

**Topic 20: Shortest Paths in Graphs**  
(CLRS 24.0–24.3, 25.2)

CPS 230, Fall 2001

Digraph  $G = (V, E)$  with weight function  $W : E \rightarrow \mathfrak{R}$

Weight of path  $p = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$  is

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

“Shortest” path = path of minimum weight.

Applications

- Static/dynamic network routing
- Robot motion planning

- Different variants of shortest path problem:

- *Single pair shortest path*: Find shortest path from  $u$  to  $v$ .
- *Single source shortest path (SSSP)*: Find shortest path from source  $s$  to all vertices  $v \in V$ .
- *All pair shortest path (APSP)*: Find shortest path from  $u$  to  $v$  for all  $u, v \in V$ .

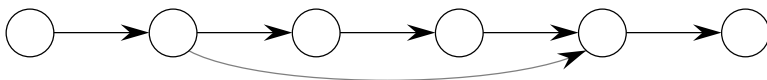
- Note:

- No algorithm is known for computing a single pair shortest path better than solving the (“bigger”) SSSP problem.
- APSP can be solved by running SSSP  $|V|$  times.  
 $\implies$  Lets focus on the SSSP problem.

**Optimal Substructure**

**Theorem:** Subpaths of shortest paths are shortest paths.

**Proof:** Cut and paste:



If some subpath were **not** a shortest path, we could substitute the shorter subpath and create an even shorter total path.  $\square$

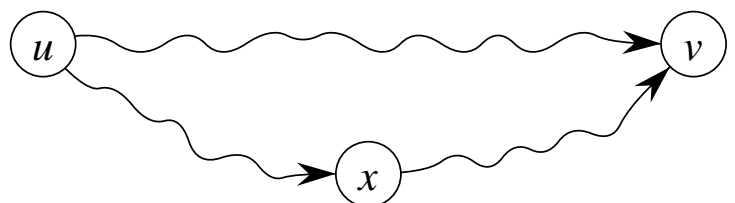
**Triangle Inequality**

**Definition:**  $\delta(u, v) \equiv$  weight of a shortest path from  $u$  to  $v$ .

**Theorem:**  $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$

**Proof:** Shortest path  $u \rightarrow \dots \rightarrow v$  is no longer than any other path  $u \rightarrow \dots \rightarrow v$ . In particular, it is no longer than the path concatenating the shortest path  $u \rightarrow \dots \rightarrow x$  with the shortest path  $x \rightarrow \dots \rightarrow v$ .

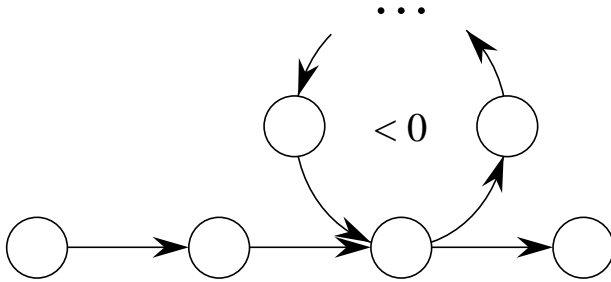
$\square$



## Is shortest-path well-defined?

**Negative weight cycle**  $\Rightarrow$  no shortest path.

**Argument:** Can shorten path by traversing cycle.  $\square$



## Bellman-Ford Algorithm

Most basic “single-source” shortest-paths algorithm

- Finds shortest path weights from specified **source**  $s$  to **all**  $v \in V$
- Maintains estimate  $d[v]$  of path length from  $s$  to  $v$ , which is updated iteratively
- Actual paths easily reconstructed (CLRS §24.3)

## Bellman-Ford Algorithm

```
BELLMAN-FORD( $G, w, s$ )
1 for each  $v \in V$ 
2   do  $d[v] \leftarrow \infty$ 
3  $d[s] \leftarrow 0$   $\triangleright$  INITIALIZE-SINGLE-SOURCE( $G, s$ )

4 for  $i \leftarrow 1$  to  $|V| - 1$ 
5   do for each edge  $(u, v) \in E$   $\triangleright$  RELAX
6     do if  $d[v] > d[u] + w(u, v)$ 
7       then  $d[v] \leftarrow d[u] + w(u, v)$ 

8 for each edge  $(u, v) \in E$ 
9   do if  $d[v] > d[u] + w(u, v)$ 
10    then no solution
```

Why call it “Relax”? Chain is getting tighter!

