Hamilton (1805 - 1865)

Hamiltonian Path and Circuit (cont)

Graph and two possible Hamiltonian paths:



Hamiltonian Path and Circuit (cont)

Approach:

- generate all possible simple paths (using e.g. DFS)
- keep a counter of vertices visited in current path
- stop when find a path containing *V* vertices

Can be expressed via a recursive DFS algorithm

- similar to simple path finding approach, except
  - keeps track of path length; succeeds if length = *v*
  - resets "visited" marker after unsuccessful path

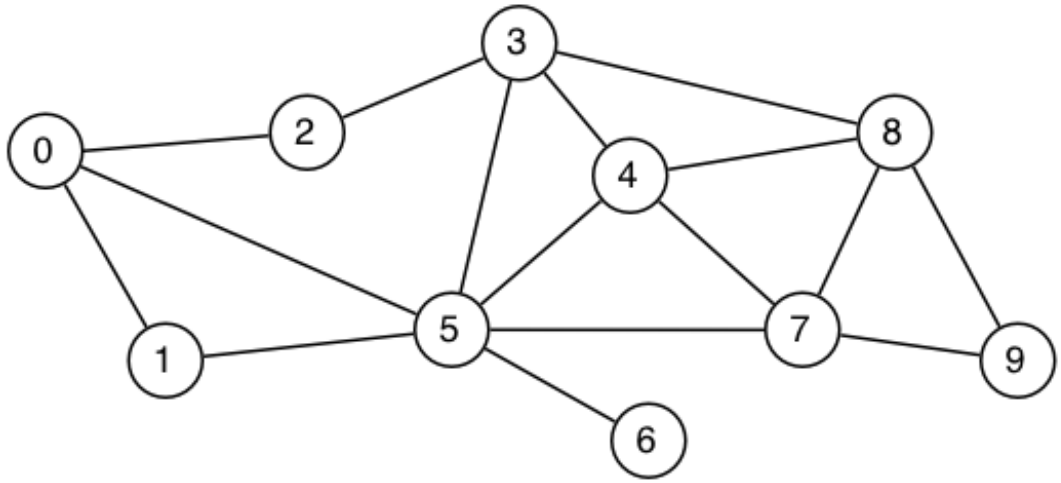
Hamiltonian Path and Circuit (cont)

Algorithm for finding Hamiltonian path:

```
visited[] // array [0..nV-1] to keep track of visited vertices

hasHamiltonianPath(G,src,dest):
|   for all vertices v∈G do
|      visited[v]=false
|   end for
|   return hamiltonR(G,src,dest,#vertices(G)-1)

hamiltonR(G,v,dest,d):
|   Input G    graph
|         v    current vertex considered
|         dest destination vertex
|         d    distance "remaining" until path found
|
|   if v=dest then
|      if d=0 then return true else return false
|   else
|   |  visited[v]=true
|   |  for each (v,w)∈edges(G) ∧ ¬visited[w] do
|   |     if hamiltonR(G,w,dest,d-1) then
|   |        return true
|   |     end if
|   |  end for
|   end if
|   visited[v]=false              // reset visited mark
|   return false
```

Exercise #8: Hamiltonian Path

Trace the execution of the algorithm when searching for a Hamiltonian path from 1 to 6:



Consider neighbours in ascending order

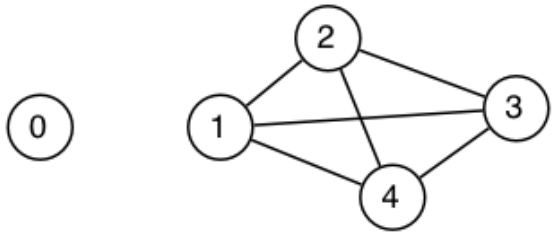
1-0-2-3-4-5-6	d≠0
1-0-2-3-4-5-7-8-9	no unvisited neighbour
1-0-2-3-4-5-7-9-8	no unvisited neighbour
1-0-2-3-4-7-5-6	d≠0
1-0-2-3-4-7-8-9	no unvisited neighbour
1-0-2-3-4-7-9-8	no unvisited neighbour
1-0-2-3-4-8-7-5-6	d≠0
1-0-2-3-4-8-7-9	no unvisited neighbour
1-0-2-3-4-8-9-7-5-6	✓

