COMP202
Complexity of Algorithms

Graph algorithms

Reading materials: Chapters 24.1, 25.1 in CLRS

Learning Outcomes

At the conclusion of this set, and the next set, of lecture notes students will

1. Be familiar with some basic graph theory terminology (have received a review of it);
2. Understand and be able to use different algorithms for finding shortest paths in (weighted) graphs and digraphs;
3. Comprehend the *Maximum Flow Problem*, the classical Ford-Fulkerson (augmenting path) algorithm for finding maximum flows in directed graphs (flow networks), and be familiar with the Max Flow/Min Cut theorem.

Connectivity information

Connectivity information can be defined by many kinds of relationships that exist between pairs of objects.

OR CITIES

For example, connectivity information is present in city maps, where the objects are roads, and also in the routing tables for the Internet, where the objects are computers.

Connectivity information

Connectivity information can be defined by many kinds of relationships that exist between pairs of objects.

For example, connectivity information is present in city maps, where the objects are roads, and also in the routing tables for the Internet, where the objects are computers.

Connectivity information is also present in the parent-child relationships defined by a binary tree, where the objects are tree nodes.

Connectivity information

Connectivity information can be defined by many kinds of relationships that exist between pairs of objects.

For example, connectivity information is present in city maps, where the objects are roads, and also in the routing tables for the Internet, where the objects are computers.

Connectivity information is also present in the parent-child relationships defined by a binary tree, where the objects are tree nodes.

Graphs are one way in which connectivity information can be stored, expressed, and utilized.

Graphs

A *graph* is a set of objects, called *vertices* (or *nodes*), together with a collection of pair-wise connections, called *edges*, between them.

Graphs

A *graph* is a set of objects, called *vertices* (or *nodes*), together with a collection of pair-wise connections, called *edges*, between them.

Graphs have applications in a number of different domains, including

- *mapping* (geographic information systems);
- *transportation* (road and flight networks);
- *electrical engineering* (circuit design);
- *process scheduling* (job makespans and assembly-line scheduling); and
- *computer networking* (connectivity of networks).

Graphs (cont.)

$V = \{1, 2, \ldots, n\}$ (12)

More formally, a graph $G = (V, E)$, is a set, V, of *vertices* and a collection, E, of pairs of vertices from V, called *edges*.

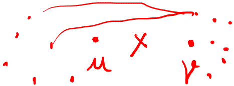
Graphs (cont.)

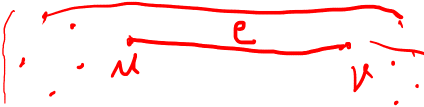
More formally, a graph $G = (V, E)$, is a set, V, of *vertices* and a collection, E, of pairs of vertices from V, called *edges*.

Edges in a graph are either *directed* or *undirected*.

- An edge (u, v) is said to be *directed* from u to v if the pair (u, v) is ordered. If all edges of a graph are directed, we usually refer to G as a *digraph*. (u, v) OR $u \to v$
- An edge (u, v) is said to be *undirected* if the pair (u, v) is unordered. Typically, undirected edges are written as $\{u, v\}$ (using braces instead of parentheses).

Graph Terminology





Two vertices are said to be *adjacent* if they are end-points of the same edge.

An edge is said to be *incident* to a vertex if the vertex is one of its end-points.

e inc. to u, e inc to v.

An *outgoing edge* of a vertex is a directed edge whose origin is that vertex.

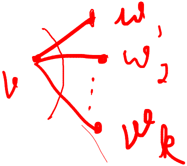
$u \xrightarrow{e} v$

e is OUTGOING FOR u

An *incoming edge* of a vertex is a directed edge whose destination is that vertex.

e IS INCOMING FOR v.

UNDIR.

