Shortest Path Algorithms

Overview…

- Last time: Minimum Spanning Tree
  - Prim's
  - Kruskal's
- Today: Shortest Path
  - Definition / properties
  - Single-source
    - DP
    - Greedy
- Next time:
  - All-pairs

Minimum spanning trees

What is the lowest weight set of edges that connects all vertices of an undirected graph with positive weights

Input: An undirected, positive weight graph, G=(V,E)

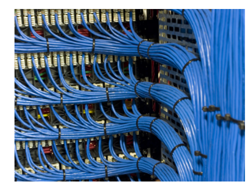
Output: A tree T=(V,E') where E' $\subseteq$ E that minimizes

$$weight(T) = \sum_{e \in E'} w_e$$

Applications?

Connectivity

- Networks (e.g. communications)
- Circuit design/wiring



Hub/spoke models (e.g. flights, transportation)

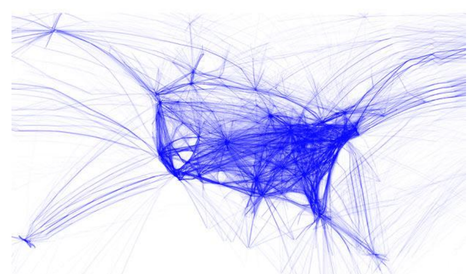
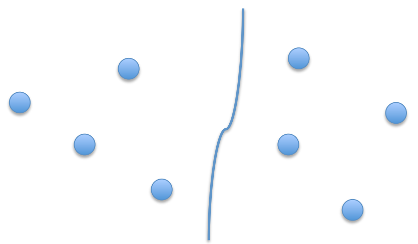
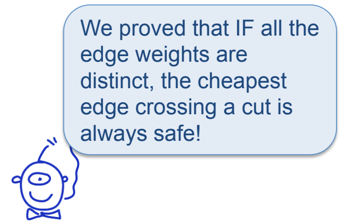


Photo Credit: Aaron Koblin

The "Generic" Greedy MST Algorithm

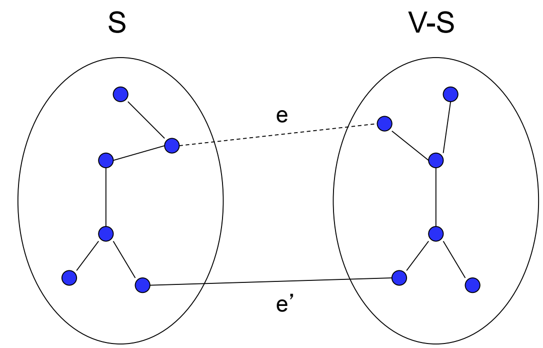
```
MST(Graph G):
  T = {}  # The tree we're growing!
  while |T| < n-1  # n is num vertices
     Find a safe edge e in G – T
     T = T + e
  return T
```





Minimum cut property

Given a partion S, let edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge *e*.



Consider an MST with edge e' that is not the minimum edge

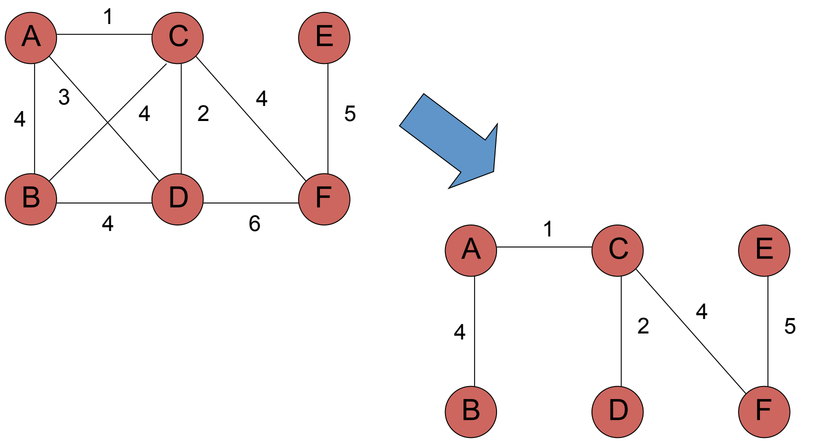
Kruskal's Algorithm (The idea)

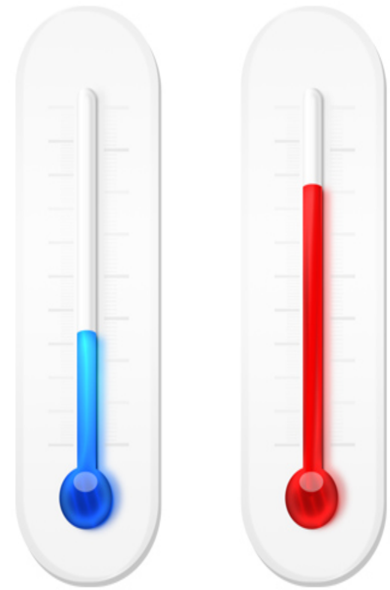
- Start with each vertex being its own component
- Repeated merge components into on by choosing the light edge that connects them
  - Sort all edge in monotonically increasing order by weight
  - Use a disjoint-set data structure to determine whether an edge connects vertices in difference components

Prim's Algorithm (The Idea)

- Incrementally build tree
  - Pick any root
  - At each step, find light edge that crosses the cut T, G-T
- Implemented using a priority queue
  - Priorities represent costs connect to T
  - Extract-Min chooses the edge to add
  - Decrease-Key updates priorities of crossing edges