Repeat on your own with **`src=0`** and **`dest=6`**

Hamiltonian Path and Circuit (cont)

Analysis: worst case requires *(V-1)!* paths to be examined

Consider a graph with isolated vertex and the rest fully-connected



Checking **`hasHamiltonianPath(g,0,`***x*`)`** for any *x*

- requires us to consider every possible path
- e.g 1-2-3-4, 1-2-4-3, 1-3-2-4, 1-3-4-2, 1-4-2-3, …
- starting from any *x*, there are 3! paths ⇒ 4! total paths
- there is no path of length 5 in these *(V-1)!* possibilities

There is no known simpler algorithm for this task ⇒ *NP*-hard.

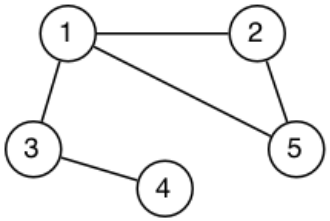
Note, however, that the above case could be solved in constant time if we had a fast check for 0 and *x* being in the same connected component

Euler Path and Circuit

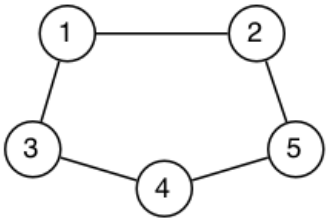
Euler path problem:

- find a path connecting two vertices *v,w* in graph *G*
- such that the path includes each edge exactly once
  (note: the path does not have to be simple ⇒ can visit vertices more than once)

If *v* = *w*, the we have an Euler circuit



Euler Path: 4-3-1-5-2-1



Euler Circuit: 1-2-5-4-3-1

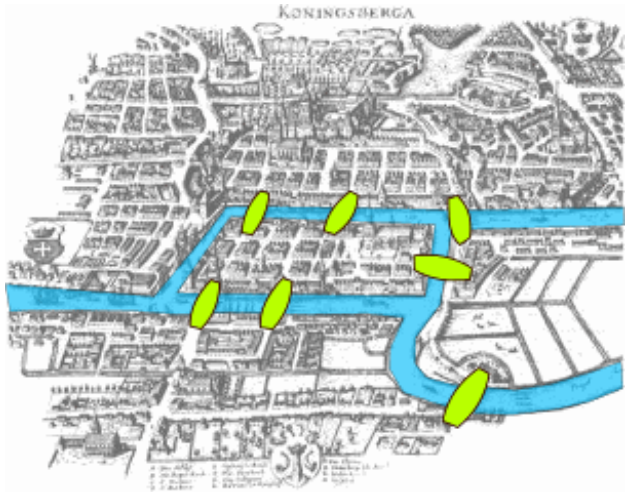
Many real-world applications require you to visit all edges of a graph:

- Postman
- Garbage pickup
- …

Euler Path and Circuit (cont)

Problem named after Swiss mathematician, physicist, astronomer, logician and engineer Leonhard Euler (1707 - 1783)

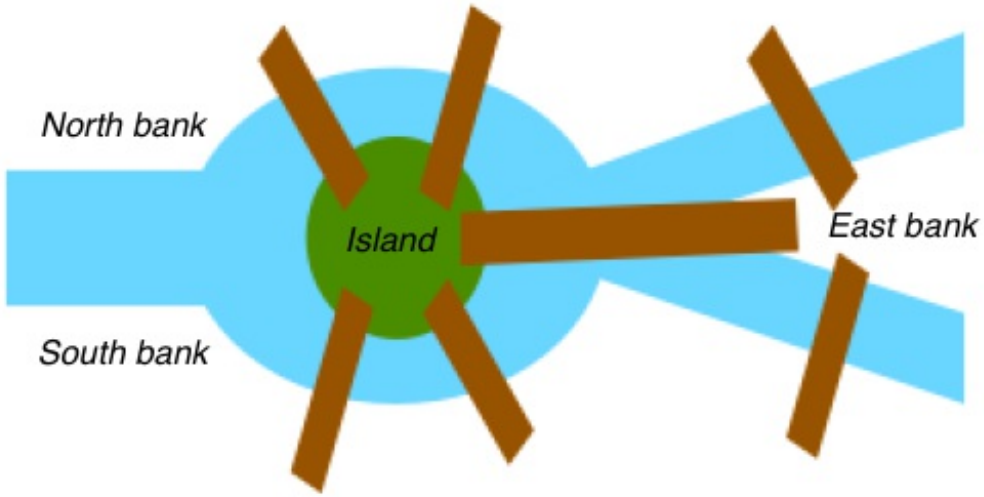
Based on a circuitous route via bridges in Konigsberg



