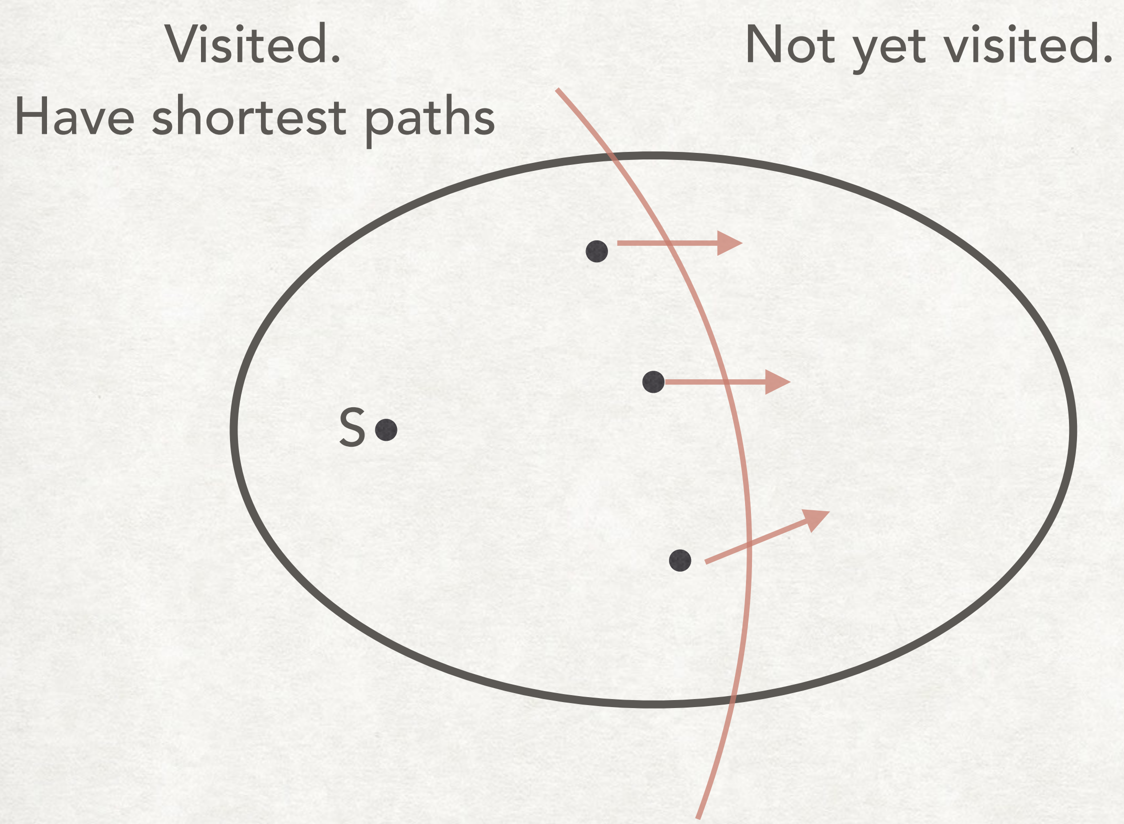
CS 170 DISCUSSION 5

SHORTEST PATHS AND SPANNING TREES

Raymond Chan
UC Berkeley Fall 17

DIJKSTRA'S SHORTEST PATH

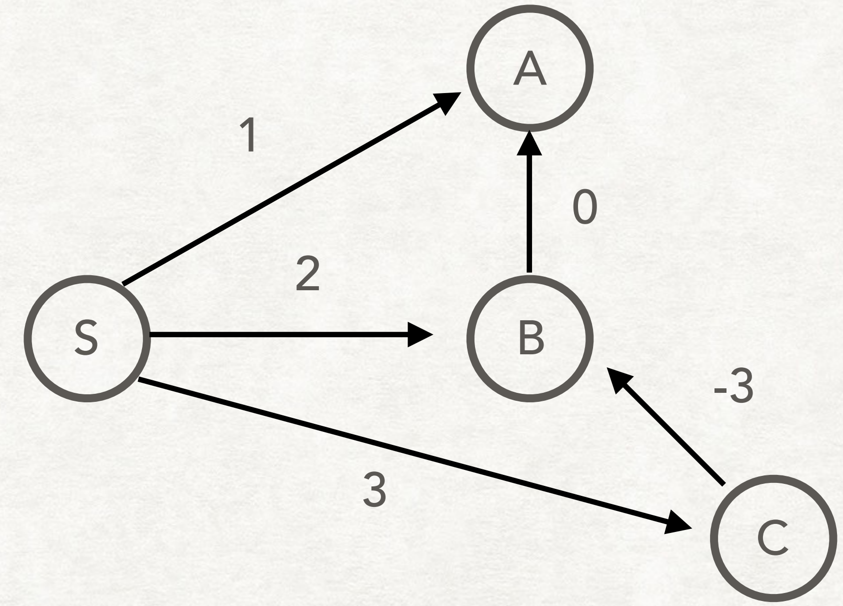
- Find shortest path from s to all other vertices.
- Once we have computed the shortest path to a vertex, we don't revisit it again.




```
dijkstra (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  PQ.add(G.V, infinity)
  PQ.add(s,0)
  while PQ not empty:
    u = PQ.DeleteMin()
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
        PQ.DecreaseKey(v, d[v])
```

NEGATIVE EDGE WEIGHTS

- Thus when we have negative edge weights. The negative weights won't propagate to other nodes if we have already visited nodes with incoming edges with negative cost.




```
dijkstra (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  PQ.add(G.V, infinity)
  PQ.add(s,0)
  while PQ not empty:
    u = PQ.DeleteMin()
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
        PQ.DecreaseKey(v, d[v])
```

Start at S
Process A. d[a] = 1
Process B. d[b] = 2
Process C. d[c] = 3
Process A. d[a] = 1.
Process B. d[b] = 2. d[a] = 1
Process C. d[c] = 3. d[b] = 0
New distance for B, but B not in PQ
Won't update D

BELLMAN-FORD ALGORITHM

- Solution: Update shortest path distances values for all vertices if we have found a shorter path.
- How many times to do this for?
- Furthest vertex from source vertex *s* (in terms of number of edges) can at most be |V| - 1 edges away.
- Update only |V| - 1 times.
- Demo

BELLMAN-FORD ALGORITHM

- Solution: Update shortest path distances values for all vertices if we have found a shorter path.
- Update only |V| - 1 times.