MST example





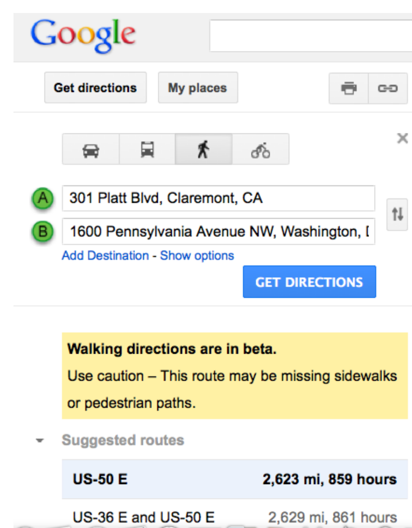
Worksheet:

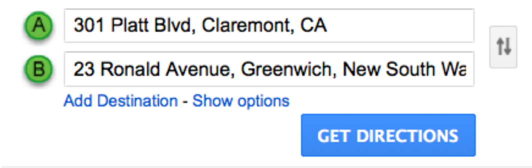
After an unexpected budget shortfall Cheapville the mayor decides there is only enough money to fix the path that connect her house with city hall!

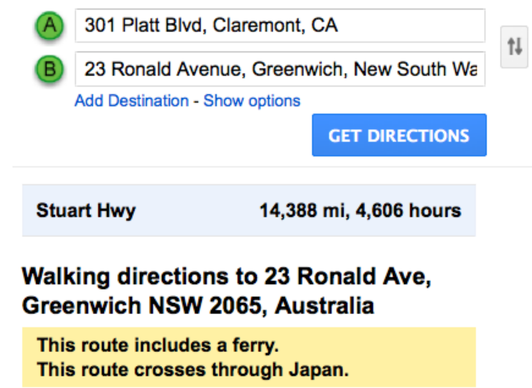
Theorem: Given a graph, G, and a m.s.t. of G, T, the min-weight path between any pair of nodes *x,y* is encoded by T.

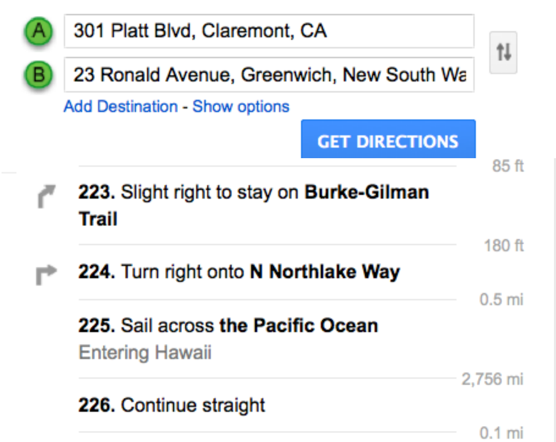
- *Prove by induction on path length; or*
- *Give counter-example*

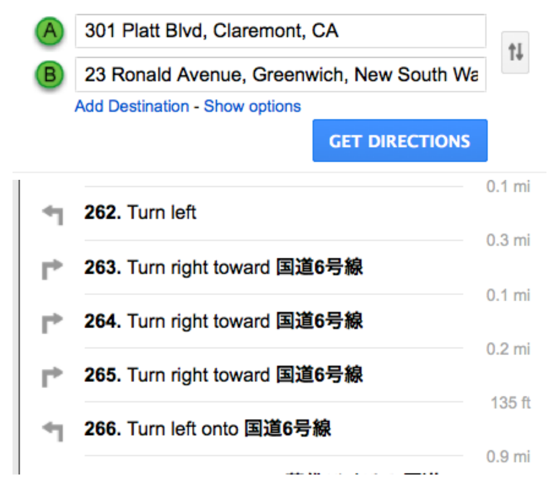
Shortest Paths!



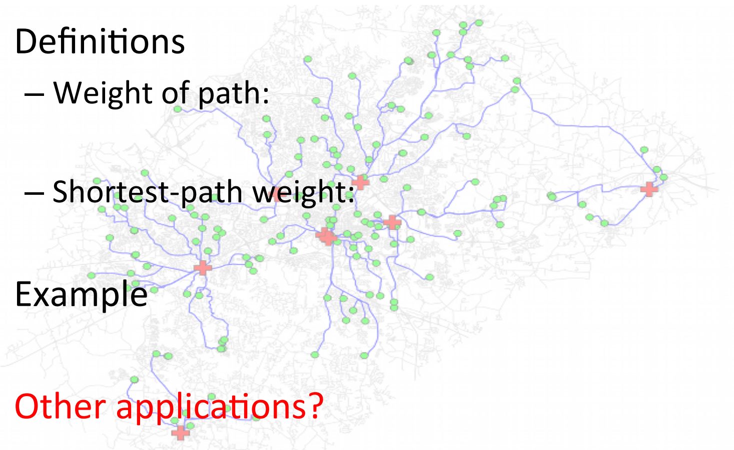

301 Platt Blvd, Claremont, CA
23 Ronald Avenue, Greenwich, New South Wa
Add Destination - Show options
GET DIRECTIONS


301 Platt Blvd, Claremont, CA
23 Ronald Avenue, Greenwich, New South Wa
Add Destination - Show options
GET DIRECTIONS
Stuart Hwy
14,388 mi, 4,606 hours
Walking directions to 23 Ronald Ave, Greenwich NSW 2065, Australia
This route includes a ferry.
This route crosses through Japan.