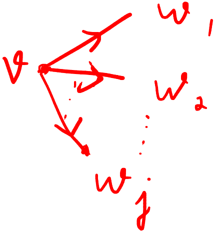


$\underline{deg(v) = k}$

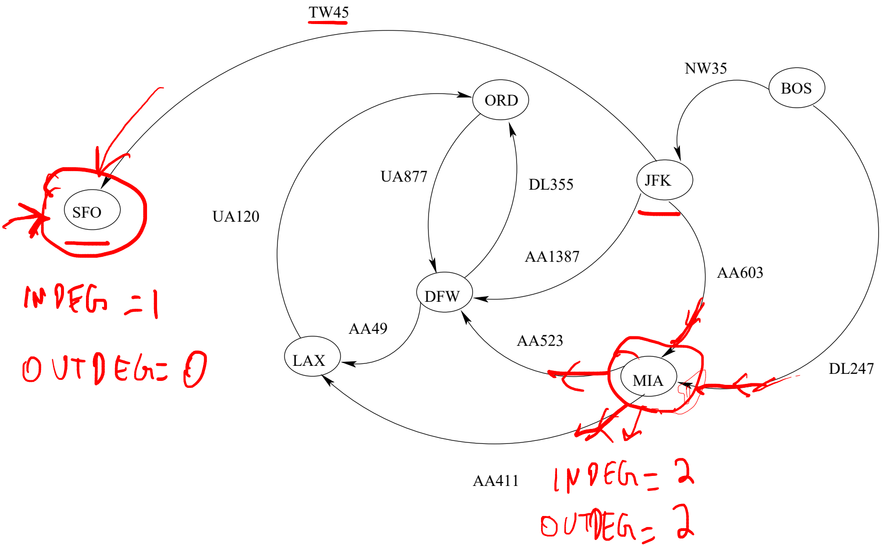
The *degree* of a vertex v, denoted $deg(v)$, is the number of edges incident to v.

In a directed graph, the *in-degree* (*out-degree*) of a vertex v is the number of *incoming* (*outgoing*) edges of v, and is denoted by $indeg(v)$ ($outdeg(v)$).

OUT.DEG(v) = j



Digraph - Example



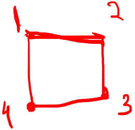
We have the following two elementary results about graphs.

Theorem: If G is an undirected graph with m edges then

$$\sum_{v \in V} deg(v) = 2m.$$

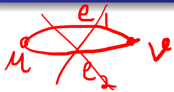
Theorem: If G is a directed graph with m edges, then

$$\sum_{v \in V} indeg(v) = \sum_{v \in V} outdeg(v) = m.$$



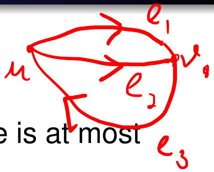
deg(1) + (2) (3) + deg(4)

(12), (14) (23), (12) (34), (23) (14), (34)



$\rightarrow$

$\rightarrow$

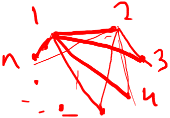


An undirected graph G is said to be *simple* if there is at most one edge between each pair of vertices u and v.

A digraph is *simple* if there is at most one directed edge from u to v for every pair of distinct vertices u and v.

Theorem: Let G be a *simple* graph with n vertices and m edges.

- If G is *undirected*, then $m \leq \frac{n(n-1)}{2}$. $= \binom{n}{2}$
- If G is *directed*, then $m \leq n(n-1)$.



$\in \{1, \dots, n\}$

$\rightarrow \deg(i) = n-1$. $\quad \sum_v \deg(v) = n(n-1)$

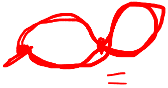
THM. $\Rightarrow$ $m = \frac{n(n-1)}{2}$.

More graph terminology

A *walk* in a graph is a sequence of alternating vertices and edges, starting at a vertex and ending at a vertex.

A *path* is a walk where each vertex in the walk is *distinct*.

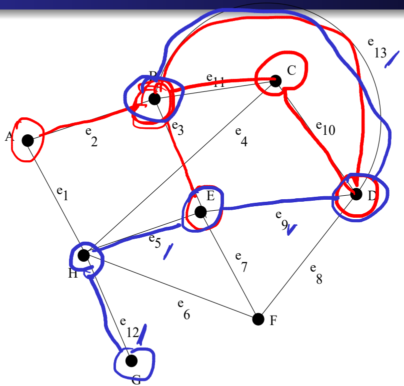
A *circuit* is a walk with the same start and end vertex.



A *cycle* is a circuit where each vertex in the circuit is *distinct* (except for first and last vertex).

A *directed walk* is a walk in which all edges are directed and are traversed along their direction. Directed paths, circuits, and cycles are defined similarly.

$u \longrightarrow v$



$A, e_2, B, e_{13}, D, e_{10}, C, e_{11}, B, e_3, E$ is a *walk* in this graph.

$G, e_{12}, H, e_5, E, e_9, D, e_{13}, B$ is a *path* (and also a walk) joining G and B.

$B, e_{11}, C, e_{10}, D, e_8, F, e_7, E, e_5, H, e_1, A, e_2, B$ is a *cycle*.

A small note...