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

```
update((u, v)):
  dist(v) = min(dist(v), dist(u) + w(u, l)

bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      update((u, v))
```

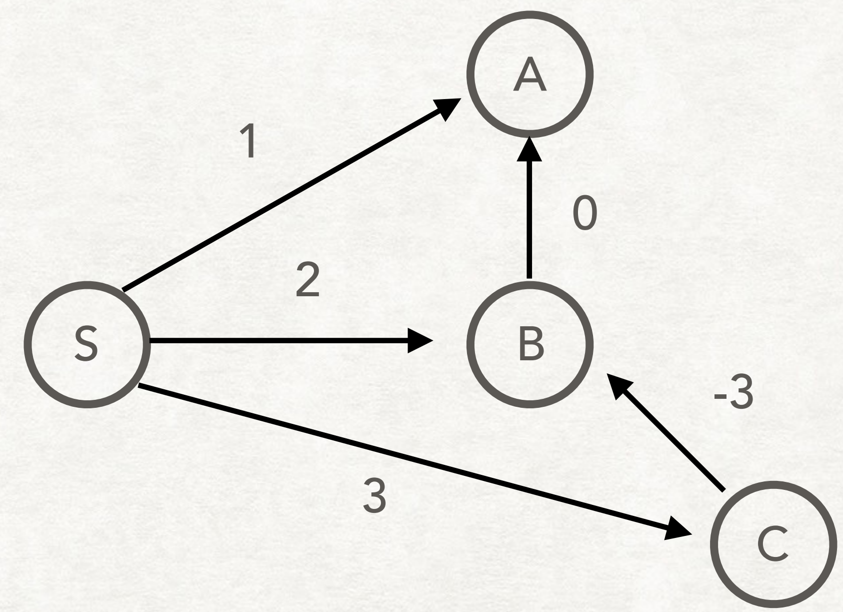
O(|V||E|)

BELLMAN-FORD ALGORITHM

- Order of iteration on edges matter.
- Shortest paths could converge sooner or later.
- There exist a path from source *s* to *u* of length *dist(u)* (unless it's ∞)
- After *i* iterations, have found shortest path from *s* to *u* that uses *i* or fewer edges.

BELLMAN-FORD ALGORITHM

Iteration Order
(S, B), (B, A), (S, A), (C, B), (S, C)



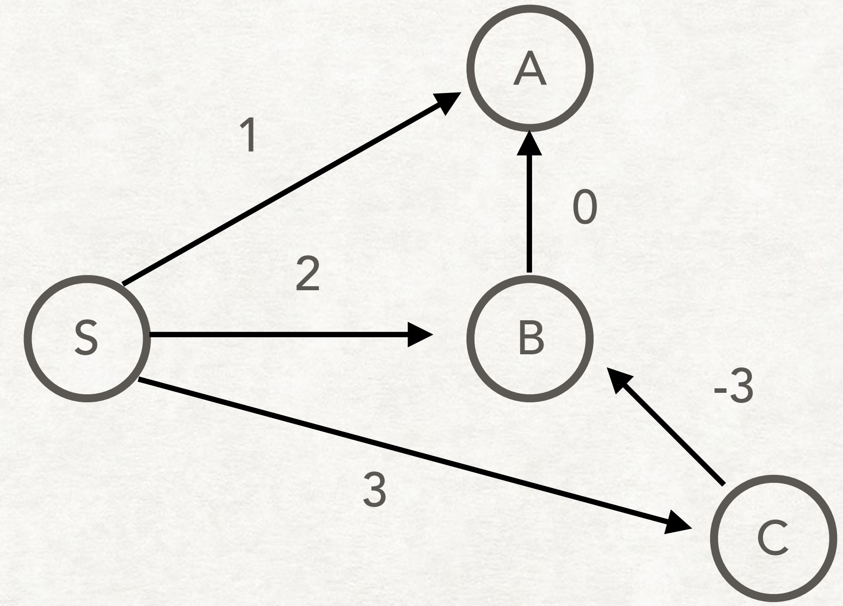
	0	1	2	3
S	0, S			
A	∞			
B	∞			
C	∞			

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order

(S, B), (B, A), (S, A), (C, B), (S, C)

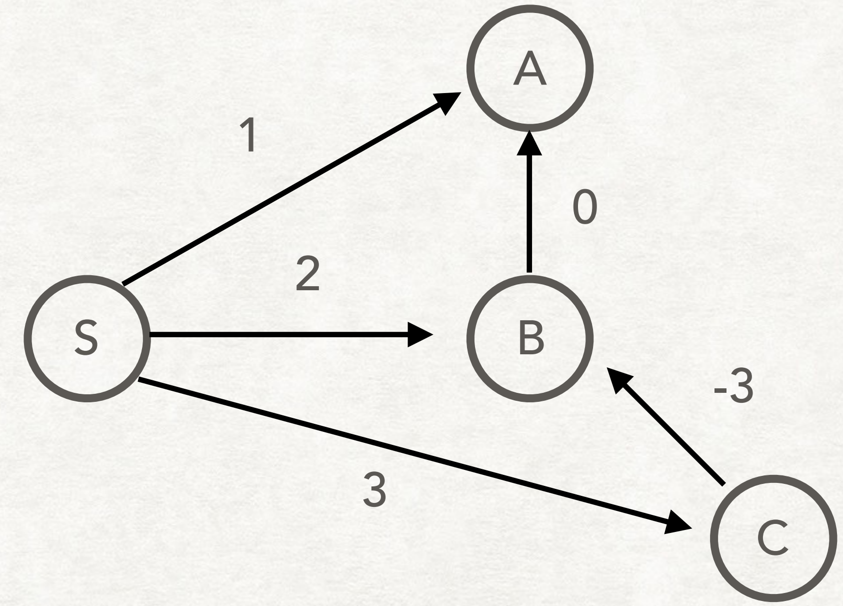


	0	1	2	3
S	0, S	0, S		
A	∞			
B	∞	**2, S**		
C	∞			

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order
(S, B), **(B, A)**, (S, A), (C, B), (S, C)