301 Platt Blvd, Claremont, CA
23 Ronald Avenue, Greenwich, New South Wa
Add Destination - Show options
GET DIRECTIONS
85 ft
223. Slight right to stay on Burke-Gilman Trail
180 ft
224. Turn right onto N Northlake Way
0.5 mi
225. Sail across the Pacific Ocean
Entering Hawaii
2,756 mi
226. Continue straight
0.1 mi


301 Platt Blvd, Claremont, CA
23 Ronald Avenue, Greenwich, New South Wa
Add Destination - Show options
GET DIRECTIONS
0.1 mi
262. Turn left
0.3 mi
263. Turn right toward 国道6号線
0.1 mi
264. Turn right toward 国道6号線
0.2 mi
265. Turn right toward 国道6号線
135 ft
266. Turn left onto 国道6号線
0.9 mi

Shortest paths
Notation / Definition



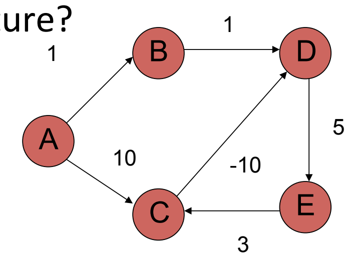
- Definitions
  - Weight of path:
  - Shortest-path weight:
- Example
- Other applications?

4 Flavors of Shortest Paths

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

Can we…?

- … have negative weight edges?
- …have cycles?
- …find optimal substructure?

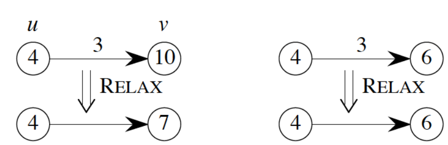


Today's goal: Single-source shortest path algorithm

Notation / Terminology:

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

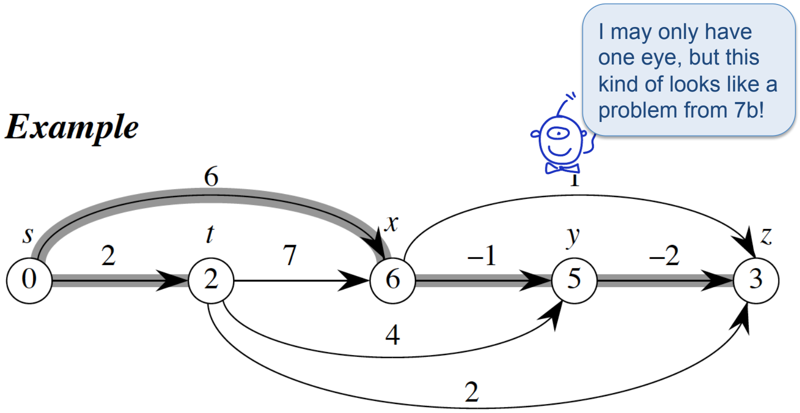
RELAX(u, v, w)
  if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$

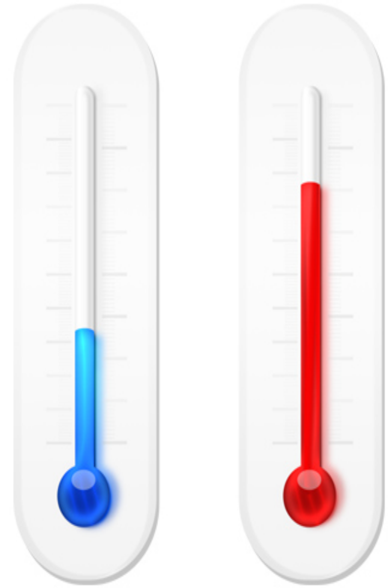


Worksheet: SSSP on DAGs

- What if we are guaranteed to have no cycles?

Example



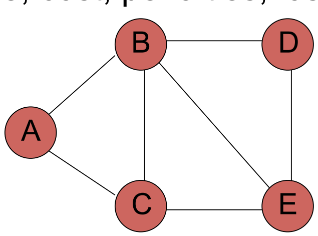


Shortest paths

Can think of weights as representing any measure that:

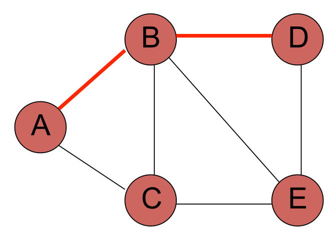
1. accumulates linearly along a path, and
2. we want to minimize.

Examples: time, cost, penalties, loss



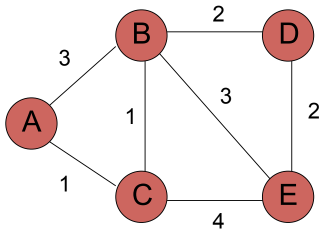
Shortest paths

BFS



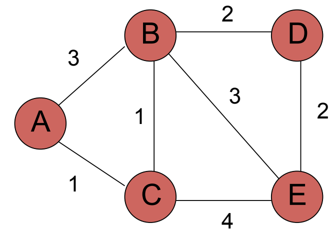
Shortest paths

What is the shortest path from a to d?



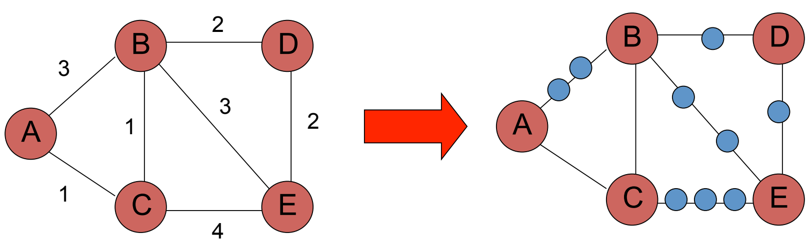
Shortest paths

We can still use BFS



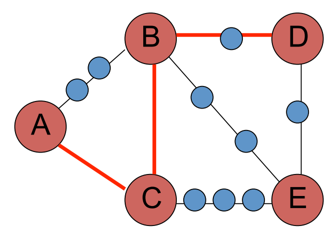
Shortest paths

We can still use BFS



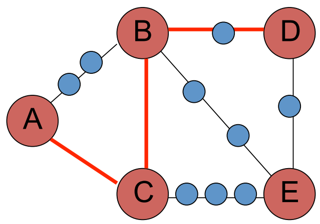
Shortest paths

We can still use BFS



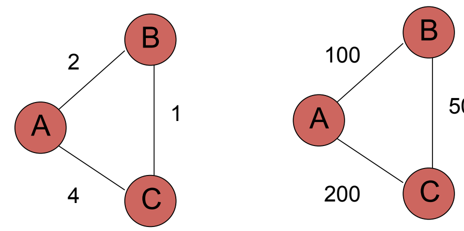
Shortest paths

What is the problem?