If G is a simple graph, then giving the sequence of vertices is sufficient to describe a walk, path, circuit, or cycle (as then the edges are implied).

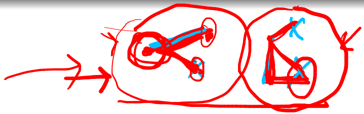
For example, the path joining G and B on the previous slide could be (more compactly) represented as

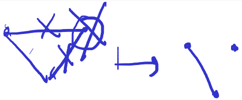
$$G, H, E, D, B.$$

Similarly the cycle could be written as

$$B, C, D, F, E, H, A, B.$$

Still more terminology





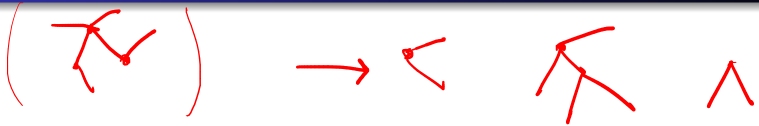
A *subgraph* of a graph G is a graph H whose vertices and edges are *subsets* of the vertices and edges of G.

A *spanning subgraph* of G is a subgraph of G that contains all the vertices of G.

A graph is *connected* if, for any two distinct vertices, there is a path between them. (IF NOT, DISCONNECTED)

If a graph G is not connected, its maximal connected subgraphs are called the *connected components* of G.

Graphs (cont.)



A *forest* is a graph without cycles.

A *tree* is a *connected forest*, i.e. a connected graph without cycles.

A tree with a distinguished node (*root*) is called a *rooted tree*, otherwise it is called a *free tree* (or, often, simply a *tree*).

A *spanning tree* of a graph is a spanning subgraph that is a free tree.

Let G be an undirected graph with n vertices and m edges. We have the following observations:

- If G is *connected*, then $m \geq n - 1$.
- If G is a tree then, $m = n - 1$.
- If G is a forest, then $m \leq n - 1$.

THEOREM: IF G IS A TREE, THEN $m = n - 1$.

→ PF: INDUCTION ON n. → (n = # VERTS, m = # EDGES)

✓ BASE: $n = 1$

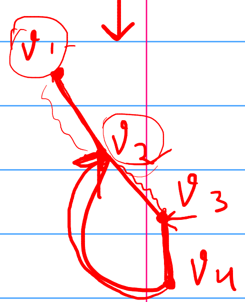
$n = 2$ $m = 1$ ✓

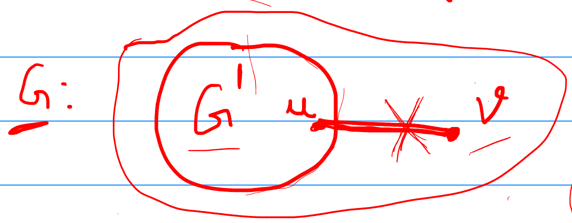
→ IND. STEP: ASSUME TRUE FOR # VERTS. $= k$

WILL PROVE → G: # VERTS $= k+1$, G IS TREE

[LEMMA: FOR A TREE G', $\exists\, v$ $\deg(v) = 1$.]

LEMMA TRUE.





DEFINE G' TO BE THE GRAPH OBTAINED BY DELETING v (& (v, u))

→ # verts OF $G' = k+1-1 = k$

G' IS TREE:

- G' HAS NO CYCLE (BECAUSE NEITHER DOES G ↓ TREE)
- G' IS CONNECTED (BECAUSE SO IS G)

I.H → ⇒ # EDGES IN G' = # VERTS IN G' $- 1$

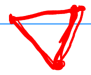
$= k - 1$

⇒ G HAS k EDGES.

THM IF G HAS $m \geq n$ EDGES, THEN G CONTAINS A CYCLE.

PF: IND ON n, $n=3$

$m=n=3$



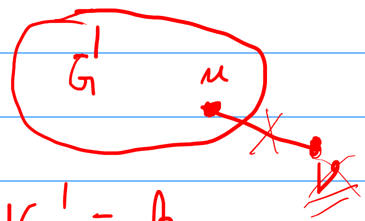
$m \geq n$

BASE ✓

→ ASSUME TRUE FOR # VERTS $= k$

→ G: $n=k+1$ VERTS, $m \geq k+1$ EDGES

→ 1) G HAS A v, $\deg(v)=1$

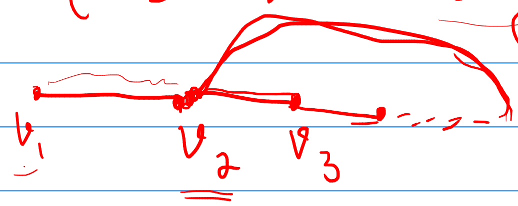
— 

[G' : G AFTER DELETING v (& (u,v))]

