>>

Directed Graphs

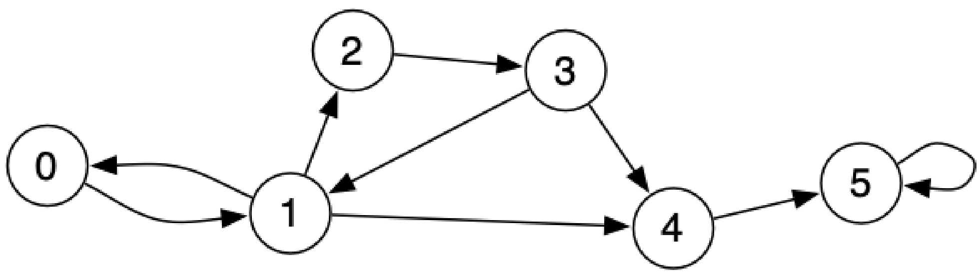
∧ >>

❖ Directed Graphs (Digraphs)

Reminder: directed graphs are ...

- graphs with *V* vertices, *E* edges *(v,w)*
- edge *(v,w)* has source *v* and destination *w*
- unlike undirected graphs, $v \rightarrow w \neq w \rightarrow v$

Example digraph:



<< ^ >>

❖ 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

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)

<< ∧ >>

❖ Transitive Closure

Problem: computing reachability (**`reachable(G,s,t)`**)

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?

<< ∧ >>

❖ ... Transitive Closure

One possibility to implement a reachability check:

- use **`hasPath(G,s,t)`** (itself implemented by DFS or BFS algorithm)
- feasible only if *reachable(G,s,t)* is an infrequent operation

What about applications that frequently check reachability?

Would be very convenient/efficient to have:

```
reachable(G,s,t)  ≡  G.tc[s][t]
```

`tc[][]` is called the transitive closure matrix

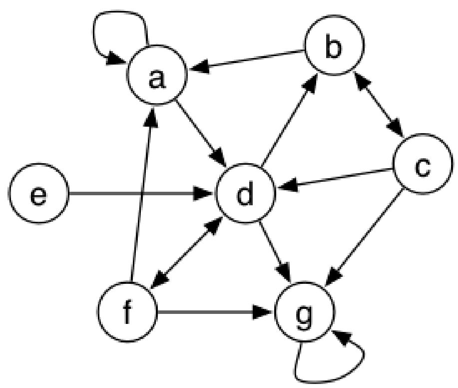
- **`tc[`***s***`][`***t***`]`** is 1 if there is a path from s to t, 0 otherwise

Of course, if V is large, then this may not be feasible either.

<< ∧ >>

❖ ... Transitive Closure

The **`tc[][]`** matrix shows all directed paths in the 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

Question: how to build **`tc[][]`** from **`edges[][]`**?

<< ∧ >>

❖ ... Transitive Closure

Goal: produce a matrix of reachability values

Observations:

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

In other words

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

<< ∧ >>

❖ ... Transitive Closure

Extending the above observations gives ...

An algorithm to convert **edges** into a *tc*

```
makeTC(G):
|  tc[][] = 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
```

This is known as Warshall's algorithm

<< ∧ >>

❖ ... Transitive Closure

How it works ...

After copying **`edges[][]`**, **`tc[s][t]`** is 1 if $s \to t$ exists

After first iteration (**`i=0`**), **`tc[s][t]`** is 1 if

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

After second interation (**`i=1`**), **`tc[s][t]`** is 1 if any of

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

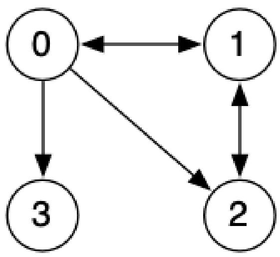
After the V^{th} iteration, **`tc[s][t]`** is 1 if

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

<< ∧ >>

❖ ... Transitive Closure

Tracing Warshall's algorithm on a simple graph:



Graph

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

Initially

	[0]	[1]	[2]	[3]
[0]	0	1	1	1
[1]	1	**1**	1	**1**
[2]	0	1	0	0
[3]	0	0	0	0

After first iteration

	[0]	[1]	[2]	[3]
[0]	**1**	1	1	1
[1]	1	0	1	0
[2]	**1**	1	**1**	**1**
[3]	0	0	0	0

After second iteration

No change on any following iterations

❖ ... Transitive Closure

Cost analysis:

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

Amortisation: need many calls to **`reachable()`** to justify setup cost

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

Cost analysis:

- storage: cost of Stack and Set during DFS calculation
- computation of **`reachable()`**: $O(V^2)$ (for adjacency matrix)

❖ Digraph Traversal

Same algorithms as for undirected graphs:

```
depthFirst(G,v):
|  mark v as visited
|  for each (v,w) ∈ edges(G) do
|  |  if w has not been visited then
|  |  |  depthFirst(w)
|  |  end if
|  end for

breadthFirst(G,v):
|  enqueue v
|  while queue not empty do
|  |  curr=dequeue
|  |  if curr not already visited then
|  |  |  mark curr as visited
|  |  |  enqueue each w where (curr,w) ∈ edges(G)
|  |  end if
|  end while
```

<< ∧ >>

❖ 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 not isEmpty(Q) do
|  |  currPage=dequeue(Q)
|  |  visit currPage
|  |  for each hyperLink on currPage 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^{10} pages, 10^{11} hyperlinks

❖ ... PageRank

Assume for the moment that we could build a graph ...

Naive 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 (normally, we check outbound)

<< ^ >>

❖ ... PageRank

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 ... $V \cong 10^{10} \Rightarrow$ matrix has 10^{20} cells
- adjacency list ... V lists, each with $\cong$10 hyperlinks $\Rightarrow 10^{11}$ list nodes

So how to really do it?

<< ∧ >>

❖ ... PageRank

The random web surfer strategy.

Each page typically has many outbound hyperlinks ...

- choose one at random, without a **`visited[]`** check
- follow link and repeat above process on destination page

If no visited check, need a way to (mostly) avoid loops

Important property of this strategy

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

<< ∧ >>

❖ ... PageRank

Random web surfer algorithm ...

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

<< ∧

❖ ... PageRank

Above is a very rough approximation to reality.

If you want the details ...

- The Anatomy of a Large-Scale Hypertextual Web Search Engine
  https://research.google/pubs/pub334/
- The PageRank Citation Ranking: Bringing Order to the Web
  http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

And the background ...

- Authoritative Sources in a Hyperlinked Environment
  https://dl.acm.org/doi/pdf/10.1145/324133.324140

Produced: 29 Jun 2020