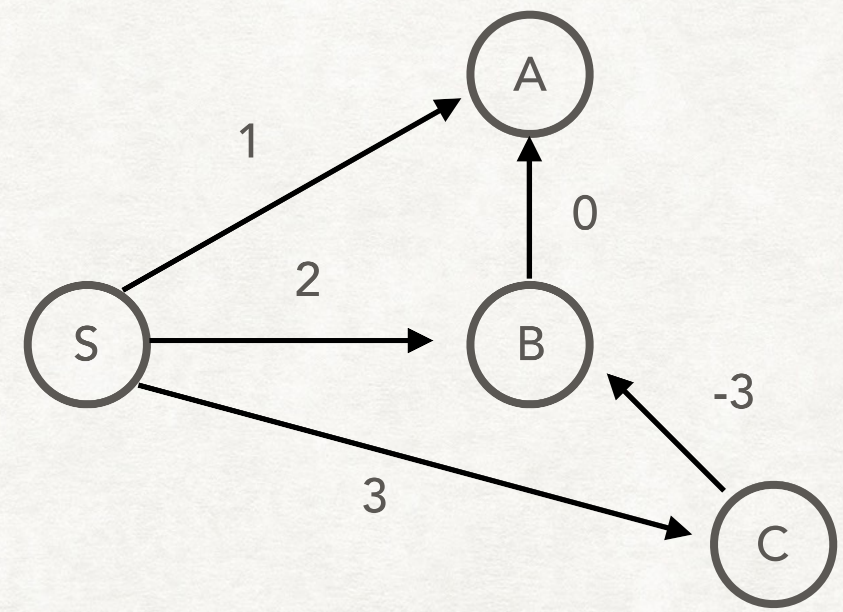
	0	1	2	3
S	0, S	0, S		
A	∞	**2, B**		
B	∞	2, S		
C	∞			

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order

(S, B), (B, A), **(S, A)**, (C, B), (S, C)

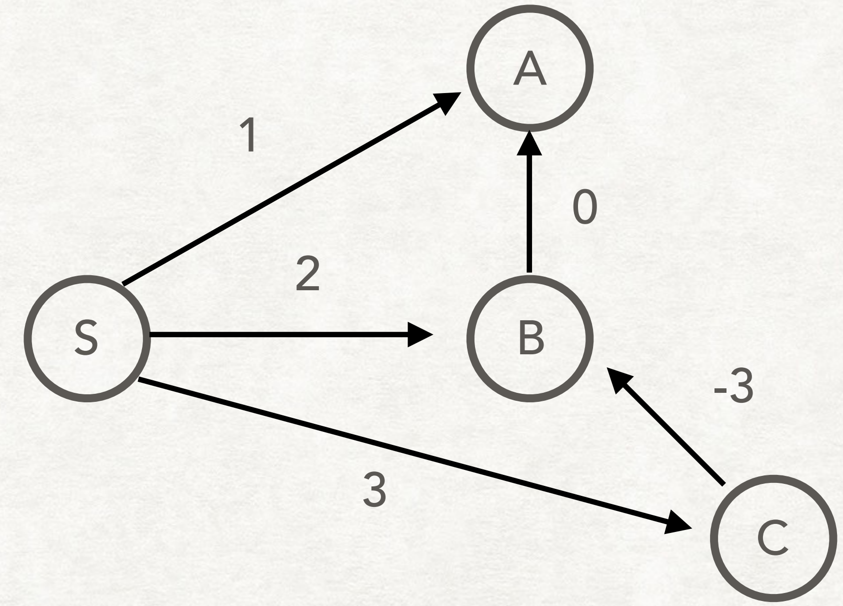


	0	1	2	3
S	0, S	0, S		
A	∞	**1, S**		
B	∞	2, S		
C	∞			

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order
(S, B), (B, A), (S, A), **(C, B)**, (S, C)



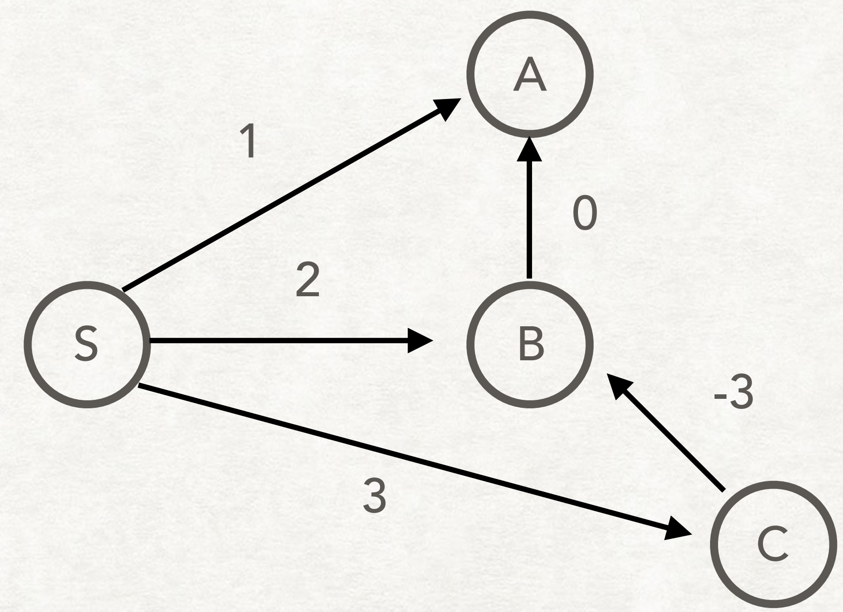
	0	1	2	3
S	0, S	0, S		
A	∞	1, S		
B	∞	2, S		
C	∞			

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order

(S, B), (B, A), (S, A), (C, B), **(S, C)**



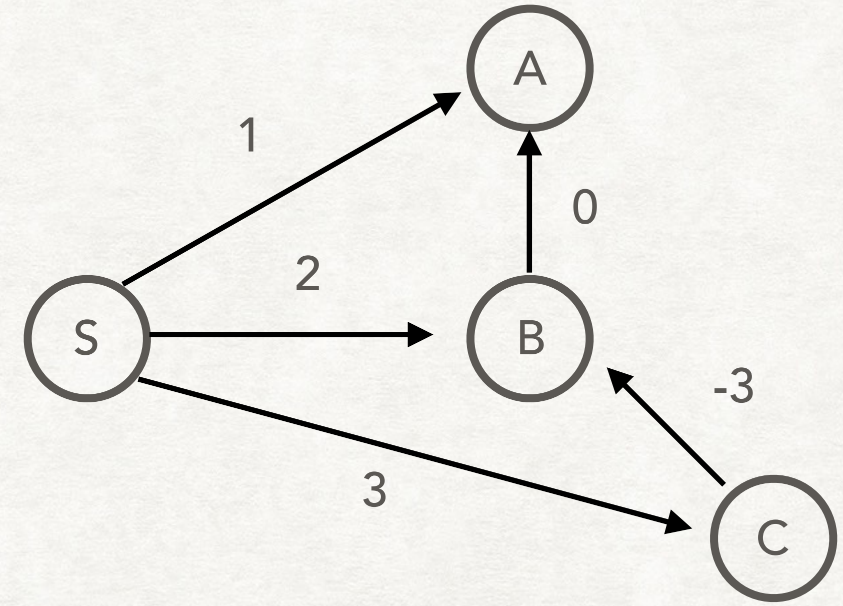
	0	1	2	3
S	0, S	0, S		
A	∞	1, S		
B	∞	2, S		
C	∞	**3, S**		