VERTS IN G' $= k$,
EDGES $\geq k$

I.H $\Rightarrow$ G' HAS CYCLE. ✓

2) NO VERTEX HAS DEG$=1$.
(ALL v: $\deg(v) \geq 2$)

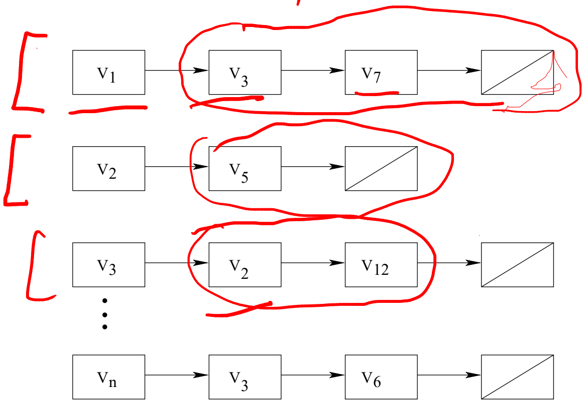


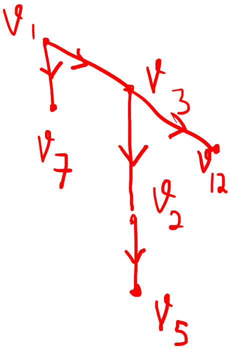
Proofs

See the proofs.pdf file in the Canvas materials for some proofs.

Representing Graphs - Adjacency List (linked lists)

$V = \{v_1, \ldots, v_n\}$





Representing Graphs - Adjacency Matrix

Here we represent the structure of the graph with a $\{0, 1\}$ matrix, the ones signifying that there is an edge present between the two vertices.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	1	0	1	0	0
v_2	1	0	1	0	1	0	0
v_3	1	1	0	0	0	0	1
v_4	0	0	0	0	1	1	0
v_5	1	1	0	1	0	0	1
v_6	0	0	0	1	0	0	0
v_7	0	0	1	0	1	0	0

$\# R = \# C = \# V.$

(v_5, v_2) EDGE

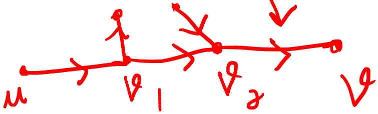
NO (v_5, v_6) EDGE

Digraphs

A *digraph* is a graph whose edges are all directed.

A fundamental issue with directed graphs is the notion of *reachability*, which deals with determining where we can get to in a directed graph.

Given two vertices u and v of a digraph G, we say that u reaches v (or v is reachable from u) if G has a *directed path* from u to v.



Graph search methods

Two common methods for exploring graphs are the *Depth-First Search* (DFS) and *Breadth-First Search* (BFS) methods.

Graph search methods

Two common methods for exploring graphs are the *Depth-First Search* (DFS) and *Breadth-First Search* (BFS) methods.

As a very brief reminder, DFS starts at a vertex, chooses an edge from that vertex and “walks out as far as possible”, finding new vertices until it encounters one it has already seen, then it backs up (as little as possible) to find other un-encountered vertices.

Graph search methods

Two common methods for exploring graphs are the *Depth-First Search* (DFS) and *Breadth-First Search* (BFS) methods.

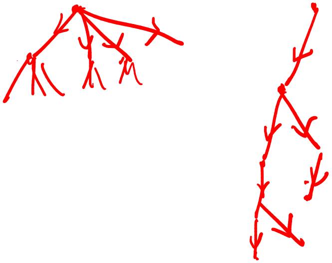
As a very brief reminder, DFS starts at a vertex, chooses an edge from that vertex and "walks out as far as possible", finding new vertices until it encounters one it has already seen, then it backs up (as little as possible) to find other un-encountered vertices.

In contrast to the DFS method, the Breadth First Search algorithm starts at a vertex and first explores the entire neighborhood of that vertex before moving onto another vertex.

Graph search methods (cont.)

Therefore, the DFS method generates "long, skinny" search trees, while the BFS method generates "short, bushy" ones.

The running time of BFS and DFS is the same, namely $O(n+m)$.



Graph search methods (cont.)

Therefore, the DFS method generates “long, skinny” search trees, while the BFS method generates “short, bushy” ones.