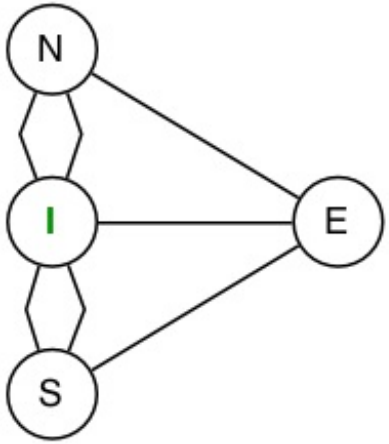
Exercise #9: Euler Path

Is there a way to cross all the bridges of Konigsberg exactly once on a walk through the town?

- treat land as nodes; bridges as edges



Bridges as schematic



Bridges as graph

No

Euler Path and Circuit (cont)

One possible "brute-force" approach:

- check for each path if it's an Euler path
- would result in factorial time performance

Can develop a better algorithm by exploiting:

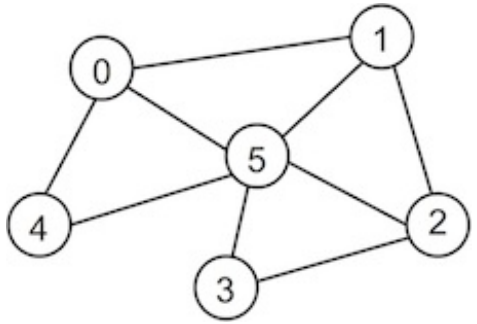
Theorem. A graph has an Euler circuit if and only if it is connected and all vertices have even degree

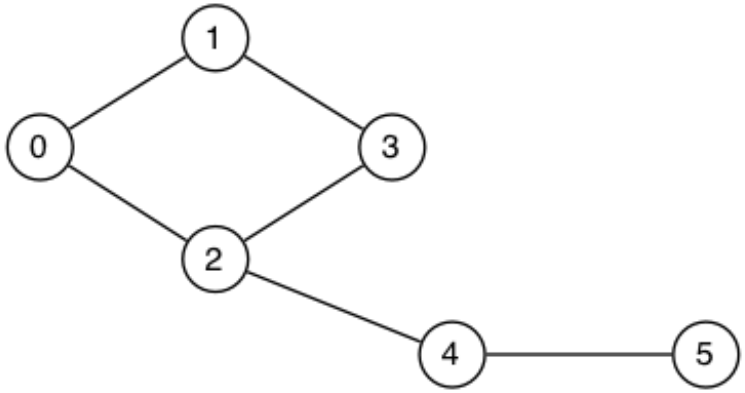
Theorem. A graph has a non-circuitous Euler path if and only if it is connected and exactly two vertices have odd degree

Exercise #10: Eulerian Graphs

Graphs with an Euler path are often called Eulerian Graphs

Which of these two graphs have an Euler path? an Euler circuit?





No Euler circuit

Only the second graph has an Euler path, e.g. 2-0-1-3-2-4-5

Euler Path and Circuit (cont)

Assume the existence of **`degree(g,v)`** (degree of a vertex, cf. problem set week 6)

Algorithm to check whether a graph has an Euler path:

```
hasEulerPath(G,src,dest):
|   Input  graph G, vertices src,dest
|   Output true if G has Euler path from src to dest
|          false otherwise
|
|   if src≠dest then
|      if degree(G,src) is even ∨ degree(G,dest) is even then
|         return false
|      end if
|   else if degree(G,src) is odd then
|      return false
|   end if
|   for all vertices v∈G do
|      if v≠src ∧ v≠dest ∧ degree(G,v) is odd then
|         return false
|      end if
|   end for
|   return true
```