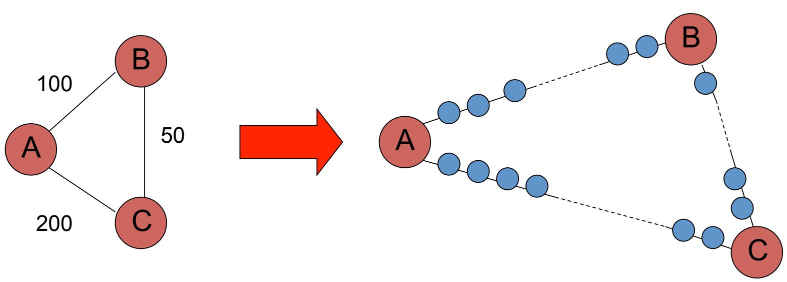


Shortest paths

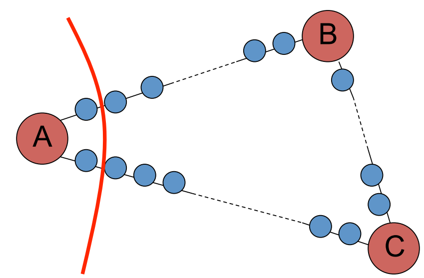
Running time is dependent on the weights



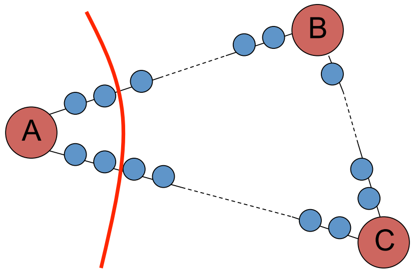
Shortest paths



Shortest paths

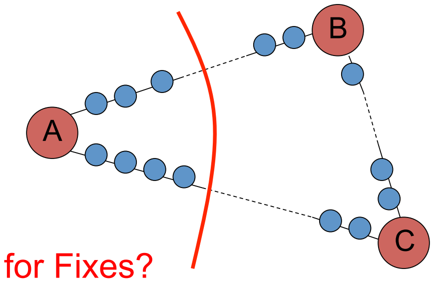


Shortest paths



Shortest paths

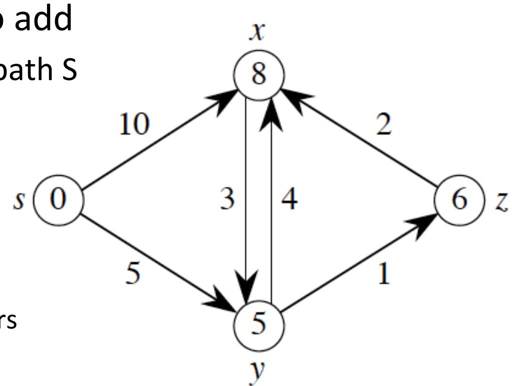
Nothing will change as we expand the frontier until we've gone out 100 levels



Ideas for Fixes?

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



Dijkstra's algorithm

```
DIJKSTRA(G, s)
  for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
  dist[s] ← 0
  Q ← MAKEHEAP(V)
  while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
              if dist[v] > dist[u] + w(u, v)
                    dist[v] ← dist[u] + w(u, v)
                    DECREASEKEY(Q, v, dist[v])
                    prev[v] ← u
```

Dijkstra's algorithm

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra's algorithm

prev keeps track of
the shortest path

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra's algorithm

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra's algorithm

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra's algorithm

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

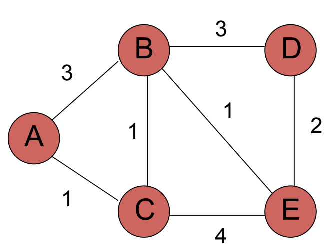
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Prim's vs. Dijkstra's algorithm

```
PRIM(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MAKEHEAP(key)
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

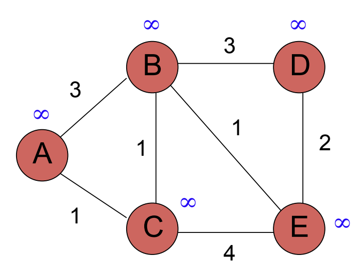
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

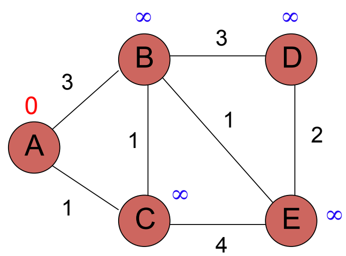


Try it!

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

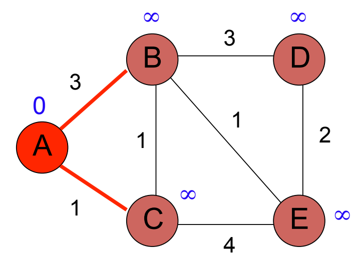