Think of violations of  $d[v] \leq d[u] + w(u, v)$  as “pressure”. The pressure is relaxed by shortening the estimate of the distance from  $s$  to  $v$ .

## Bellman-Ford Algorithm

**Three code sections:**

- Lines 1 – 3:  
**Initialize:**  $d[v]$ , which will converge to shortest-path values  $\delta$ .
- Lines 4 – 7:  
**Relax:** For  $|V| - 1$  times, do the following:  
For each edge, do a **relaxation step**.
- Lines 8 – 10:  
**Test:** Was a solution achieved (iff no negative-weight cycles)?

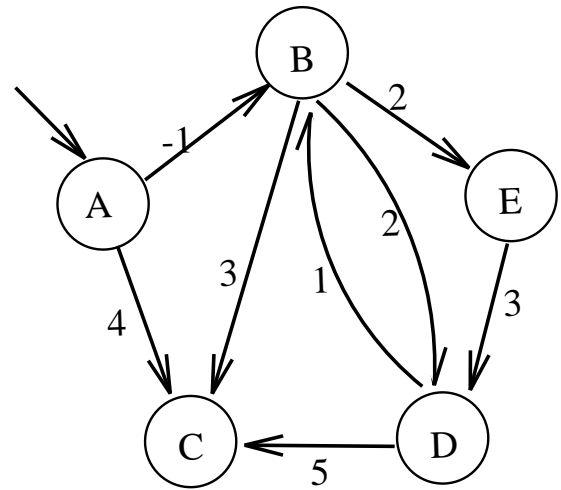
## Bellman-Ford Algorithm: Running time

Running Time:  $O(V \cdot E)$

- constants are good
- it is simple
- short code

very practical.

## Bellman-Ford Algorithm Example



## Bellman-Ford Algorithm Example

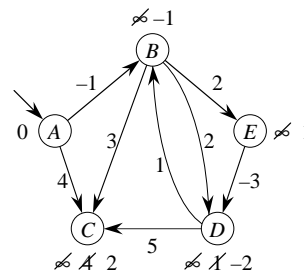
- **Initialization.** Put initial  $d$  values in nodes:  
 $A \leftarrow 0$ , rest  $\leftarrow \infty$ .
- **1st relaxation pass.** Process edges in order  $(A, B)$ ,  $(A, C)$ ,  $(B, C)$ ,  $(B, D)$ ,  $(D, B)$ ,  $(D, C)$ ,  $(E, D)$ ,  $(B, E)$ .
- **2nd relaxation pass.** Process edges in same order. Only  $D$  changes.

## Bellman-Ford Algorithm Example

- Can stop when no change is detected

$d$	A	B	C	D	E
init	0	$\infty$	$\infty$	$\infty$	$\infty$
pass 1	0	-1	2	1	1
pass 2	0	-1	2	-2	1
pass 3	0	-1	2	-2	1

- The distances in each pass and the convergence speed of the algorithm depend on the order that the edges are processed.



## Bellman-Ford Algorithm: Lemma

**Lemma:**  $d[v] \geq \delta(s, v)$  always.

**Proof:**

- Initially true
- Let  $v$  be first vertex for which  $d[v] < \delta(s, v)$ , and let  $u$  be vertex that caused  $d[v]$  to change:

$$d[v] = d[u] + w(u, v)$$

- Then

$$\begin{aligned} d[v] &< \delta(s, v) \\ &\leq \delta(s, u) + \delta(u, v) \quad (\text{Triangle inequality}) \\ &\leq \delta(s, u) + w(u, v) \quad (\text{shortest path } \leq \text{specific path}) \\ &\leq d[u] + w(u, v) \quad (v \text{ is first violation}) \end{aligned}$$

contradicts  $d[v] = d[u] + w(u, v)$  (above).

