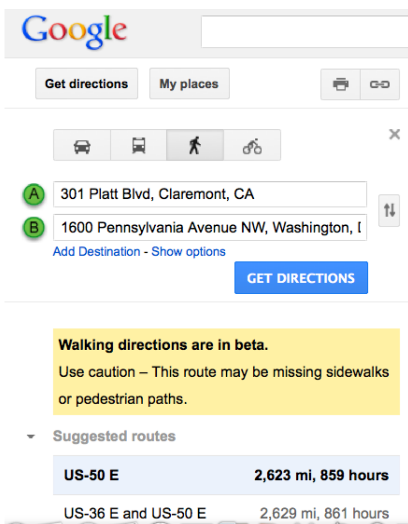
Shortest Path Algorithms

On Parallel Graph Algorithms and Their Applications

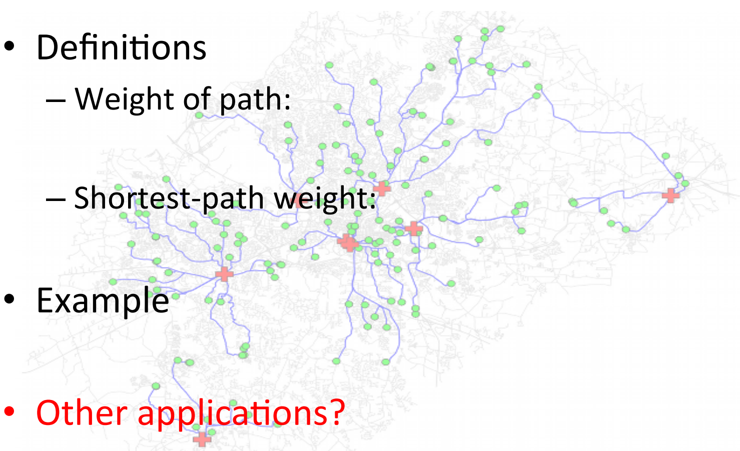
- Mahantesh Halappanavar
- Today's CS Colloquium!
  - Pomona

Shortest Paths!



Shortest paths Notation / Definition

- Definitions
  - Weight of path:
  - Shortest-path weight:
- Example
- Other applications?



4 Flavors of Shortest Paths

- Single-source:
- Single-destination:
- Single-pair:
- All-pairs:

Shortest path algorithms

Notation / Terminology:

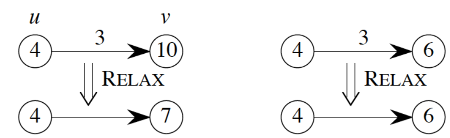
- Shortest-path estimate - *v.d*:
- Shortest-path tree - *v.π*:
- Relaxation

RELAX(u, v, w)

if $v.d > u.d + w(u, v)$

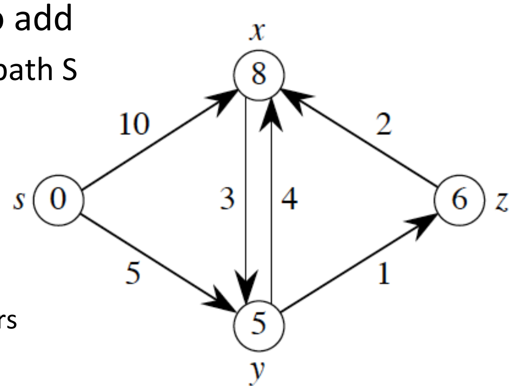
$\quad v.d = u.d + w(u, v)$

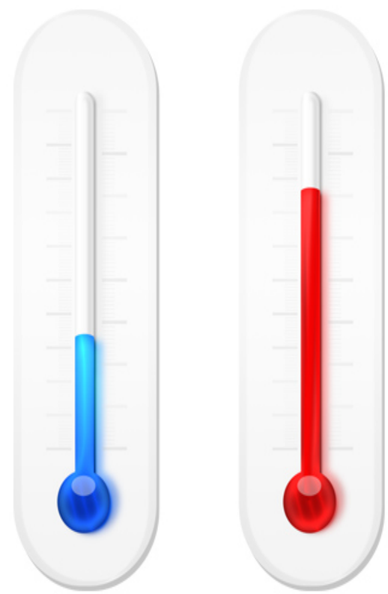
$\quad v.\pi = u$



Dijsktra's: The idea

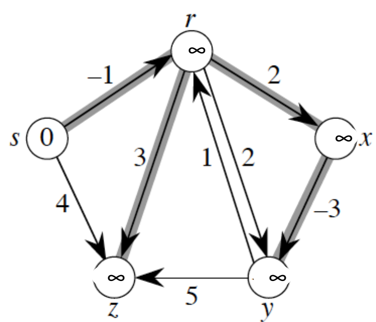
- Essentially a weighted version of BFS
- Like Prim's, makes greedy choice of which vertex / edge to add
  - Build shortest path S
  - Priority Q = V-S
    - Weighted by distance to source s
    - Extract-Min
    - Relax neighbors