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order

(S, B), (B, A), (S, A), (C, B), (S, C)

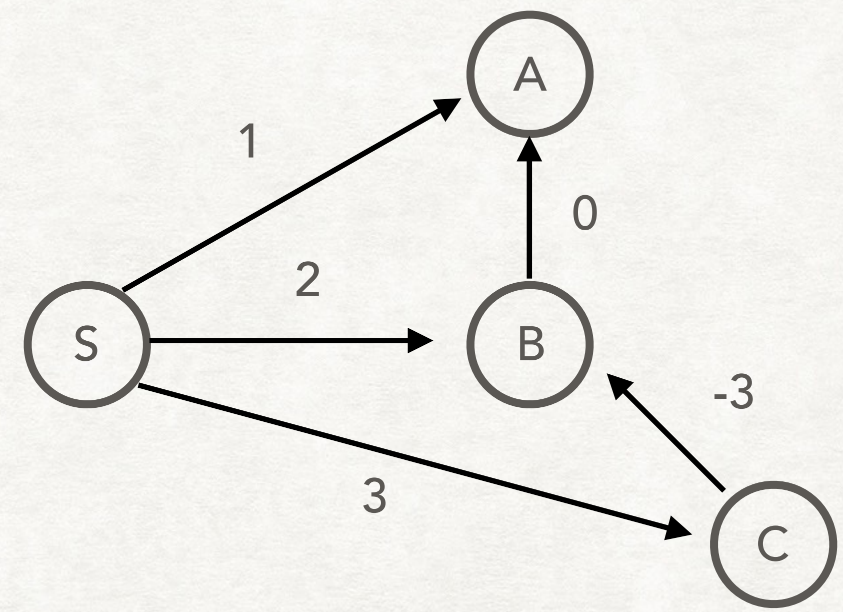


	0	1	2	3
S	0, S	0, S	0, S	
A	∞	1, S	1, S	
B	∞	2, S	2, S	
C	∞	3, S	3, S	

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order
(S, B), **(B, A)**, (S, A), (C, B), (S, C)

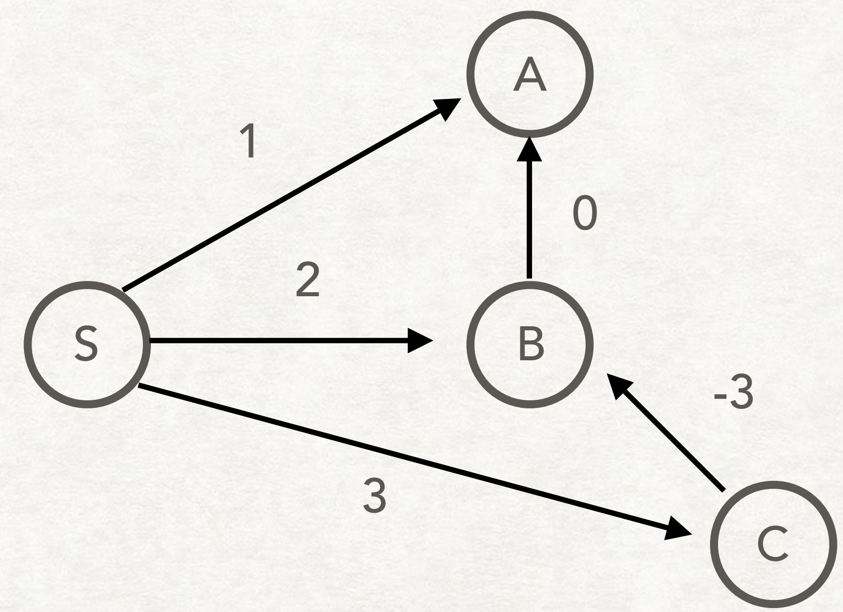


	0	1	2	3
S	0, S	0, S	0, S	
A	∞	1, S	1, S	
B	∞	2, S	2, S	
C	∞	3, S	3, S	

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order
(S, B), (B, A), **(S, A)**, (C, B), (S, C)



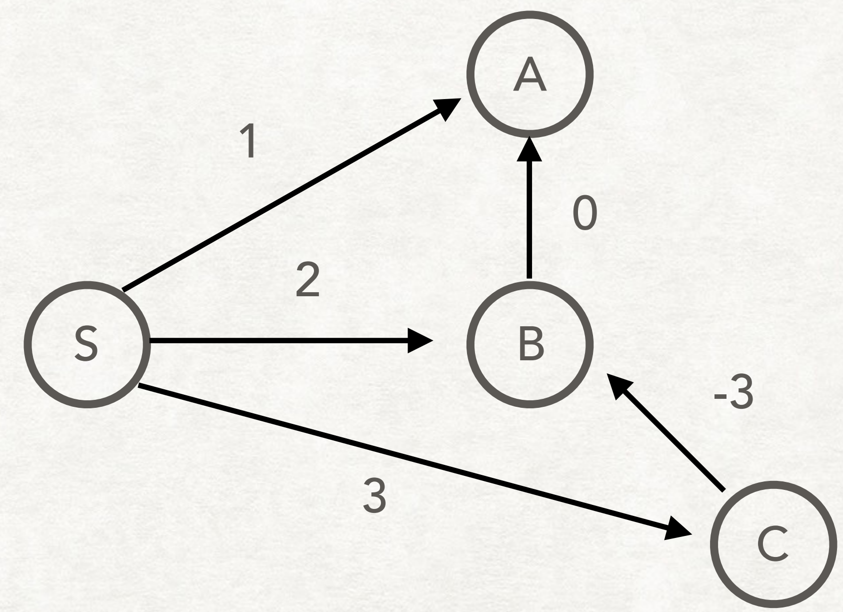
	0	1	2	3
S	0, S	0, S	0, S	
A	∞	1, S	1, S	
B	∞	2, S	2, S	
C	∞	3, S	3, S	

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order

(S, B), (B, A), (S, A), **(C, B)**, (S, C)