Therefore, once  $d[v]$  reaches  $\delta(s, v)$ , it can't change (since  $d[v]$  can only decrease, never increase).

## Bellman-Ford Algorithm: Correctness

**Claim:** Bellman-Ford correct (i.e., after  $|V| - 1$  passes, all the  $d$  values are correct)

**Proof:** Let  $v$  be any vertex, and consider a shortest path from  $s$  to  $v$  (assuming no neg-weight cycles):

$$s \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v$$

- Initially,  $d[s] = 0$  is correct (and doesn't change thereafter since the algorithm never increases  $d$ )

## Bellman-Ford Algorithm: Correctness

**Proof:** (continued)

- After 1 pass through edges,  $d[v_1]$  is correct (and doesn't change thereafter)

( $d[s]$  is correct, and by the optimal substructure, the shortest distance is  $w(s, v_1)$ .)

1st pass sets  $d[v_1] \leftarrow d[s] + w(s, v_1)$ , which is the right answer.)

- After 2 passes through edges,  $d[v_2]$  is correct (and doesn't change thereafter)

⋮

- Terminates in  $|V| - 1$  passes. Why?

If no negative-weight cycles:

- every shortest path is **simple** (no cycles)
- longest simple path has  $|V| - 1$  edges

## Bellman-Ford Algorithm: Correctness

**Proof:** (continued)

- Thus if no neg-weight cycles, all the  $d[v]$  converge in  $|V| - 1$  passes. Equivalently, if a value  $d[v]$  fails to converge after  $|V| - 1$  passes,  $\exists$  neg-weight cycle.
- Last part of algorithm tests for success by seeing if another pass would change anything.

The converse is also true:

If  $\exists$  neg-weight cycle reachable from  $s$ , then some value  $d[v]$  fails to converge after  $|V| - 1$  passes.

(Proof left as exercise.) (CLRS Theorem 24.4.)

**So...** Bellman-Ford can be used to check for negative-weight cycles.

## SSSP in DAG

- If graph is acyclic, we can solve SSSP by relaxing outgoing edges from vertices in the topological sort order of the vertices.
- Running time is  $O(|E|)$ .

## Dijkstra's Algorithm

Dijkstra's Algorithm:

- Non-negative edge weights  
 $\implies$  shortest paths always exist.  
(If there are no negative weights, Dijkstra's algorithm is faster than Bellman-Ford.)
- Like breadth-first-search  
(If all weights = 1, use BFS, otherwise Dijkstra.)
- Use for  $Q$  a priority queue keyed by  $d[v]$ .  
Greedy, like Prim's algorithm for MST  
BFS used FIFO queue

## Dijkstra's Algorithm: Pseudocode

```
DIJKSTRA( $G, w, s$ )
1 for each  $v \in V$ 
2   do  $d[v] \leftarrow \infty$ 
3  $d[s] \leftarrow 0$ 
4  $S \leftarrow \emptyset$ 
5  $Q \leftarrow V$ 
6 while  $Q \neq \emptyset$ 
7   do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
8      $S \leftarrow S \cup \{u\}$ 
9     for each  $v \in \text{Adj}[u]$ 
10      do if  $d[v] > d[u] + w(u, v)$ 
11        then  $d[v] \leftarrow d[u] + w(u, v)$ 
```

What is line 7 doing?

What is line 11 doing?

