Directed Graphs

❖ 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*? $s \to t$

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

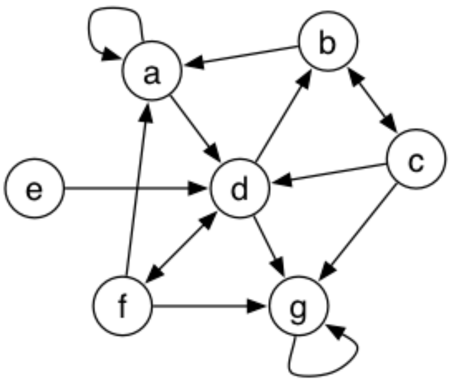
- **`tc[`***s***`][`***t***`]`** is 1 if there is a path from s to t, 0 otherwise

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

可用

❖ ... 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) $\Rightarrow$ $tc[s][t] = 1$ tc[s][t]=1

s → i → t

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 ($s \to i \to t$)

❖ ... Transitive Closure

Extending the above observations gives ...

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

tc[s][t] 已有

```
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 $O(V^3)$

❖ ... 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

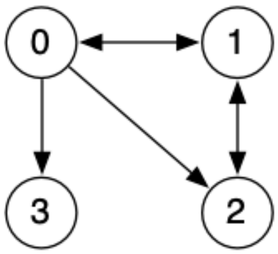
- $s \to t$ or $s \to 0 \to t$ or $s \to 1 \to t$ or $s \to 0 \to 1 \to t$ or $s \to 1 \to 0 \to t$

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
1→0

2→1
1→0

1→0
0→1
1→1

1→0
0→3
1→3

	[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	1	1	1
[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

Start hy① hy② hy③ hy①① . . .

❖ 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 ...

$10^{10} \times 10^{10}$

- 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 15
        curr=random page, not prev  // avoid getting stuck
        prev=null
    end if
end for
```

Shortest Path Algorithms

❖ Shortest Path

Path = sequence of edges in graph G

- $p = (v_0,v_1,weight_1), (v_1,v_2,weight_2), \ldots, (v_{m-1},v_m,weight_m)$

cost(path) = sum of edge weights along path

min cost

Shortest path between vertices *s* and *t*

- a simple path *p(s,t)* where *s = first(p)*, *t = last(p)*
- no other simple path *q(s,t)* has *cost(q) < cost(p)*

Assumptions: weighted digraph, no negative weights.

Applications: navigation, routing in data networks, ...

❖ ... Shortest Path

Some variations on shortest path (SP) ...

Source-target SP problem

- shortest path from source vertex s to target vertex t

Single-source SP problem

- set of shortest paths from source vertex s to all other vertices

All-pairs SP problems

- set of shortest paths between all pairs of vertices s and t

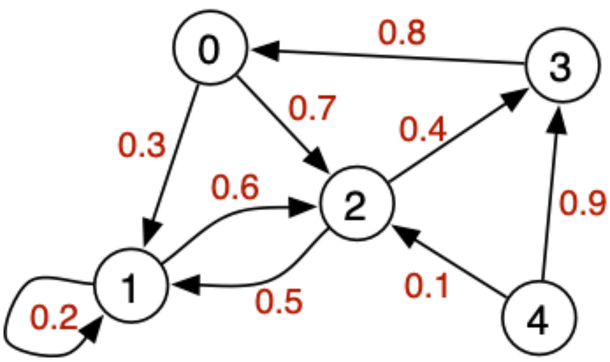
❖ Single-source Shortest Path (SSSP)

Shortest paths from *s* to all other vertices

- **`dist[]`** *V*-indexed array of cost of shortest path from *s*
- **`pred[]`** *V*-indexed array of predecessor in shortest path from *s*

previous node

Example:



no way

	0	1	2	3	4
dist	0	0.3	0.7	1.1	inf
pred	-	0	0	2	-

Shortest paths from s=0

	0	1	2	3	4
dist	1.3	0.6	0.1	0.5	0
pred	3	2	4	2	-

Shortest paths from s=4

❖ Edge Relaxation

Assume: **`dist[]`** and **`pred[]`** as above

- but containing data for shortest paths *discovered so far*

If we have ...