The running time of BFS and DFS is the same, namely $O(n + m)$.

Generally speaking, we may utilize a *stack* data structure to control a DFS search, while a *queue* data structure may be used for a BFS search.

Graph search methods (cont.)

Therefore, the DFS method generates “long, skinny” search trees, while the BFS method generates “short, bushy” ones.

The running time of BFS and DFS is the same, namely $O(n+m)$.

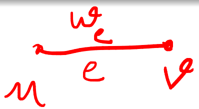
Generally speaking, we may utilize a *stack* data structure to control a DFS search, while a *queue* data structure may be used for a BFS search.

We will not go into great detail about these search methods here. See sections 22.3, 22.4 in CLRS!

BFS and DFS can be used to answer a variety of questions about graphs including:

- Testing whether G is connected.
- Computing the connected components of G.
- Finding a spanning forest of G (or spanning tree if G is connected).
- Searching for a cycle in G, or reporting that G is acyclic.
- Given a start vertex x of G, computing, for every vertex v of G, a path with the minimum number of edges between x and v, or reporting that no such path exists (BFS).
- Testing for *strong connectivity* of digraphs. (Is there a directed path from u to v, for all u and v in D?)

Weighted Graphs



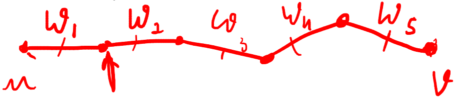
A *weighted graph* is a graph that has a numerical label $w(e)$ associated with each edge e, called the *weight* of e.

Alternatively, we might sometimes consider graphs having weights on the *vertices*, or on both the vertices and edges.

Single-Source Shortest Paths

Often in the case of weighted graphs, we want to consider the following problem:

For some fixed vertex v, find a *shortest path* from v to all other vertices $u \neq v$ in G (viewing weights on edges as distances between adjacent vertices).



Single-Source Shortest Paths

Often in the case of weighted graphs, we want to consider the following problem:

For some fixed vertex v, find a *shortest path* from v to all other vertices $u \neq v$ in G (viewing weights on edges as distances between adjacent vertices).

The *length* (or *weight*) of a path P is the *sum of the weights of the edges* $e_1, \ldots e_k$ of P, i.e. $w(P) = \sum_{i=1}^{k} w(e_i)$.

Single-Source Shortest Paths

Often in the case of weighted graphs, we want to consider the following problem:

For some fixed vertex v, find a *shortest path* from v to all other vertices $u \neq v$ in G (viewing weights on edges as distances between adjacent vertices).

The *length* (or *weight*) of a path P is the *sum of the weights of the edges* $e_1, \ldots e_k$ of P, i.e. $w(P) = \sum_{i=1}^{k} w(e_i)$.

The *distance* from a vertex u to a vertex v in G, denoted $d(u, v)$ is the value of a *minimum length path* (also called a *shortest path*) from u to v.

Single-Source Shortest Paths

Often in the case of weighted graphs, we want to consider the following problem:

For some fixed vertex v, find a *shortest path* from v to all other vertices $u \neq v$ in G (viewing weights on edges as distances between adjacent vertices).

The *length* (or *weight*) of a path P is the *sum of the weights of the edges* $e_1, \ldots e_k$ of P, i.e. $w(P) = \sum_{i=1}^{k} w(e_i)$.

The *distance* from a vertex u to a vertex v in G, denoted $d(u, v)$ is the value of a *minimum length path* (also called a *shortest path*) from u to v.

This problem is known as the *Single-Source Shortest Path* problem (SSSP for short).

Greedy Approach to SSSP

The main idea in applying the greedy method to SSSP is to perform a “weighted” Breadth First Search.

One algorithm using this design pattern in known as *Dijkstra’s algorithm*.

Dijkstra’s algorithm can be used in both directed and undirected graphs, with one requirement: all edges in the graph have *non-negative weights*. (This is a requirement for Dijkstra’s algorithm to work correctly.)

Greedy Approach to SSSP

The main idea in applying the greedy method to SSSP is to perform a “weighted” Breadth First Search.

One algorithm using this design pattern in known as *Dijkstra’s algorithm*.

Dijkstra’s algorithm can be used in both directed and undirected graphs, with one requirement: all edges in the graph have *non-negative weights*. (This is a requirement for Dijkstra’s algorithm to work correctly.)

You’ve already seen Dijkstra’s algorithm in COMP108.

What about negative weights?

If the graph G contains some edges with negative weights, then Dijkstra's algorithm may not give the correct results.