Euler Path and Circuit (cont)

Analysis of **`hasEulerPath`** algorithm:

- assume that connectivity is already checked
- assume that degree is available via *O(1)* lookup
- single loop over all vertices ⇒ *O(V)*

If degree requires iteration over vertices

- cost to compute degree of a single vertex is *O(V)*
- overall cost is $O(V^2)$

⇒ problem tractable, even for large graphs (unlike Hamiltonian path problem)

For the keen, a linear-time (in the number of edges, *E*) algorithm to compute an Euler path is described in [Sedgewick] Ch.17.7.

Directed Graphs

Directed Graphs (Digraphs)

In our previous discussion of graphs:

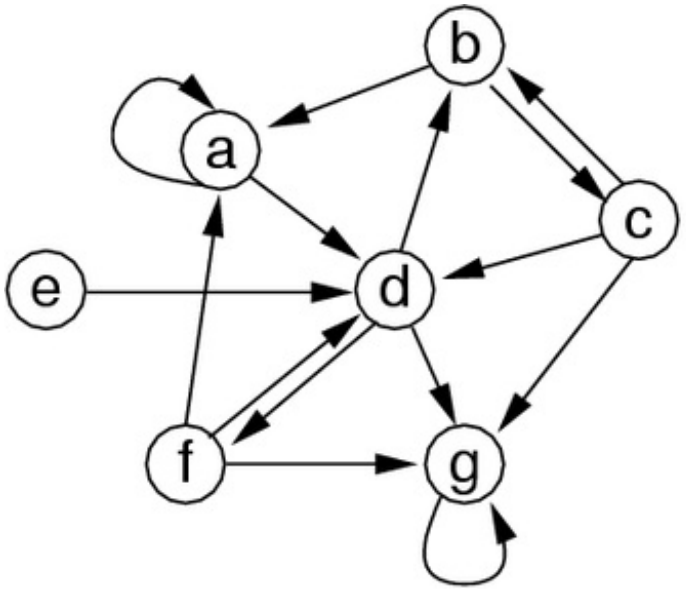
- an edge indicates a relationship between two vertices
- an edge indicates nothing more than a relationship

In many real-world applications of graphs:

- edges are directional ($v \rightarrow w \neq w \rightarrow v$)
- edges have a weight (cost to go from $v \rightarrow w$)

Directed Graphs (Digraphs) (cont)

Example digraph and adjacency matrix representation:



	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

Undirectional $\Rightarrow$ symmetric matrix
Directional $\Rightarrow$ non-symmetric matrix

Maximum #edges in a digraph with V vertices: V^2

Directed Graphs (Digraphs) (cont)

Terminology for digraphs …

Directed path: sequence of $n \geq 2$ vertices $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$

- where $(v_i, v_{i+1}) \in edges(G)$ for all v_i, v_{i+1} in sequence
- if $v_1 = v_n$, we have a directed cycle

Degree of vertex: $deg(v)$ = number of edges of the form $(v, \_) \in edges(G)$

- Indegree of vertex: $deg^{-1}(v)$ = number of edges of the form $(\_, v) \in edges(G)$

Reachability: w is reachable from v if $\exists$ directed path $v, \ldots, w$

Strong connectivity: every vertex is reachable from every other vertex

Directed acyclic graph (DAG): graph containing no directed cycles

Digraph Applications

Potential application areas:

Domain	Vertex	Edge
Web	web page	hyperlink
scheduling	task	precedence
chess	board position	legal move
science	journal article	citation
dynamic data	malloc'd object	pointer
programs	function	function call
`make`	file	dependency

Digraph Applications (cont)

Problems to solve on digraphs:

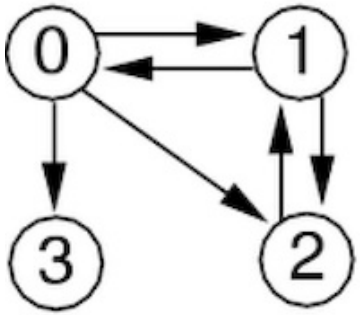
- is there a directed path from *s* to *t*? (transitive closure)
- what is the shortest path from *s* to *t*? (shortest path)
- are all vertices mutually reachable? (strong connectivity)
- how to organise a set of tasks? (topological sort)
- which web pages are "important"? (PageRank)
- how to build a web crawler? (graph traversal)