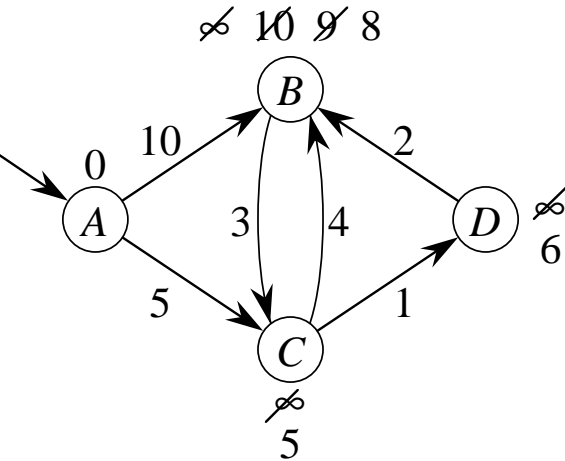
## Dijkstra's Algorithm: Notes

Observe:

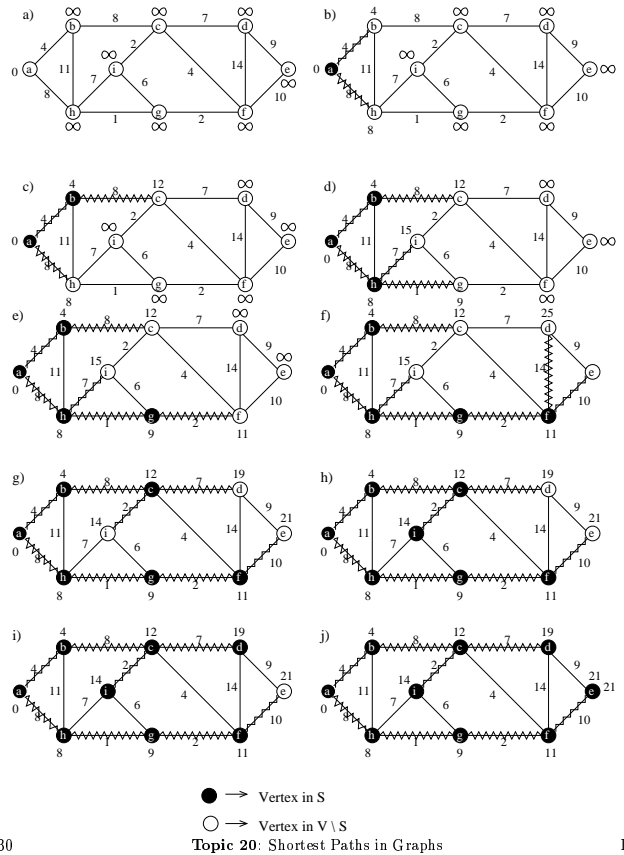
- relaxation step
- setting  $d[v]$  updates  $Q$   
(DECREASE-KEY operation)
- similar to Prim's minimum-spanning-tree algorithm

# Dijkstra's Algorithm

Example:



# Another Example of Dijkstra's Algorithm



## Dijkstra's Algorithm: Run-Time Analysis

- EXTRACT-MIN executed  $|V|$  times
- DECREASE-KEY executed  $|E|$  times

$$\text{Time} = |V| \cdot T_{\text{EXTRACT-MIN}} + |E| \cdot T_{\text{DECREASE-KEY}}$$

Analysis: Look at different  $Q$  implementations.

Priority queue	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O( V )$	$O(1)$	$O( V ^2)$
binary heap	$O(\log  V )$	$O(\log  V )$	$O( E  \log  V )$
Fibonacci heap	$O(\log  V )$	$O(1)$	$O( V  \log  V  +  E )$
	amortized	amortized	worst case

## Dijkstra's Algorithm: Analysis

- $Q =$  unsorted array:
  - scan to find minimum
  - just index and update to change key

- $Q =$  Fibonacci heap

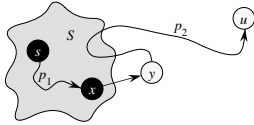
Note advantage of amortized analysis: Can use amortized Fibonacci heap bounds in analysis, as if they were worst-case bounds, and get (real) worst-case bounds on aggregate running time.