What about negative weights?

If the graph G contains some edges with negative weights, then Dijkstra's algorithm may not give the correct results.

Other algorithms, such the Bellman-Ford algorithm may be used if there are negative weights (but in this case, the graph must be directed).

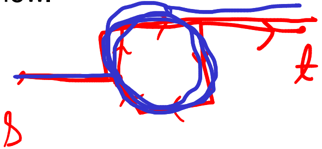
We'll next see the Bellman-Ford algorithm, and then algorithms for the *All-Pairs-Shortest-Paths* problem (henceforth referred to as APSP).

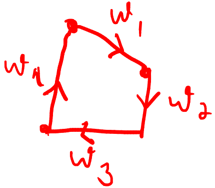
Input: Weighted, directed graph $G = (V, E)$ with source s and a weight function $w : E \to \mathbb{R}$.

Output: 'False' if there is a negative-weight cycle. Otherwise, output shortest paths and their weights, between s and all other vertices.

$n-1$

Idea of algorithm: Iterate n times (where $n = |V|$): "relax" (as in Dijkstra) all edges. Let's look at the algorithm in more detail now.





$w_1 + w_2 + w_3 + w_4 < 0$

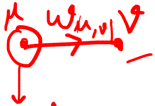
|||

NEG. WT. CYCLE

WT SOURCE # V

BELLMAN-FORD(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
for $i = 1$ **to** $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)
for each edge $(u, v) \in G.E$
  if $v.d > u.d + w(u, v)$
    return FALSE
return TRUE



"DIST" $u.d$

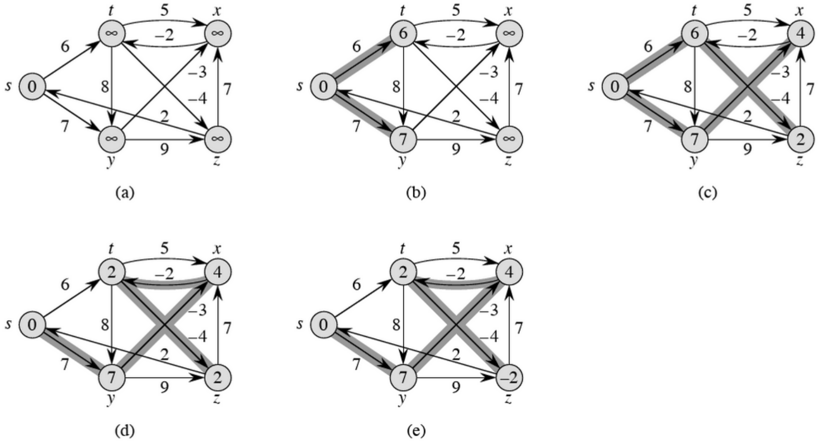
"PREDECESSOR" $u.\pi$

Here, $v.d$ eventually is intended to be $\delta(s, v)$, the length of the shortest path between s and v.

RELAX(u,v,w): IF $\boxed{u.d + w_{u,v}} < v.d$

$v.d =$ & $v.\pi = u$

An example



(a) (b) (c) (d) (e)

Order of relaxation of edges in each pass:
$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.

Running time and correctness of Bellman-Ford

$|V|-1$ + 1 ITERATION FOR CHECKING NEG. WT. CYCLES.

Total of $|V|$ iterations, each iteration relaxes all of the $|E|$ edges. Each edge relaxation takes $O(1)$ time. Total time: $O(|V| \cdot |E|)$.

IF NO NEG WT. CYCLES, LENGTH OF SHORTEST PATH $\leq n-1$

SOURCE

$u \xrightarrow{w} v$

u w v

S.P. FROM $u \to w$

S.P. $w \to v$.

Running time and correctness of Bellman-Ford

Total of $|V|$ iterations, each iteration relaxes all of the $|E|$ edges. Each edge relaxation takes $O(1)$ time. Total time: $O(|V| \cdot |E|)$.

Correctness: Lemma 24.2, 24.15 in CLRS.

→ BELLMAN-FORD : $d(s,v)$ FOR ALL VERTS v.

→ $O(|V| \cdot |E|)$.

What if we do not want shortest paths from a single source, but shortest path lengths between *all* pairs of vertices?

GOAL: $d(u,v)$ FOR ALL u,v.

We refer to this problem as the APSP problem.

We will use a *dynamic programming* approach to solve this problem.

$F(1)=F(2)=1$

$|V|$ TIMES

Can run Bellman-Ford once for each vertex as source. Total running time $O(|V|^2 \cdot |E|)$, which is potentially $O(|V|^4)$.