```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



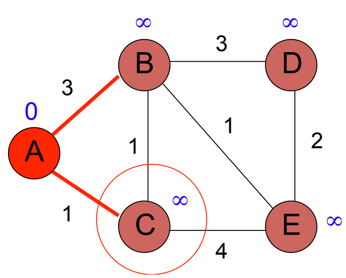
Heap	
A	0
B	∞
C	∞
D	∞
E	∞


```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

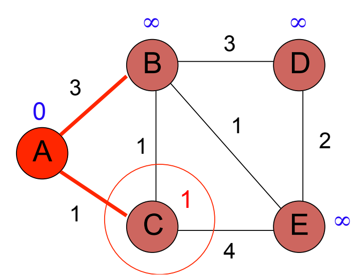
Heap	
B	∞
C	∞
D	∞
E	∞


```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

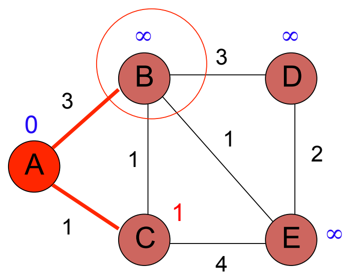
Heap	
B	∞
C	∞
D	∞
E	∞


```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

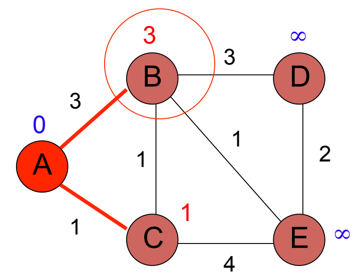
Heap	
C	1
B	∞
D	∞
E	∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



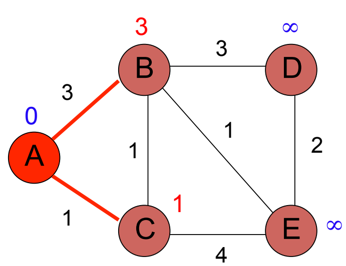
Heap	
C	1
B	∞
D	∞
E	∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



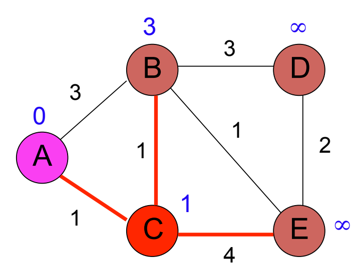
Heap	
C	1
B	3
D	∞
E	∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



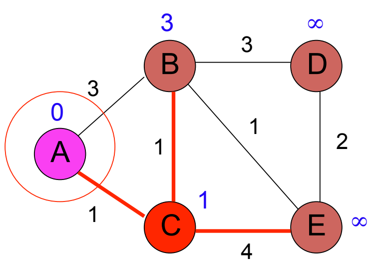
Heap	
C	1
B	3
D	∞
E	∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap	
B	3
D	∞
E	∞

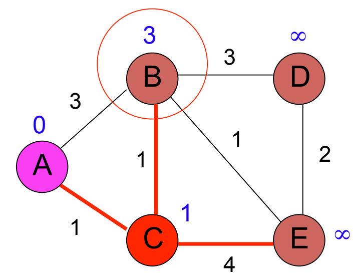
```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B 3
D ∞
E ∞

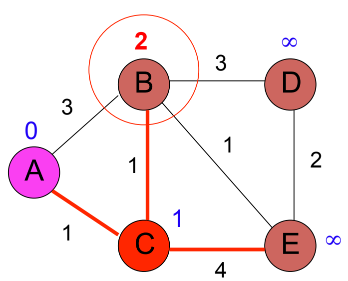
```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B 3
D ∞
E ∞

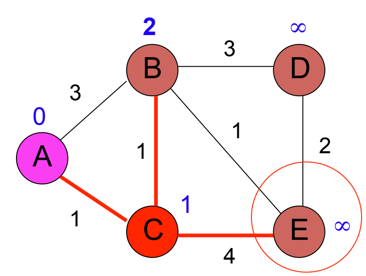
```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B 2
D ∞
E ∞

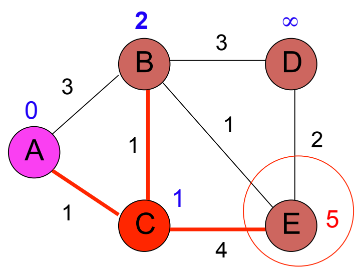
```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B 2
D ∞
E ∞

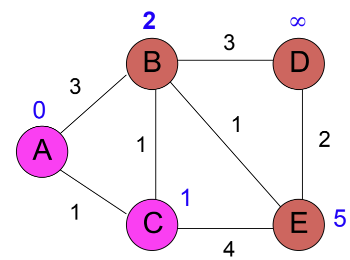
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B	2
E	5
D	∞


```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

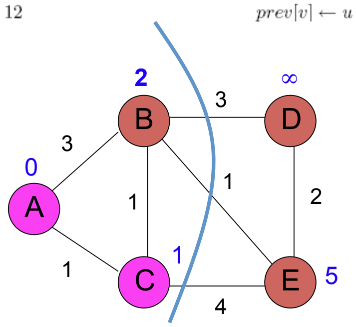



Heap

B	2
E	5
D	∞

Frontier?


```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

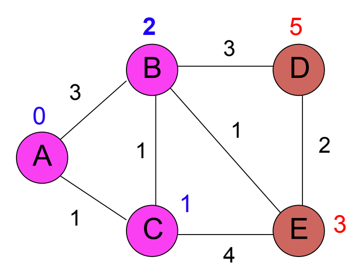



Heap

B	2
E	5
D	∞

All nodes reachable
from starting node
within a given distance