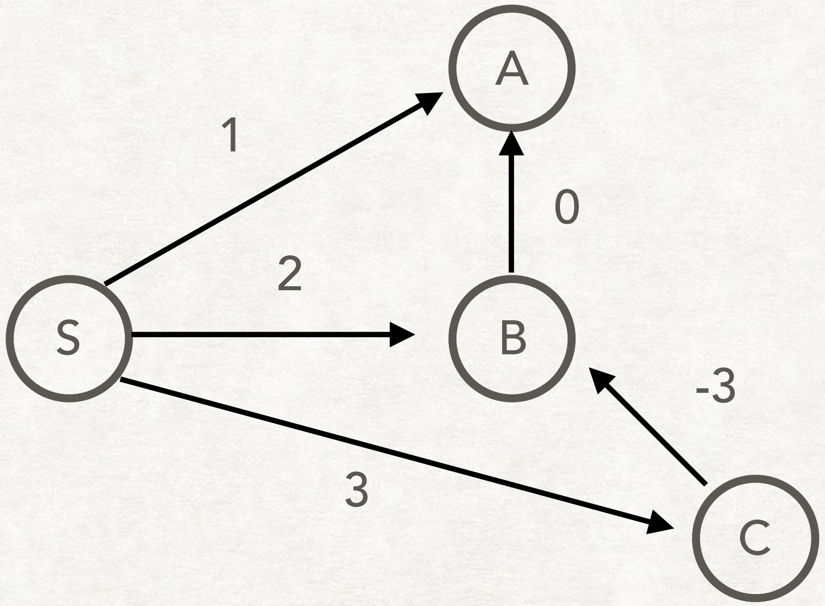
	0	1	2	3
S	0, S	0, S	0, S	
A	∞	1, S	1, S	
B	∞	2, S	**0, C**	
C	∞	3, S	3, S	

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order

(S, B), (B, A), (S, A), (C, B), **(S, C)**



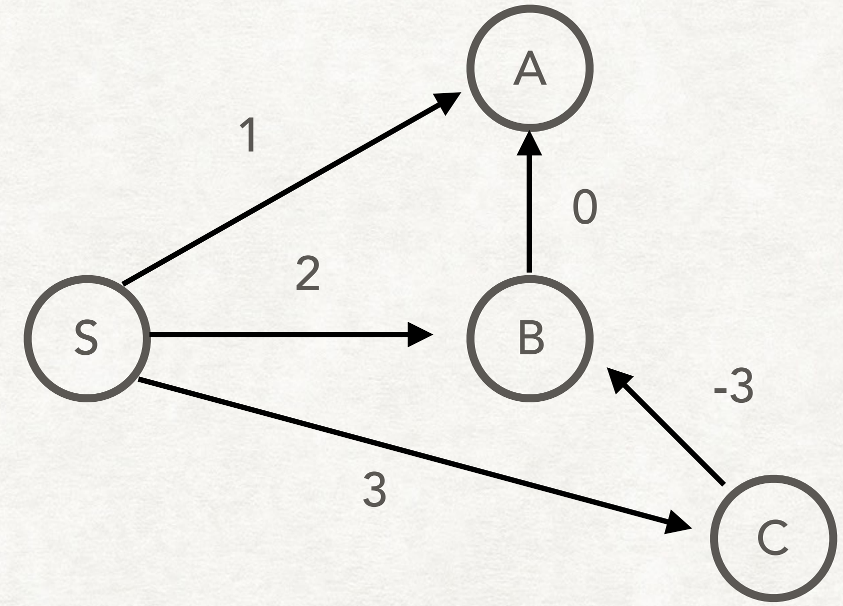
	0	1	2	3
S	0, S	0, S	0, S	
A	∞	1, S	1, S	
B	∞	2, S	0, C	
C	∞	3, S	3, S	

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order

(S, B), (B, A), (S, A), (C, B), (S, C)



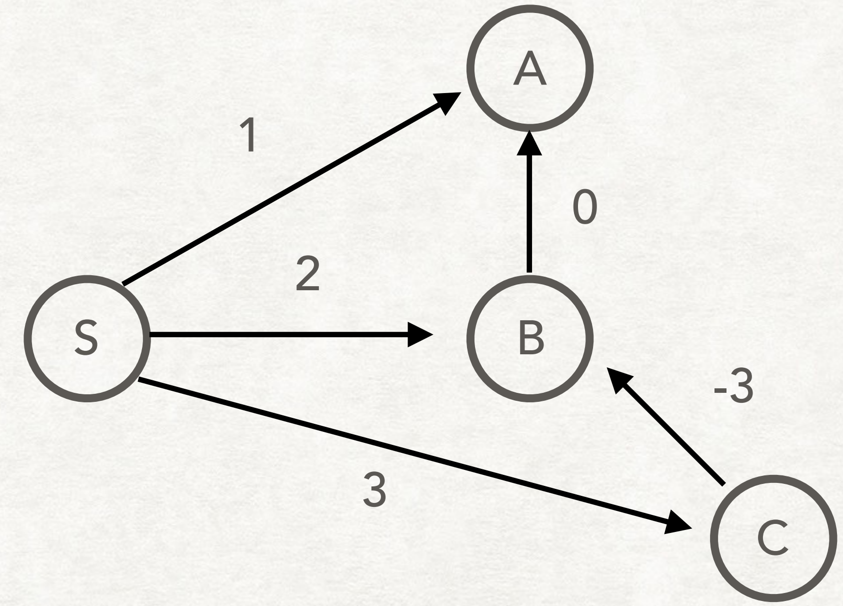
	0	1	2	3
S	0, S	0, S	0, S	0, S
A	∞	1, S	1, S	1, S
B	∞	2, S	0, C	0, C
C	∞	3, S	3, S	3, S

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order

(S, B), **(B, A)**, (S, A), (C, B), (S, C)