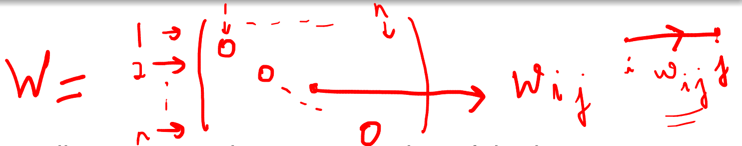
$F(n)=F(n-1)+F(n-2)$

STRUCTURE SUBPROBLEMS

— OPTIMAL "SUBSTRUCTURE".

— OVERLAP IN SUBPROBLEMS

Input to APSP



We'll assume adjacency-matrix representation of the input graph. Assume the vertices are numbered $1, 2, \ldots, |V|$. Thus, the input is a $|V| \times |V|$ matrix $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & i = j \\ \text{weight of directed edge}(i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E. \end{cases}$$

Define $\ell_{ij}^{(m)}$ to be the minimum weight of any path from vertex i to vertex j that contains at most m edges.. Thus,

$m=1$ $\quad \ell_{ij}^{(1)} = w_{ij}$

$$\ell_{ij}^{(0)} = \begin{cases} 0 & i = j \\ \infty & i \neq j. \end{cases}$$

Observe that

COVERED WHEN $k=j$

$$\ell_{ij}^{(m)} = \min\left(\ell_{ij}^{(m-1)}, \min_{1\leq k\leq n}\{\ell_{i,k}^{(m-1)} + w_{kj}\}\right)$$

$\leq m-1$

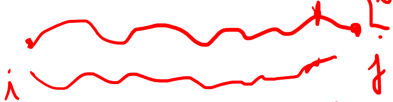
$< m$ EDGES

$$= \min_{1\leq k\leq n}\{\ell_{ik}^{(m-1)} + w_{kj}\}.$$

(a)

(b) HAS $= m$ EDGES

k



$\underbrace{i \to k}_{m-1}, \underbrace{k \to j}_{\text{EDGE}}$

$\leq m$ EDGES

APSP and matrix multiplication

Define $\ell_{ij}^{(m)}$ to be the minimum weight of any path from vertex i to vertex j that contains at most m edges.. Thus,

$$\ell_{ij}^{(0)} = \begin{cases} 0 & i = j \\ \infty & i \neq j. \end{cases}$$

Observe that

$$\begin{aligned} \ell_{ij}^{(m)} &= \min\left(\ell_{ij}^{(m-1)}, \min_{1\le k\le n}\{\ell_{i,k}^{(m-1)} + w_{kj}\}\right) \\ &= \min_{1\le k\le n}\{\ell_{ik}^{(m-1)} + w_{kj}\}. \end{aligned}$$

Also note that shortest paths must contain at most $n-1$ edges (why?).

MIN WT. PATH FROM $i \to j$. $\leftarrow$ $\delta(i,j) = \ell_{ij}^{(n-1)} = \ell_{ij}^{(n)} = \ell_{ij}^{(n+1)} = \cdots$.

Computing the shortest-path weights bottom up

Input matrix $W = (w_{ij})$. Compute a series of matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where $L^{(m)} = (\ell_{ij}^{(m)})$ for each m. As argued on the previous slide, $\underline{\underline{L^{(n-1)}}}$ contains all the shortest-path weights.

$$L^{(m)} = \begin{array}{c} \\ 1 \to \\ \vdots \\ n \to \end{array} \begin{array}{c} \begin{array}{ccc} 1 & \cdots & n \end{array} \\ \left(\begin{array}{ccc} & & \\ & & \\ & & \end{array} \right) \end{array} \quad |V| \times |V|$$

$$L^{(m)}_{ij} = \ell^{(m)}_{ij}$$

Computing the shortest-path weights bottom up

Input matrix $W = (w_{ij})$. Compute a series of matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where $L^{(i)} = (\ell_{ij}^{(m)})$ for each i. As argued on the previous slide, $L^{(n-1)}$ contains all the shortest-path weights.