Bellman-Ford: The idea

- Allows negative-weight edges
- DP approach
  - Calls RELAX on all edges repeatedly
  - Guaranteed to converge after V-1 iterations (though could converge sooner)
- Must check for negative cycles



Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

Initialize all the distances

do it |V| -1 times

iterate over all edges/vertices and apply update rule

Bellman-Ford algorithm

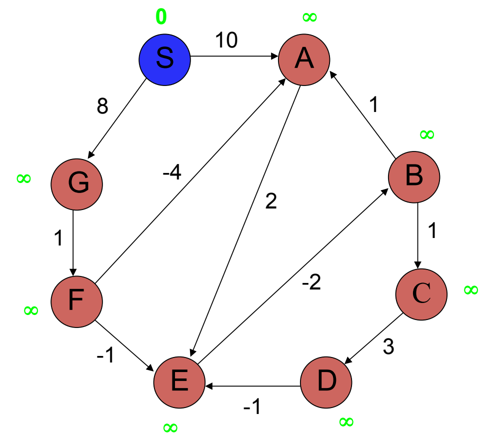
```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

check for negative cycles

Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

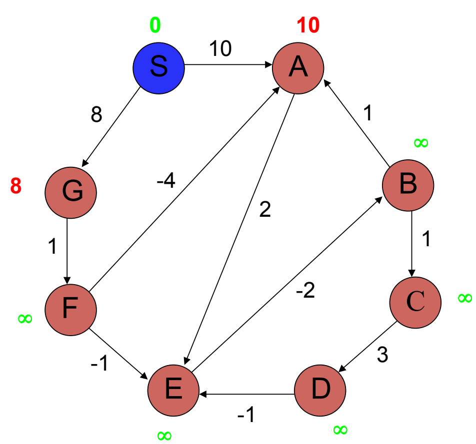
Bellman-Ford algorithm



Iteration: 0

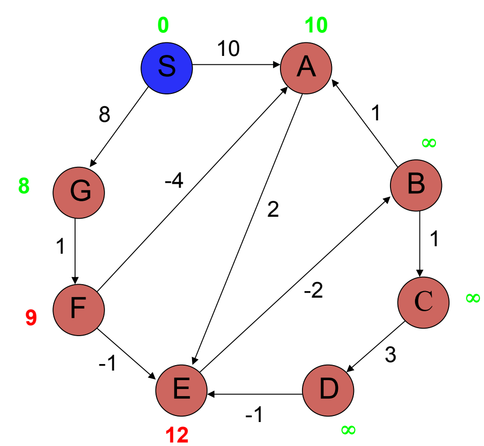
Try it!

Bellman-Ford algorithm



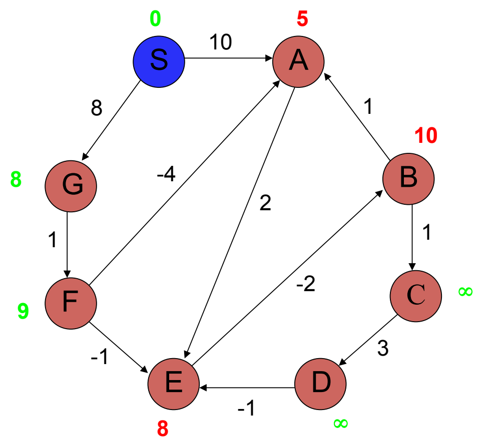
Iteration: 1

Bellman-Ford algorithm



Iteration: 2

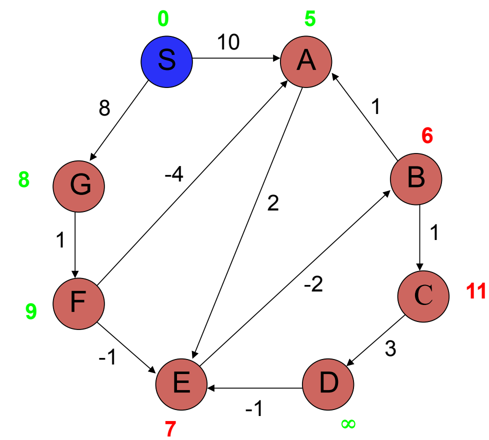
Bellman-Ford algorithm



Iteration: 3

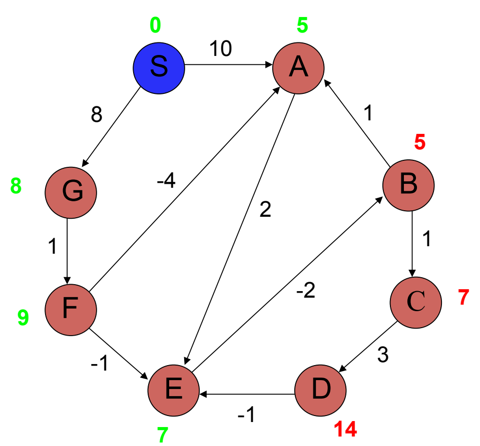
A has the correct distance and path

Bellman-Ford algorithm



Iteration: 4

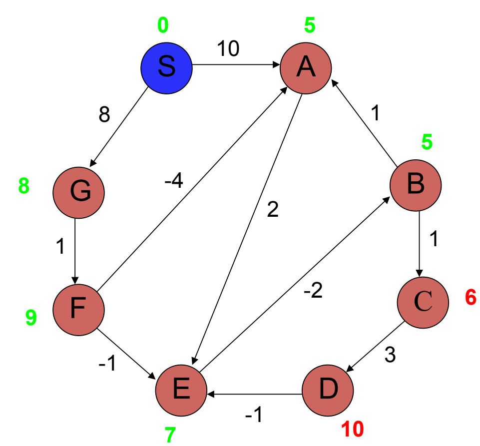
Bellman-Ford algorithm



Iteration: 5

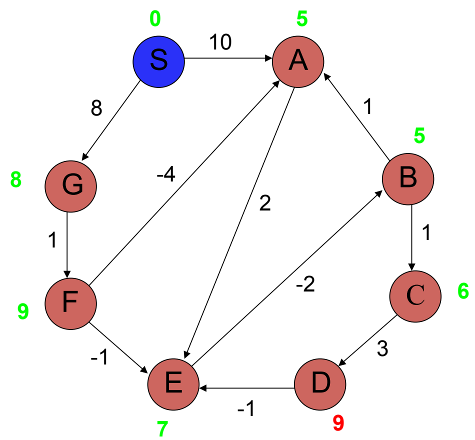
B has the correct distance and path

Bellman-Ford algorithm



Iteration: 6

Bellman-Ford algorithm



Iteration: 7

D (and all other nodes) have the correct distance and path

Correctness of Bellman-Ford

Loop invariant:

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

Correctness of Bellman-Ford

Loop invariant: After iteration i, all vertices with shortest paths from s of length i edges or less have correct distances

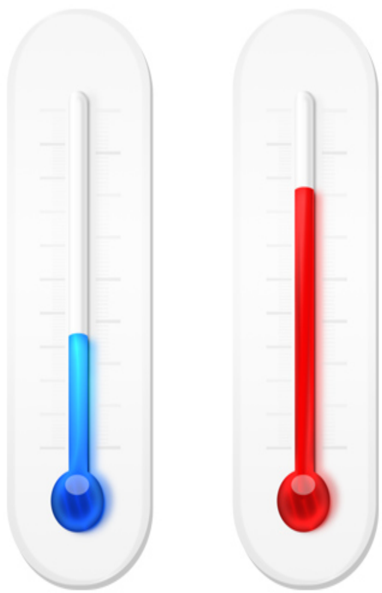
Falls out of a property we will prove later called Path-Relaxation!

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

Runtime of Bellman-Ford

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

$O(|V|\ |E|)$



Bellman-Ford vs. Dijkstra’s

Shortest Path Properties