```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

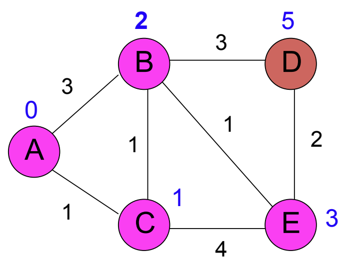
Heap

E	3
D	5

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

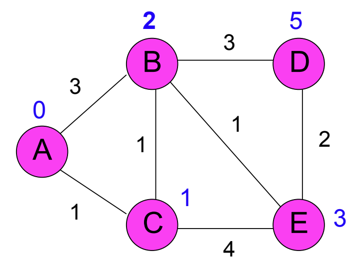
Heap

D 5



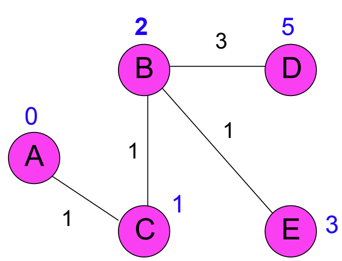

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```


Heap




```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```


Heap



Is Dijkstra's algorithm correct?

Invariant:


```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Is Dijkstra's algorithm correct?

Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

why?

Is Dijkstra's algorithm correct?

Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v

- The only time a vertex gets visited is when the distance from s to that vertex is smaller than the distance to any remaining vertex

- Therefore, there cannot be any other path that hasn't been visited already that would result in a shorter path

Running time?

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

Running time?

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)          1 call to MakeHeap
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

Running time?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)                          |V| iterations
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Running time?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)                |V| calls
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Running time?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])   O(|E|) calls
                        prev[v] ← u
```

Running time?

Depends on the heap implementation

	1 MakeHeap	|V| ExtractMin	|E| DecreaseKey	Total
Array	O(|V|)	O(|V|^2)	O(|E|)	O(|V|^2)
Bin heap	O(|V|)	O(|V| log |V|)	O(|E| log |V|)	O((|V|+|E|) log |V|) O(|E| log |V|)

Running time?

Depends on the heap implementation

	1 MakeHeap	$\lvert V \rvert$ ExtractMin	$\lvert E \rvert$ DecreaseKey	Total
Array	$O(\lvert V \rvert)$	$O(\lvert V \rvert^2)$	$O(\lvert E \rvert)$	$O(\lvert V \rvert^2)$
Bin heap	$O(\lvert V \rvert)$	$O(\lvert V \rvert \log \lvert V \rvert)$	$O(\lvert E \rvert \log \lvert V \rvert)$	$O((\lvert V \rvert+\lvert E \rvert) \log \lvert V \rvert)$ $O(\lvert E \rvert \log \lvert V \rvert)$

Is this an improvement? If $\lvert E \rvert < \lvert V \rvert^2 / \log \lvert V \rvert$

Running time?

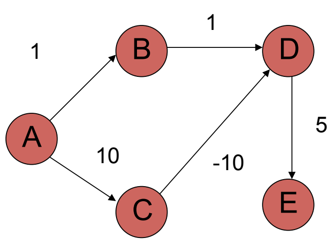
Same as Prim's algorithm

Depends on the heap implementation

	1 MakeHeap	$\lvert V \rvert$ ExtractMin	$\lvert E \rvert$ DecreaseKey	Total
Array	$O(\lvert V \rvert)$	$O(\lvert V \rvert^2)$	$O(\lvert E \rvert)$	$O(\lvert V \rvert^2)$
Bin heap	$O(\lvert V \rvert)$	$O(\lvert V \rvert \log \lvert V \rvert)$	$O(\lvert E \rvert \log \lvert V \rvert)$	$O((\lvert V \rvert+\lvert E \rvert) \log \lvert V \rvert)$ $O(\lvert E \rvert \log \lvert V \rvert)$
Fib heap	$O(\lvert V \rvert)$	$O(\lvert V \rvert \log \lvert V \rvert)$	$O(\lvert E \rvert)$	$O(\lvert V \rvert \log \lvert V \rvert + \lvert E \rvert)$

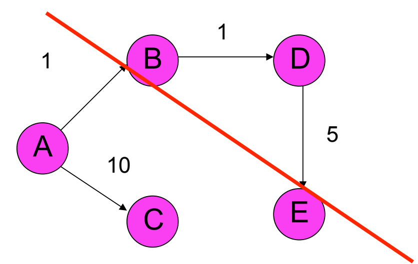
What about Dijkstra's on…?

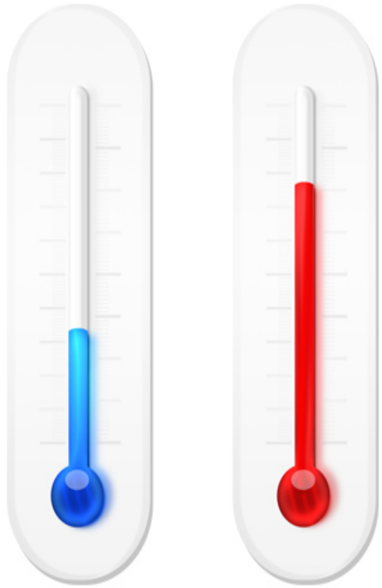
Worksheet: Try it!



What about Dijkstra's on…?

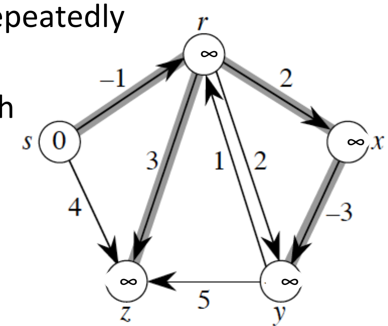
Dijkstra's algorithm only works for positive edge weights





Bellman-Ford: The idea

- Allows negative-weight edges
- DP approach
  - Computes *v.d* and *v.π* for all edges
  - 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

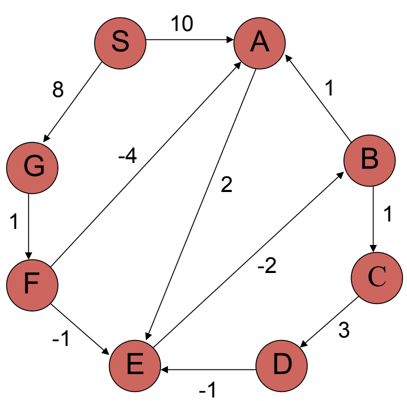
```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v)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