Digraph Representation

Similar set of choices as for undirectional graphs:

- array of edges (directed)
- vertex-indexed adjacency matrix (non-symmetric)
- vertex-indexed adjacency lists

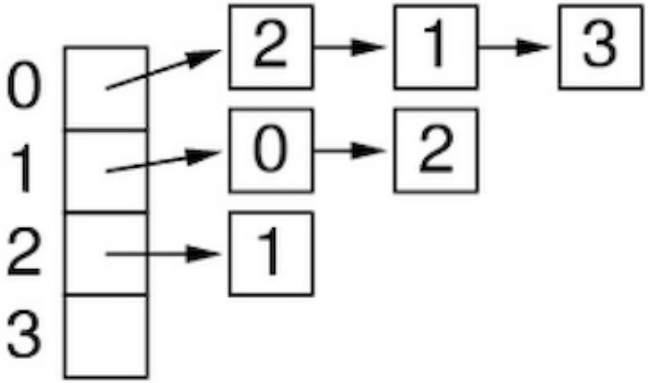
V vertices identified by *0 .. V-1*



digraph

	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	0	1	0	0
3	0	0	0	0

adj matrix



adj lists

Digraph Representation (cont)

Costs of representations: (where degree *deg(v)* = #edges leaving *v*)

	array of edges	adjacency matrix	adjacency list
space usage	*E*	V^2	*V+E*
insert edge	*E*	*1*	*1*
exists edge *(v,w)*?	*E*	*1*	*deg(v)*
get edges leaving *v*	*E*	*V*	*deg(v)*

Overall, adjacency list representation is best

- real graphs tend to be sparse
  (large number of vertices, small average degree *deg(v)*)
- algorithms frequently iterate over edges from *v*

Reachability

Transitive Closure

Given a digraph *G* it is potentially useful to know

- is vertex *t* reachable from vertex *s*?

Example applications:

- can I complete a schedule from the current state?
- is a malloc'd object being referenced by any pointer?

How to compute transitive closure?

Transitive Closure (cont)

One possibility:

- implement it via **`hasPath(G,s,t)`** (itself implemented by DFS or BFS algorithm)
- feasible if *reachable(G,s,t)* is infrequent operation

What if we have an algorithm that frequently needs to check rechability?

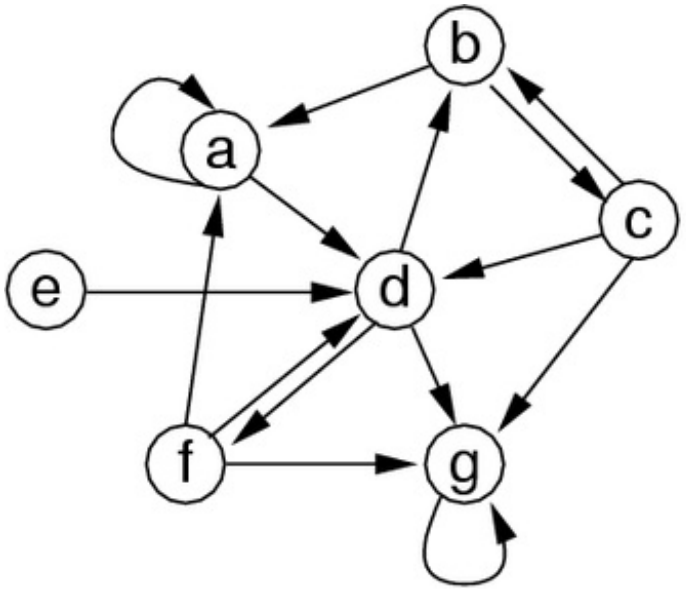
Would be very convenient/efficient to have:

```
reachable(G,s,t):
|   return G.tc[s][t]    // transitive closure matrix
```

Of course, if *V* is very large, then this is not feasible.