Using the definitions on the board, try proving any of the next 5 properties on your worksheet!

Triangle inequality

For all $(u, v) \in E$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$.

Upper-bound property

Always have $v.d \geq \delta(s, v)$ for all v. Once $v.d = \delta(s, v)$, it never changes.

No-path property

If $\delta(s, v) = \infty$, then $v.d = \infty$ always.

Convergence property

If $s \leadsto u \to v$ is a shortest path, $u.d = \delta(s, u)$, and we call RELAX(u, v, w), then $v.d = \delta(s, v)$ afterward.

Path relaxation property

Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k. If we relax, *in order*, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $v_k.d = \delta(s, v_k)$.

All-Pairs Shortest Paths

- Run Bellman-Ford, once from each vertex
  - $O(V^2E)$
  - $O(V^4)$ if dense (E in $\theta(V^2)$)
- Run Dijkstra's, once from each vertex
  - Limited to graphs with non-negative edges
  - $O(VE\log V)$ with binary heap
    - $O(V^3\log V)$ if dense
  - $O(V^2\log V+VE)$ with Fibonacci heap
    - $O(V^3)$ if dense
- Wouldn't it be nice to get the best of both worlds:
  - General algorithm that runs on any graph
  - $O(V^3)$ worst-case runtimes?
  - No funny complex data structures

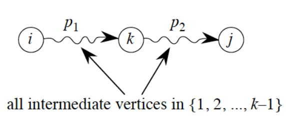
APSP Setup

- Input: a directed graph $G=(V,E)$, weight function w: E→ R, and assume vertices numbered 1…n
- Goal: create *nxn* distance matrix $D=(d_{ij})$ so that $d_{ij} = \delta(i,j)$ for all vertices i,j
- Initialization: create wright matrix W

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

Floyd-Warshall

- DP over the distance matrix (store values per edge, rather than per node)
  - Intermediate vertex:



  - Recursive formulation:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \, , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \, . \end{cases}$$

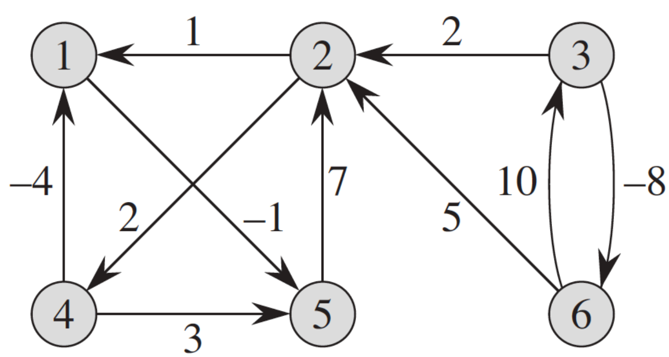
Floyd-Warshall