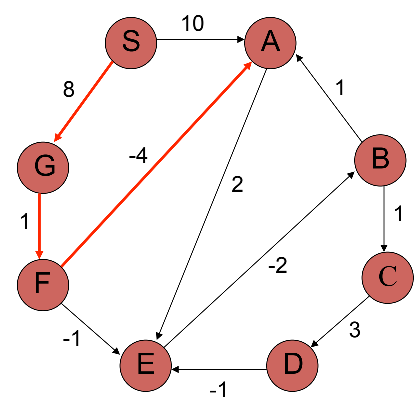
Bellman-Ford algorithm



How many edges is the shortest path from s to:

A:

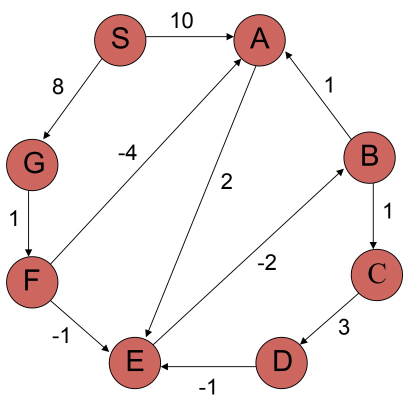
Bellman-Ford algorithm



How many edges is the shortest path from s to:

A: 3

Bellman-Ford algorithm

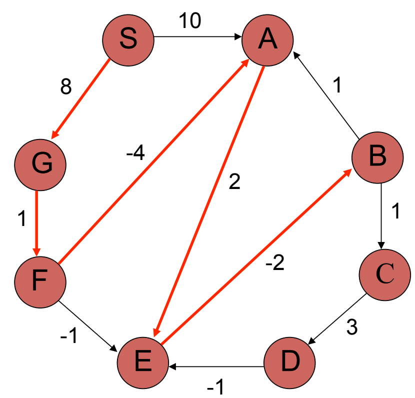


How many edges is the shortest path from s to:

A: 3

B:

Bellman-Ford algorithm

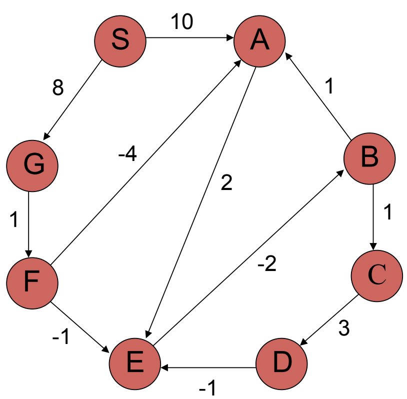


How many edges is the shortest path from s to:

A: 3

B: 5

Bellman-Ford algorithm



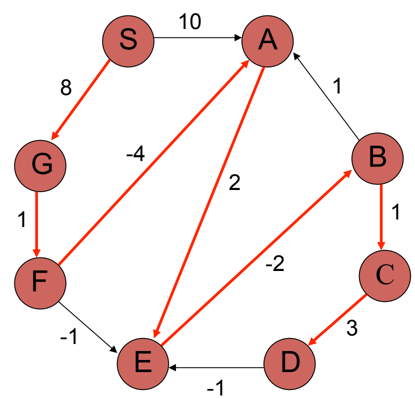
How many edges is the shortest path from s to:

A: 3

B: 5

D:

Bellman-Ford algorithm



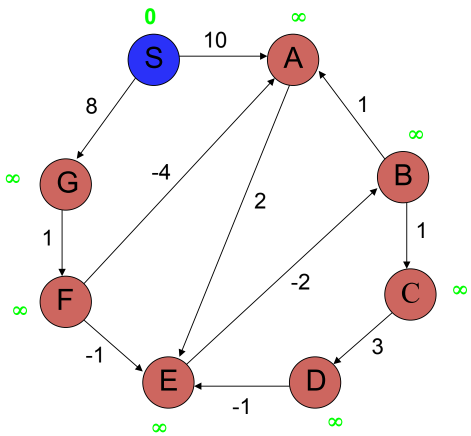
How many edges is the shortest path from s to:

A: 3

B: 5

D: 7

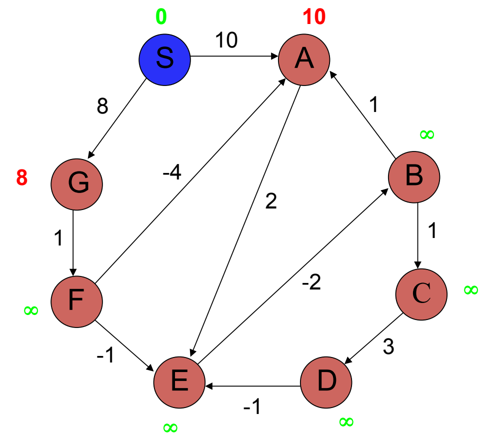
Bellman-Ford algorithm



Iteration: 0

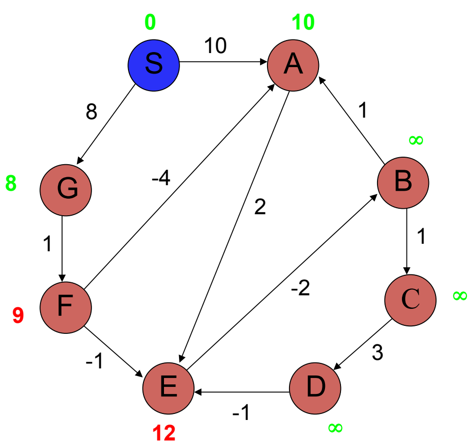
Try it!