- **`dist[v]`** is length of shortest known path from *s* to *v*
- **`dist[w]`** is length of shortest known path from *s* to *w*
- edge *(v,w,weight)*

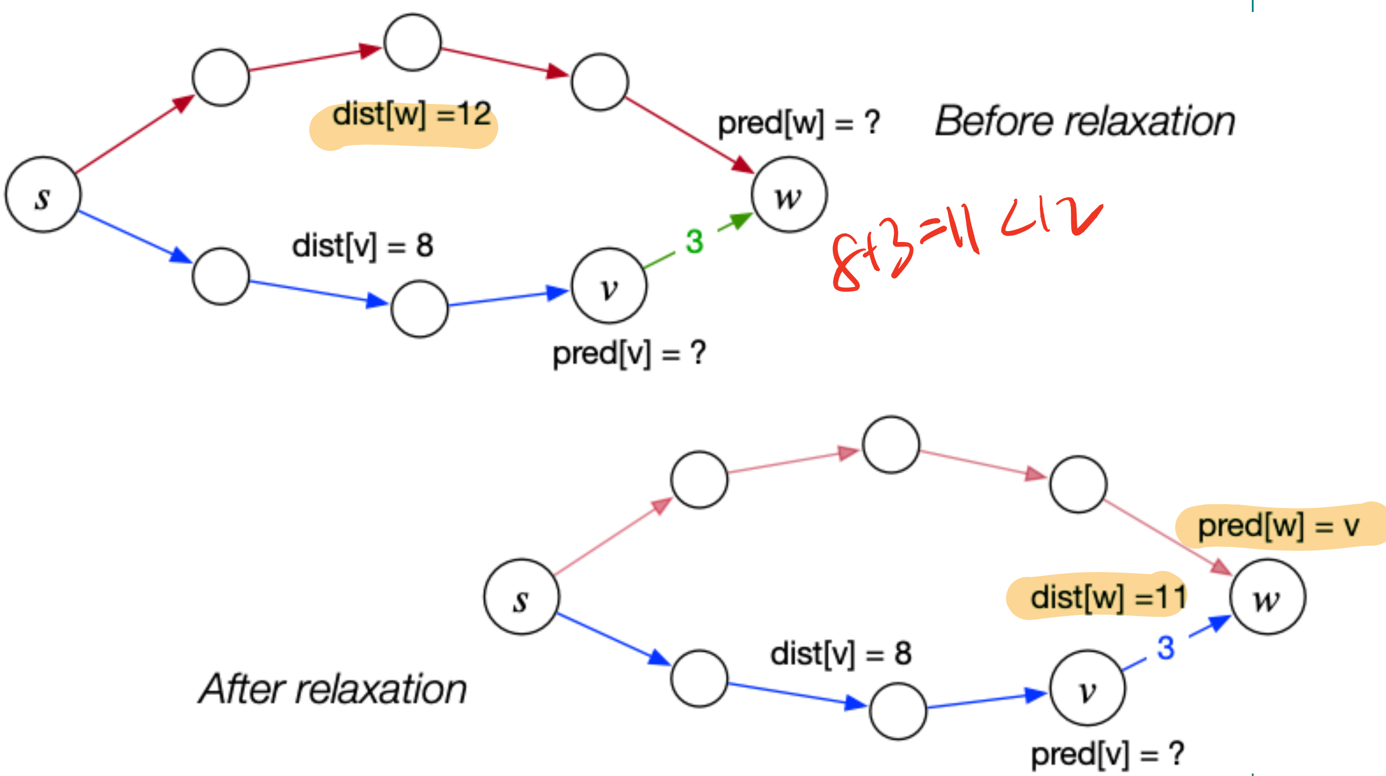
Relaxation updates data for *w* if we find a shorter path from *s* to *w*:

- if **`dist[v]+weight < dist[w]`** then
  update **`dist[w]←dist[v]+weight`** and **`pred[w]←v`**

w 之前是 v

❖ ... Edge Relaxation

Relaxation along edge *e = (v,w,3)* :