```
FLOYD-WARSHALL(W, n)
  D^(0) = W
  for k = 1 to n
      let D^(k) = (d_ij^(k)) be a new n × n matrix
      for i = 1 to n
          for j = 1 to n
              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
  return D^(n)
```

Time

$\Theta(n^3)$

FW Example



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

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \boxed{5} & -5 & 0 & \boxed{-2} \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

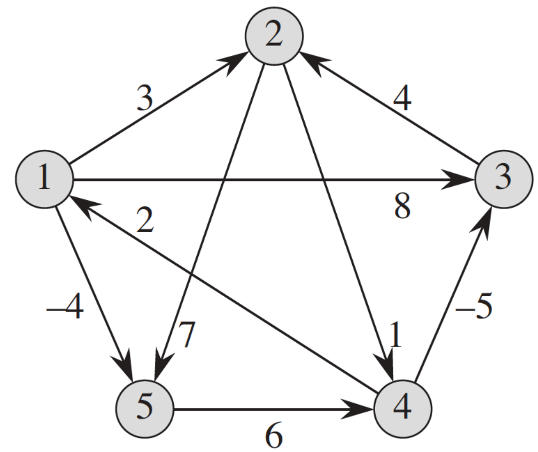
$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & \boxed{4} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

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

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

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

FW Example

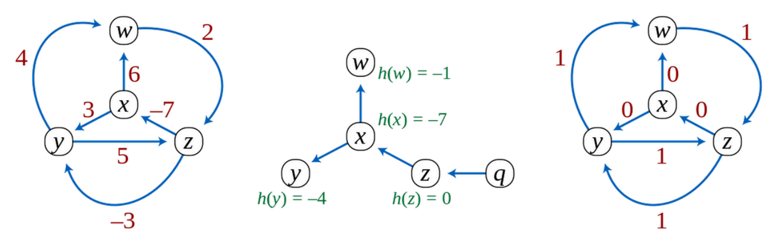


Johnson's Algorithm

- The Problem:
  - If graph is sparse, running Dijkstra fom each vertex would be faster
  - $O(V^2 \log V + VE)$ (Requires Fibonacci heap)
- Idea:
  - Reweight edges so that they are non-negative
    - And still give you correct shortest paths (paths that are min-weight in the original graph)
  1. Add new node q, connect to teach of the other nodes with zero-weight edges
  2. Run Bellman-Ford from q to calculate min path to every node h(v); return false if negative cycle is found
  3. Reweight all edge weights w' using the rule $w'(u,v) = w(u,v)+h(u)-h(v)$
  4. Remove q, run Dijsktra's from every node

Johnson's Algorithm

- The idea:
  - Reweight edges so that they are non-negative
    - And still give you correct shortest paths (paths that are min-weight in the original graph)



original graph
with negative edges

shortest path tree
found by Bellman-Ford

reweighted graph with
no negative edges

Credit: wikimedia.org

All pairs shortest paths

Simple approach
- Call Bellman-Ford |V| times
- $O(|V|^2 |E|)$

Floyd-Warshall – $\Theta(|V|^3)$

Johnson's algorithm – $O(|V|^2 \log |V| + |V| |E|)$

Pettie's algorithm – $O(nm+n^2 \log \log n)$

Snowball [Planken et al. 2011]