Bellman-Ford algorithm



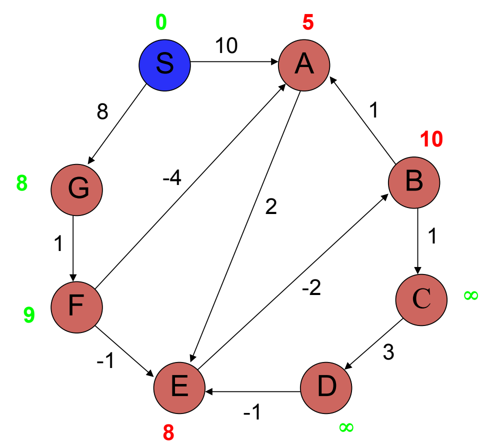
Iteration: 1

Bellman-Ford algorithm



Iteration: 2

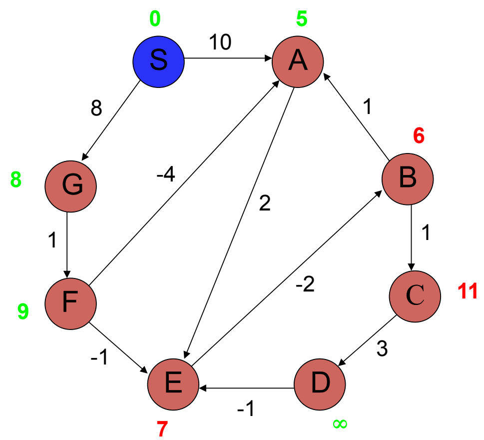
Bellman-Ford algorithm



Iteration: 3

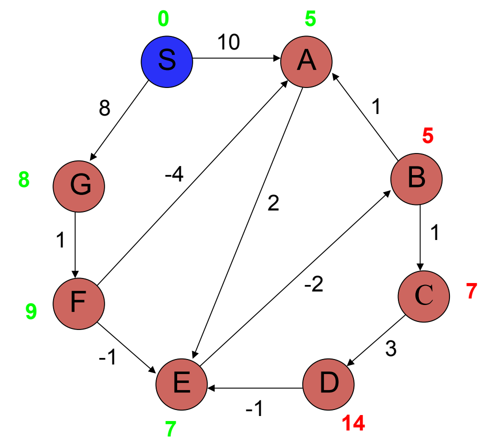
A has the correct distance and path

Bellman-Ford algorithm



Iteration: 4

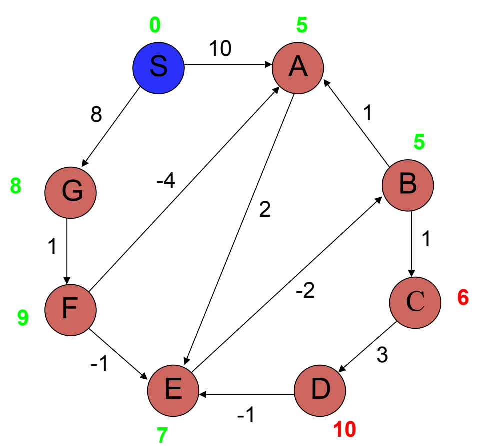
Bellman-Ford algorithm



Iteration: 5

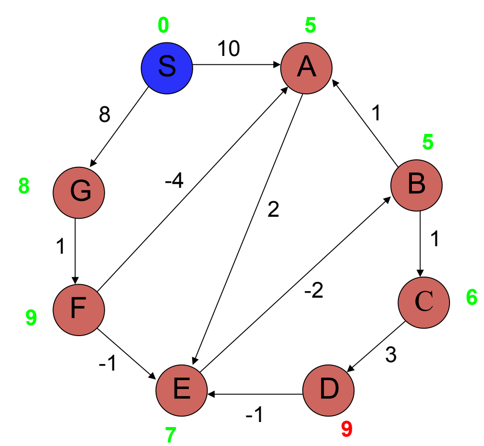
B has the correct distance and path

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
```

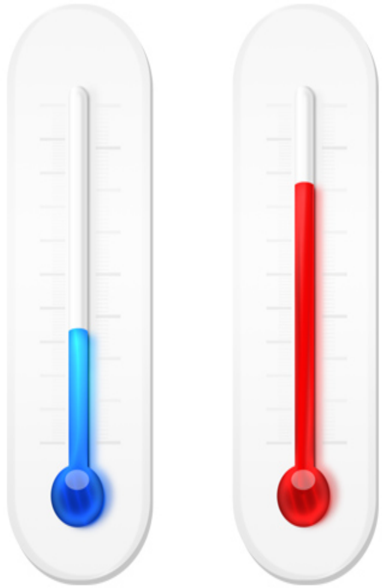
Runtime of Bellman-Ford

```
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
```

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

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
```



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 $\text{RELAX}(u, v, w)$, 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

Simple approach

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

Floyd-Warshall – $\Theta(|V|^3)$

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