❖ Dijkstra's Algorithm

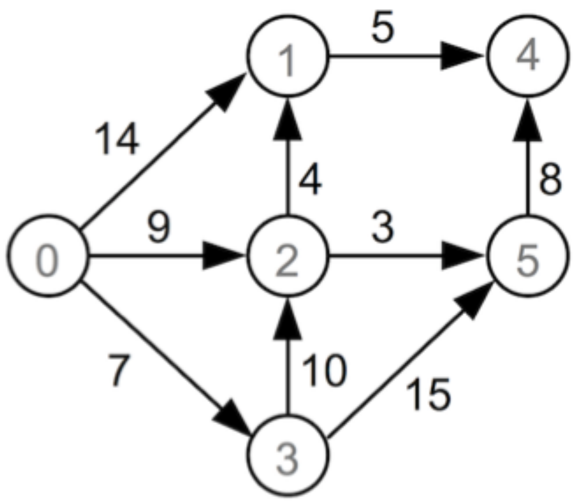
One approach to solving single-source shortest path ...

```
dist[]   // array of cost of shortest path from s
pred[]   // array of predecessor in shortest path from s
vSet     // vertices whose shortest path from s is unknown

dijkstraSSSP(G,source):
|  Input graph G, source node
|
|  initialise all dist[] to ∞
|  dist[source]=0
|  initialise all pred[] to -1
|  vSet=all vertices of G
|  while vSet ≠ ∅ do until empty
|  |  find v ∈ vSet with minimum dist[v]
|  |  for each (v,w,weight) ∈ edges(G) do
|  |     relax along (v,w,weight)
|  |  end for
|  |  vSet=vSet \ {v} - remove v node
|  end while
```

❖ Tracing Dijkstra's Algorithm

How Dijkstra's algorithm runs when source = 0:



Initially

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	inf	inf	inf	inf	inf
pred	-	-	-	-	-	-

从0开始

First iteration, v=0

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	14	9	7	inf	int
pred	-	0	0	0	-	-

```
while vSet not empty do
   find v in vSet
     with min dist[v]
   for each (v,w,weight) in E do
      relax along (v,w,weight)
   end for
   vSet = vSet \ {v}
end while
```

0 to 2,3

Second Iteration, v=3

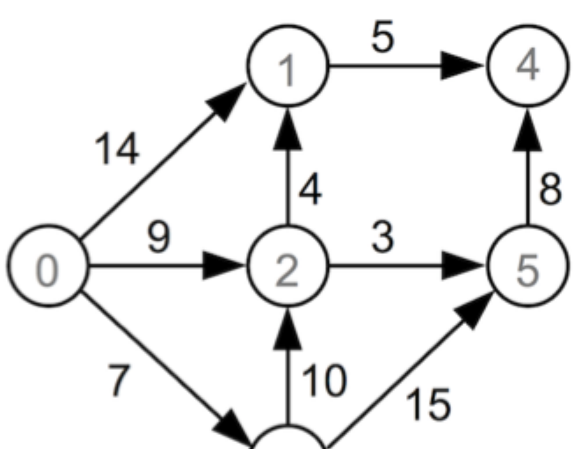
	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	14	9	min 7	inf	22
pred	-	0	0	0	-	3

15+7

Third iteration, v=2

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	min 9	7	inf	12
pred	-	2	0	0	-	2

9+4
9+3



Fourth iteration, v=5

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	20	min 12
pred	-	2	0	0	5	2

pred	-	2	0	0	5	2

```
while vSet not empty do
  find v in vSet
    with min dist[v]
  for each (v,w,weight) in E do
     relax along (v,w,weight)
  end for
  vSet = vSet \ {v}
end while
```

v=4

nothing

remove 4

Fifth iteration, v=1

13+5

	[0]	[1] min	[2]	[3]	[4]	[5]
dist	0	13	9	7	18	12
pred	-	2	0	0	1	2

Sixth iteration, v=4

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	18	12
pred	-	2	0	0	1	2

Completed, vSet is empty

❖ Analysis of Dijkstra's Algorithm

Why Dijkstra's algorithm is correct ...

Hypothesis:

(a) for visited *s*, dist[*s*] is shortest distance from source

(b) for unvisited *t*, dist[*t*] is shortest distance from source *via visited nodes*

Ultimately, all nodes are visited, so ...

- ∀ *v*, dist[v] is shortest distance from source

❖ ... Analysis of Dijkstra's Algorithm

Time complexity analysis ...

Each edge needs to be considered once ⇒ $O(E)$.

Outer loop has $O(V)$ iterations.

Implementing "**`find s ∈ vSet with minimum dist[s]`**"

1. try all **`s`**∈**`vSet`** ⇒ cost = $O(V)$ ⇒ overall cost = $O(E + V^2) = O(V^2)$
2. using a PQueue to implement extracting minimum
   - can improve overall cost to $O(E + V \cdot \log V)$