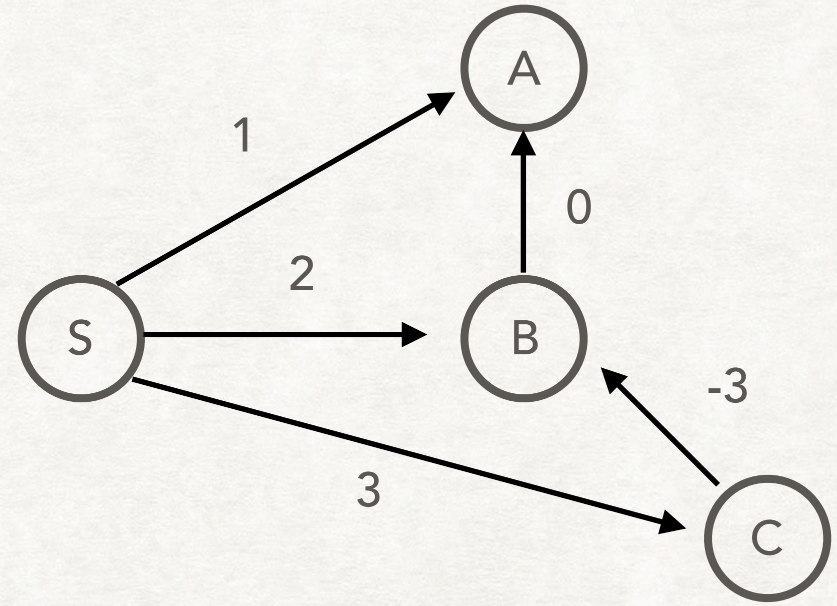


	0	1	2	3
S	0, S	0, S	0, S	0, S
A	∞	1, S	1, S	0, B
B	∞	2, S	0, C	0, C
C	∞	3, S	3, S	3, S

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order
(S, B), (B, A), **(S, A)**, (C, B), (S, C)

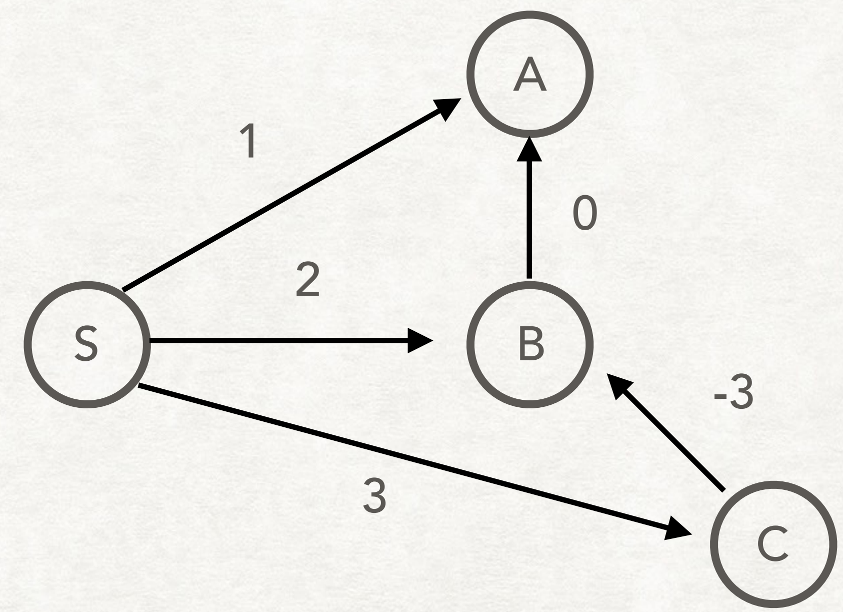


	0	1	2	3
S	0, S	0, S	0, S	0, S
A	∞	1, S	1, S	0, B
B	∞	2, S	0, C	0, C
C	∞	3, S	3, S	3, S

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order
(S, B), (B, A), (S, A), **(C, B)**, (S, C)



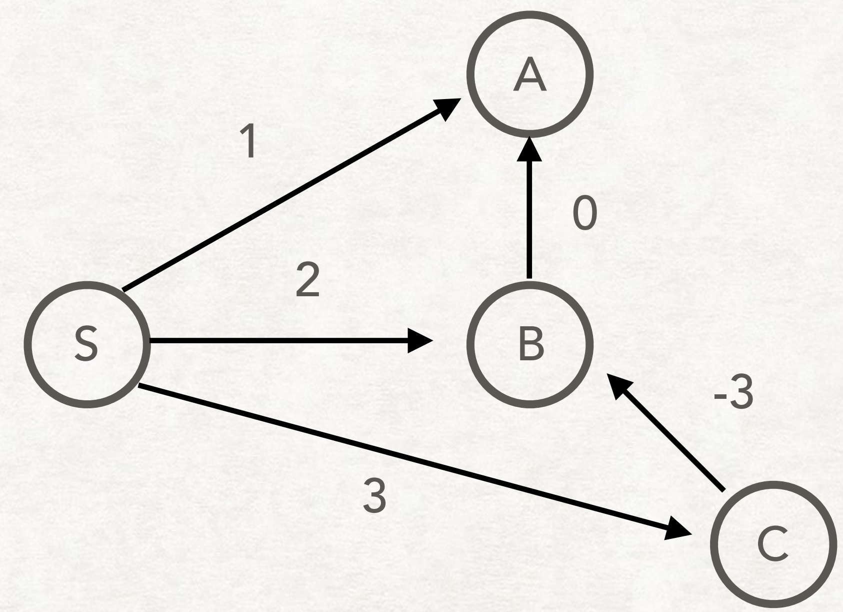
	0	1	2	3
S	0, S	0, S	0, S	0, S
A	∞	1, S	1, S	0, B
B	∞	2, S	0, C	0, C
C	∞	3, S	3, S	3, S

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order

(S, B), (B, A), (S, A), (C, B), **(S, C)**



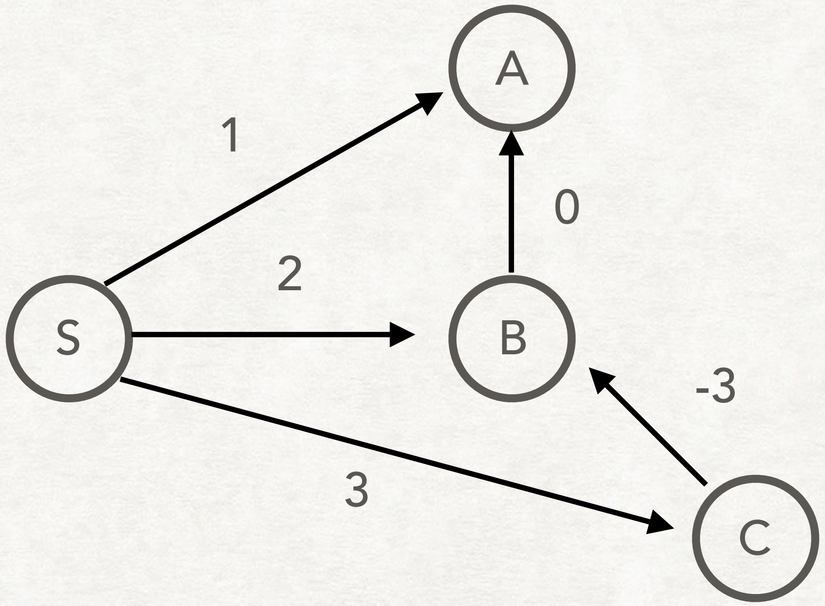
	0	1	2	3
S	0, S	0, S	0, S	0, S
A	∞	1, S	1, S	0, B
B	∞	2, S	0, C	0, C
C	∞	3, S	3, S	3, S