Compare with matrix multiplication $C = A \cdot B$:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

PREV. SLIDE

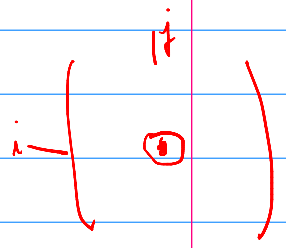
$$\boxed{\ell_{ij}^{(m)} = \min_{1 \le k \le n} \{\ell_{ik}^{(m-1)} + w_{kj}\}.}$$

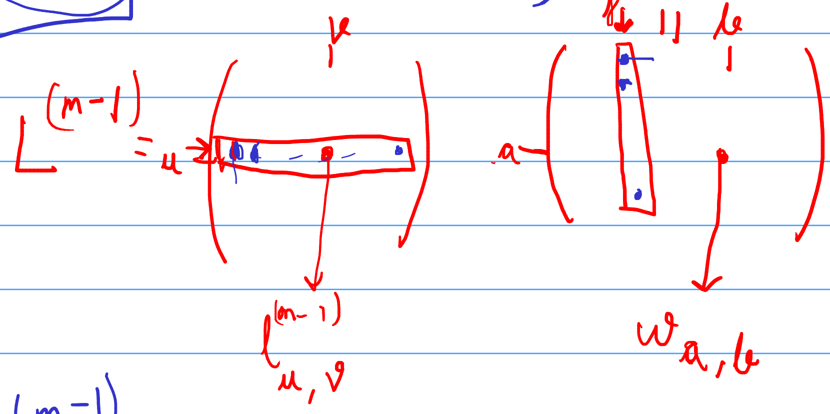
We have $C = L^{(m)}$, $A = L^{(m-1)}$, $B = W$, and $\sum \mapsto \min$, $(\cdot \mapsto +)$.
Call this type of "matrix multiplication" $A \odot B$.

$|V| = n$

$$\rightarrow \quad l_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}$$

$L^{(m)}$





$$L^{(m)} = L^{(m-1)} \odot W.$$

TO COMPUTE $L_{ij}^{(m)}$, NEED TO COMPUTE n SUMS

A $|V|^4$-cost algorithm

COMPUTING SINGLE ENTRY TAKES $O(n)$ TIME, n^2 ENTRIES

$\Downarrow$

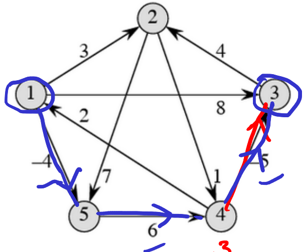
EACH EQ (MATRIX "MULT") $O(n^3)$

$O(n)$ EQS.

$$
\begin{aligned}
L^{(1)} &= L^{(0)} \odot W = W \\
L^{(2)} &= L^{(1)} \odot W = W^{\odot 2} \\
&\vdots \\
L^{(n-1)} &= L^{(n-2)} \odot W = W^{\odot n-1}.
\end{aligned}
$$

Each multiplication takes $|V|^3$ time (note that Strassen's algorithm cannot be applied here!), giving a total time complexity of $|V|^4$. **Can we improve upon this?**

An example



$$L^{(3)} = L^{(2)} \odot W$$

$$L^{(3)}_{1,3} = \min[8, \infty, 8, -3, \infty] = -3$$

$$W = L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Repeated squaring

Can show that $A \odot (B \odot C) = (A \odot B) \odot C$. Thus, assuming $n-1$ is a power of 2,

M^{2^n}

$M \cdot M = M^2$

$M^2 \cdot M^2 = M^4$

$M^4 \cdot M^4 = M^8$

$O(\log n)$

$$
\begin{aligned}
L^{(1)} &= W, \\
L^{(2)} &= W^{\odot 2} = W \odot W, \\
L^{(4)} &= W^{\odot 4} = W^{\odot 2} \odot W^{\odot 2}, \\
&\vdots \\
L^{(n-1)} &= W^{\odot (n-1)} = W^{\odot (n-1)/2} \odot W^{\odot (n-1)/2}.
\end{aligned}
$$

There are $1 + \log(n-1)$ equations above, each of them takes $O(n^3)$ time to compute, giving a total running time of $O(n^3 \log n)$.

Some notes about APSP

$O(n^{2.99})$?

- The algorithm we saw gave a running time of $O(n^3 \log n)$, which is better than the $O(n^4)$ algorithm obtained by repeating Bellman-Ford for each vertex.
- One can use dynamic programming to improve this to $O(n^3)$ using the Floyd-Warshall algorithm.
- It is a very big open question to show an $O(n^{3-\varepsilon})$ algorithm for APSP for any constant $\varepsilon > 0$.
- In fact there is a whole research area in theoretical computer science (fine-grained complexity) that assumes that APSP and $(\min, +)$-matrix multiplication cannot be done faster than $O(n^3)$.