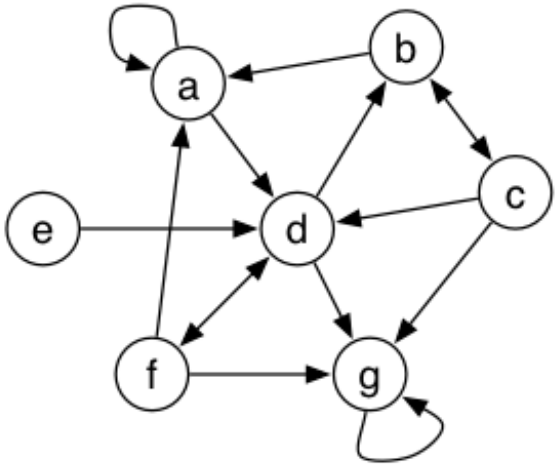
Exercise #11: Transitive Closure Matrix

Which reachable *s* .. *t* exist in the following graph?



	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

Transitive closure of example graph:



	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

adjacency matrix

	a	b	c	d	e	f	g
a	1	1	1	1	0	1	1
b	1	1	1	1	0	1	1
c	1	1	1	1	0	1	1
d	1	1	1	1	0	1	1
e	1	1	1	1	0	1	1
f	1	1	1	1	0	1	1
g	0	0	0	0	0	0	1

reachability matrix

Transitive Closure (cont)

Goal: produce a matrix of reachability values

- if *tc[s][t]* is 1, then *t* is reachable from *s*
- if *tc[s][t]* is 0, then *t* is not reachable from *s*

So, how to create this matrix?

Observation:

$\forall i,s,t \in \text{vertices}(G)$:
$(s,i) \in \text{edges}(G) \wedge (i,t) \in \text{edges}(G) \Rightarrow tc[s][t] = 1$

`tc[s][t]=1` if there is a path from *s* to *t* of length 2 $(s \rightarrow i \rightarrow t)$

Transitive Closure (cont)

If we implement the above as:

```
make tc[][] a copy of edges[][]
for all i∈vertices(G) do
   for all s∈vertices(G) do
      for all t∈vertices(G) do
         if tc[s][i]=1 ∧ tc[i][t]=1 then
            tc[s][t]=1
         end if
      end for
   end for
end for
```

then we get an algorithm to convert **edges** into a *tc*

This is known as Warshall's algorithm

Transitive Closure (cont)

How it works …

After iteration 1, `tc[s][t]` is 1 if

- either $s \rightarrow t$ exists or $s \rightarrow 0 \rightarrow t$ exists

After iteration 2, `tc[s][t]` is 1 if any of the following exist

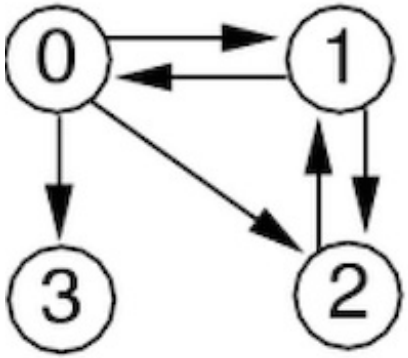
- $s \rightarrow t$ or $s \rightarrow 0 \rightarrow t$ or $s \rightarrow 1 \rightarrow t$ or $s \rightarrow 0 \rightarrow 1 \rightarrow t$ or $s \rightarrow 1 \rightarrow 0 \rightarrow t$

Etc. … so after the V^{th} iteration, `tc[s][t]` is 1 if

- there is any directed path in the graph from *s* to *t*

Exercise #12: Transitive Closure

Trace Warshall's algorithm on the following graph:



digraph

	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	0	1	0	0
3	0	0	0	0

adj matrix

1st iteration **`i=0`**:

tc	[0]	[1]	[2]	[3]
[0]	0	1	1	1
[1]	1	1	1	1
[2]	0	1	0	0
[3]	0	0	0	0

2nd iteration **`i=1`**:

tc	[0]	[1]	[2]	[3]
[0]	1	1	1	1
[1]	1	1	1	1
[2]	1	1	1	1
[3]	0	0	0	0

3rd iteration **`i=2`**: unchaged

4th iteration **`i=3`**: unchanged

Transitive Closure (cont)

Cost analysis:

- storage: additional V^2 items (each item may be 1 bit)
- computation of transitive closure: V^3
- computation of **`reachable()`**: *O(1)* after having generated **`tc[][]`**

Amortization: would need many calls to **`reachable()`** to justify other costs

Alternative: use DFS in each call to **`reachable()`**
Cost analysis:

- storage: cost of queue and set during reachable
- computation of **`reachable()`**: cost of DFS = $O(V^2)$ (for adjacency matrix)

Digraph Traversal

Same algorithms as for undirected graphs:

`depthFirst(v):`

1. mark **`v`** as visited
2. for each **`(v,w)`**$\in$*edges*(G) do
   if **`w`** has not been visitied then
   `depthFirst(w)`

`breadth-first(v):`

1. enqueue **`v`**
2. while queue not empty do
   dequeue **`v`**
   if **`v`** not already visited then
   mark **`v`** as visited
   enqueue each vertex **`w`** adjacent to **`v`**

Example: Web Crawling

Goal: visit every page on the web

Solution: breadth-first search with "implicit" graph

```
webCrawl(startingURL):
|   mark startingURL as alreadySeen
|   enqueue(Q,startingURL)
|   while isEmpty(Q) do
|   |   nextPage=dequeue(Q)
|   |   visit nextPage
|   |   for each hyperLink on nextPage do
|   |   |   if hyperLink not alreadySeen then
|   |   |      mark hyperLink as alreadySeen
|   |   |      enqueue(Q,hyperLink)
|   |   |   end if
|   |   end for
|   end while
```

`visit` scans page and collects e.g. keywords and links

PageRank

Goal: determine which Web pages are "important"

Approach: ignore page contents; focus on hyperlinks

- treat Web as graph: page = vertex, hyperlink = di-edge
- pages with many incoming hyperlinks are important
- need to computing "incoming degree" for vertices

Problem: the Web is a *very* large graph

- approx. 10^{14} pages, 10^{15} hyperlinks

Assume for the moment that we could build a graph …

Most frequent operation in algorithm "Does edge (v,w) exist?"

PageRank (cont)

Simple PageRank algorithm:

```
PageRank(myPage):
|   rank=0
|   for each page in the Web do
|   |   if linkExists(page,myPage) then
|   |      rank=rank+1
|   |   end if
|   end for
```

Note: requires inbound link check (not outbound as assumed above for cost of representation)

PageRank (cont)

V = # pages in Web, E = # hyperlinks in Web

Costs for computing PageRank for each representation:

Representation	linkExists(v,w)	Cost
Adjacency matrix	`edge[v][w]`	*1*
Adjacency lists	`inLL(list[v],w)`	$\cong E/V$

Not feasible ...

- adjacency matrix ... $V \cong 10^{14} \Rightarrow$ matrix has 10^{28} cells
- adjacency list ... V lists, each with $\cong$10 hyperlinks $\Rightarrow 10^{15}$ list nodes

So how to really do it?

PageRank (cont)

Approach: the random web surfer

- if we randomly follow links in the web …
- … more likely to re-discover pages with many inbound links

```
curr=random page, prev=null
for a long time do
|   if curr not in array ranked[] then
|      rank[curr]=0
|   end if
|   rank[curr]=rank[curr]+1
|   if random(0,100)<85 then              // with 85% chance ...
|      prev=curr
|      curr=choose hyperlink from curr    // ... crawl on
|   else
|      curr=random page                   // avoid getting stuck
|      prev=null
|   end if
end for
```

Could be accomplished while we crawl web to build search index

Exercise #13: Implementing Facebook

Facebook could be considered as a giant "social graph"

- what are the vertices?
- what are the edges?
- are edges directional?

What kind of algorithm would …

- help us find people that you might like to "befriend"?

Summary

- Graph traversal
  - depth-first search (DFS)
  - breadth-first search (BFS)
  - applications: path finding, connected components
- Hamiltonian paths/circuits, Euler paths/circuits
- Digraphs: representations, applications, reachability

- Suggested reading (Sedgewick):
  - Hamiltonian/Euler paths … Ch.17.7
  - Graph search … Ch.18.1-18.3,18.7
  - Digraphs … Ch.19.1-19.3