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

Iteration Order

(S, B), (B, A), (S, A), (C, B), (S, C)

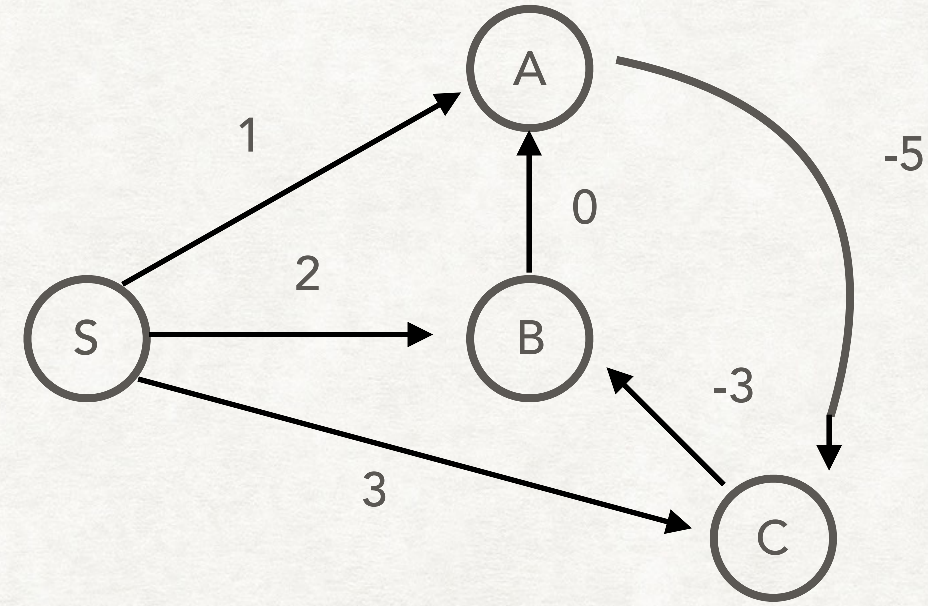


	0	1	2	3
S	0, S	0, S	0, S	0, S
A	∞	1, S	1, S	0, B
B	∞	2, S	0, C	0, C
C	∞	3, S	3, S	3, S

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

BELLMAN-FORD ALGORITHM

- Negative Cycle A - C - B - A
- Perform 1 more iteration to see if any nodes have a shorter distance



	0	1	2	3	4
S	0, S	0, S	0, S	0, S	0, S
A	∞	1, S	1, S	0, B	0, B
B	∞	2, S	0, C	0, C	**-8, C**
C	∞	3, S	*-4, A*	*-5, A*	-5, A

```
bellman_ford (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  do |V| - 1 iterations:
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
```

Iteration Order
(S, B), (B, A), (S, A), (C, B), (S, C), (A, C)

SHORTEST PATH IN DAG

- In any path of a DAG, vertices appear in increasing linearized order.
- Topological sort, and then visit vertices in sorted order.
- Update distance to neighbor.

```
update((u, v)):
  dist(v) = min(dist(v), dist(u) + w(u, l)

dag_shortest_path (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  Linearze G
  do vertex u in linearized order:
    for edge (u, v):
      update((u, v))
```

$O(|V| + |E|)$

SHORTEST PATH IN DAG

- When you reach a vertex *v*, you have already found shortest path to it.
- Visited all vertices that has an edge to *v*.

```
update((u, v)):
 dist(v) = min(dist(v), dist(u) + w(u, l)

dag_shortest_path (G, s):
 d[v] = infinity
 d[s] = 0
 prev[s] = s
 Linearze G
 do vertex u in linearized order:
  for edge (u, v):
   update((u, v))
```

O(|V| + |E|)

SHORTEST PATH IN DAG

- Find longest path by negating all edge lengths.
- Negative edge weights work.
- No need to propagate them forward.
- Visit each vertex in turn.

```
update((u, v)):
  dist(v) = min(dist(v), dist(u) + w(u, l)

dag_shortest_path (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  Linearze G
  do vertex u in linearized order:
    for edge (u, v):
      update((u, v))
```

$O(|V| + |E|)$

MINIMUM SPANNING TREE

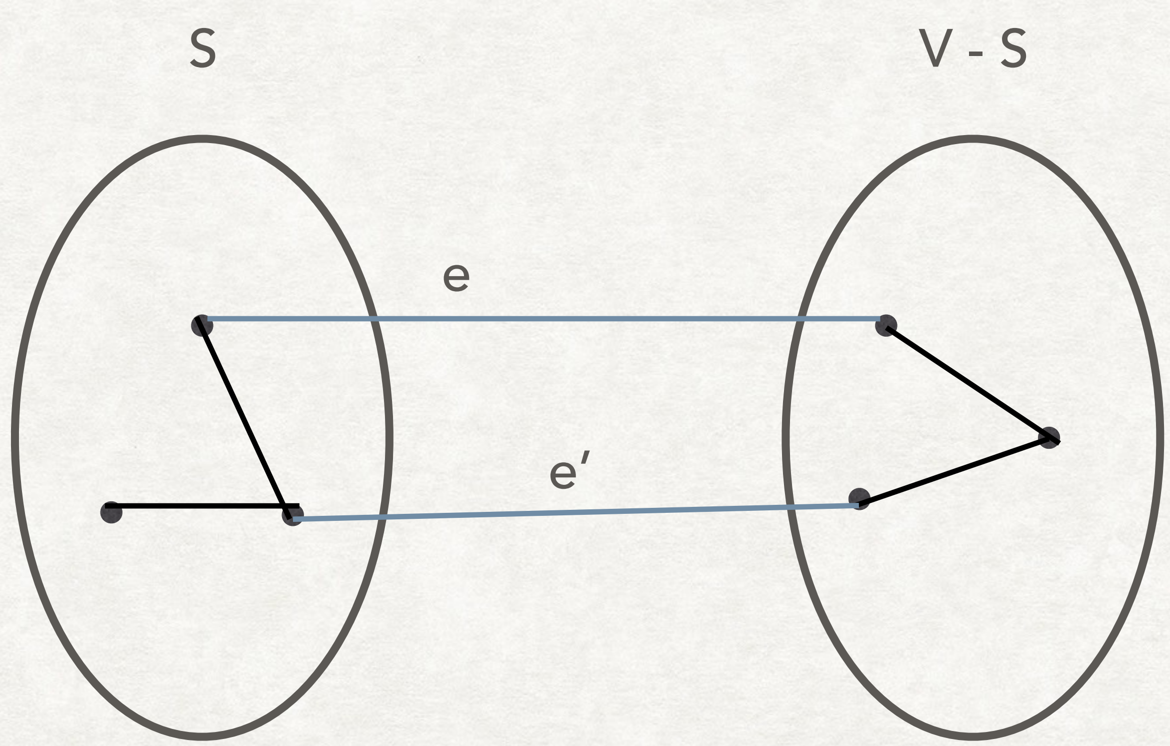
- Spanning Tree is some tree of the graph.
- All vertices connected if graph is connected.
- Minimum Spanning Tree takes edges with lowest total cost.