## Dijkstra's Algorithm: Correctness

**Correctness:** Prove that whenever  $u$  is added to  $S$ ,  
 $d[u] = \delta(s, u)$

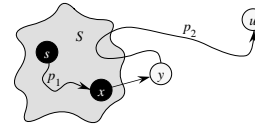
**Proof:**

- Note that  $\forall v \ d[v] \geq \delta(s, v)$
- Let  $u$  be first vertex added to  $S$  such that there is a path from  $s$  to  $u$  of length shorter than  $d[u]$   
 $\implies d[u] > \delta(s, u)$
- Let's consider the set  $S$  immediately before  $u$  is processed (i.e.,  $u$  is not yet in  $S$ , but is about to be picked next):



## Dijkstra's Algorithm: Correctness

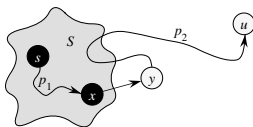
(Proof continued)



- Let  $y$  be first vertex  $\in V - S$  on actual shortest path from  $s$  to  $u$   
 $\implies d[y] = \delta(s, y)$   
 Because:
  - $d[x]$  is set correctly for  $y$ 's predecessor  $x \in S$  on the shortest path (by choice of  $u$  as first choice for which that's not true)
  - When we put  $x$  into  $S$ , we relaxed  $(x, y)$ , giving  $d[y]$  its correct value

## Dijkstra's Algorithm: Correctness

(Proof continued)



$$\begin{aligned}
 d[u] &> \delta(s, u) \\
 &= \delta(s, y) + \delta(y, u) && \text{(optimal substructure)} \\
 &= d[y] + \delta(y, u) \\
 &\geq d[y] && \text{(no negative weights)}
 \end{aligned}$$

- But  $d[u] > d[y] \implies$  algorithm would have chosen  $y$  to process next, not  $u$ . *Contradiction.*

## All pairs shortest path (APSP) with nonnegative weights

- In the APSP problem, we want to compute the shortest path between any two vertices  $u, v \in V$ .
  - The output has size  $O(|V|^2)$ , so we cannot hope to design a better than  $O(|V|^2)$ -time algorithm.
- We can solve the problem simply by running Dijkstra's algorithm  $|V|$  times  
 $\implies O(|V| \cdot |E| \log |V|)$  algorithm.
  - In the worst case (dense graph) the running time is  $O(|V|^3 \log |V|)$ .
- The Floyd-Warshall algorithm (see CLRS Exercise 25.2-4) runs in only  $O(|V|^3)$  time by working on adjacency matrix  $A$ :

```

FOR k = 1 to |V| do
  FOR i = 1 to |V| DO
    FOR j = 1 to |V| DO
      IF A[i, j] > A[i, k] + A[k, j]
        THEN
          A[i, j] = A[i, k] + A[k, j]
        FI
      OD
    OD
  OD

```

- Correctness:
  - We prove correctness by induction.
  - We will prove that, after each iteration of the outer loop on  $k$ , the following *invariant* holds: After the  $k$ th iteration (out of  $|V|$ ),  $A[i, j]$  contains the length of the shortest path from  $v_i$  to  $v_j$  that (apart from  $v_i$  and  $v_j$ ) only contains vertices of index at most  $k$ .  
 $\implies$  When algorithm terminates we have solved APSP.
  - Proof:
    - \* Invariant holds initially for  $k = 0$  (i.e., we start with adjacency matrix  $A$ , and the only allowed path from  $v_i$  to  $v_j$  is the edge from  $v_i$  to  $v_j$ ).
    - \* When “adding” vertex with index  $k$ , we explicitly check all new paths between  $v_i$  and  $v_j$  that pass through  $v_k$ , for all  $|V|^2$  pairs of  $v_i$  and  $v_j$ .
- Note:
  - We can easily produce adjacency-matrix from adjacency list in  $O(|V|^2)$  time.
  - Algorithm runs in  $O(|V|^3)$  time, even if the graph is sparse. Using algorithm based on Dijkstra's algorithm we will get much better performance for sparse graphs.
  - Using a Fibonacci heap, in which DECREASE-KEY operations take constant time amortized, Dijkstra's algorithm can be improved to  $O(|V|^2 \log |V| + |V| \cdot |E|) = O(|V|^3)$  time.