MINIMUM SPANNING TREE

- Cut Property
  - Set of edges whose removal disconnects the graph.
  - For any partition of vertices V, (S, V - S), set of edges that crosses the two partitions.
- Any edge of minimal weight in a cut is in some MST.
- If it is unique, it must be in the MST.

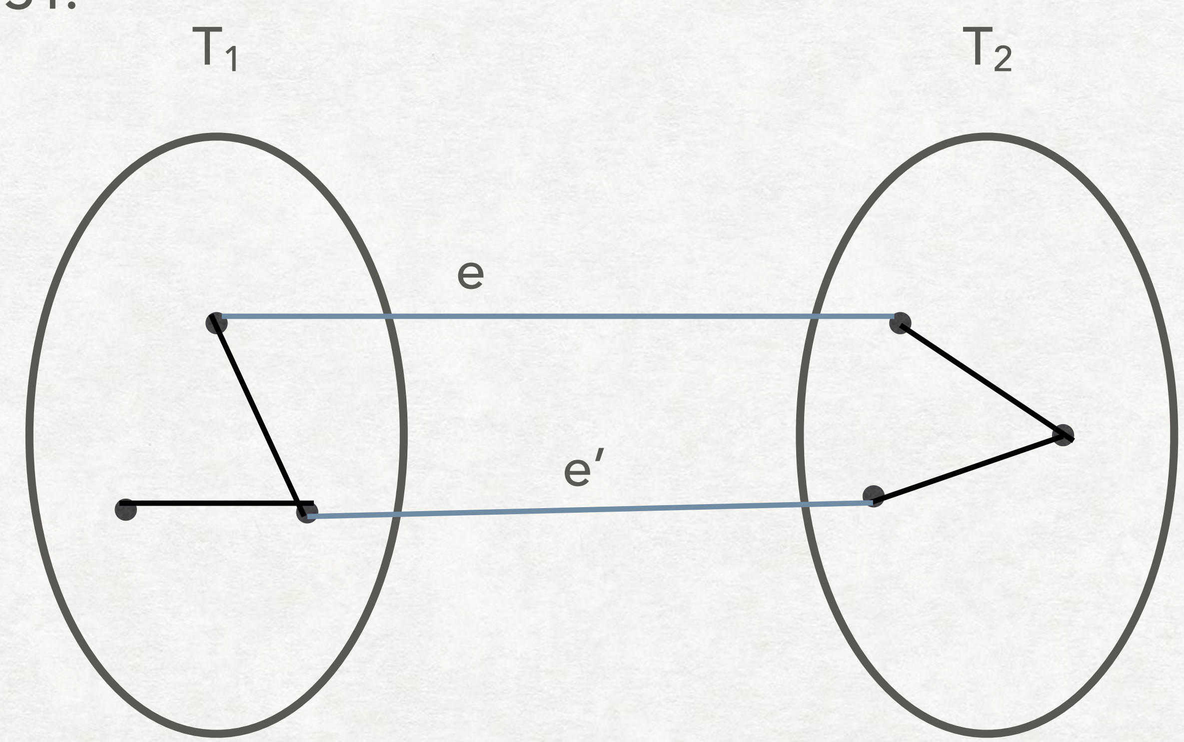
MINIMUM SPANNING TREE

- Cut Property
  - Set of edges whose removal disconnects the graph.
  - For any partition of vertices V, (S, V - S), set of edges that crosses the two partitions.



MINIMUM SPANNING TREE

- Suppose we have two MSTs T_1 and T_2.
- Edge *e* crosses the $\{T_1, T_2\}$ cut. Adding edge *e′* creates a cycle.
- Removing the largest edge in this cycle keeps T_1 and T_2 connected, and creates a spanning tree of T_1 and T_2.
- Since T_1 and T_2 were both MSTs, and removing largest edge creates another spanning tree, $\{T_1, T_2\}$ is a MST.



KRUSKAL'S ALGORITHM

- Choose lightest edge that does not form a cycle.
- O(|E| log |E| + |E| log |V|) = O(|E| log |E|) = O(|E| log |V|)
- Demo

```
kruskal(G):
  sort edges
  for each edge in sorted order:
    if no cycle:
      add edge
```

PRIM'S ALGORITHM

- Grow tree similar to Dijkstra's algorithm.
- Take lightest edge that connects existing tree to unseen vertex.
- O((|V| + |E|) log |V|) with binary heap
- O(|E| + |V| log |V|) with Fibonacci heap
- Demo

```
prim (G, s):
  c[v] = infinity
  c[s] = 0
  prev[s] = s
  PriorityQueue.add(G.V, c)
  while PQ not empty:
    u = PQ.DeleteMin()
    add (u, prev(u)) to T
    for edge (u, v):
      if w(u, v) < c(v):
        c(v) = w(u, v)
        prev(v) = u
        PQ.DecreaseKey(v, c[v])
```

```
generic_MST(G):
  start S = v:
    find lightest edge (x, y)
      in crossing (S, V - S)
    S = S union {y}
```

MST NEGATIVE WEIGHTS

- Both Kruskal's and Prim's work with negative weights.
- Smallest edge is defined the same for positive or negative weights.
- Kruskal's candidate edge is least weight edge that connects two distinct components.
- Prim's candidate edge is least weight edge connecting to seen set to an unseen vertex.