Shortest Paths and Minimum Spanning Trees

David Kauchak
cs302
Spring 2013

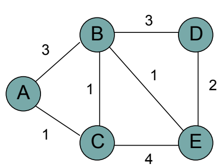
Admin

Can resubmit homeworks 12-15 for up to half credit back

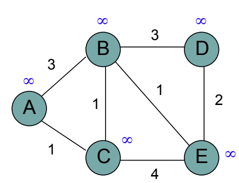
- Due by the end of the week

Read book

```
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
```



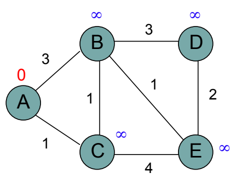
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                      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



```
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

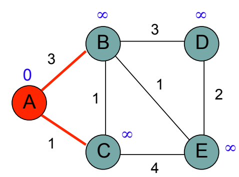
A 0
B ∞
C ∞
D ∞
E ∞




```
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

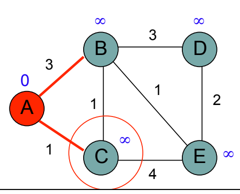
B ∞
C ∞
D ∞
E ∞




```
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

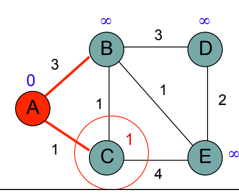
B ∞
C ∞
D ∞
E ∞

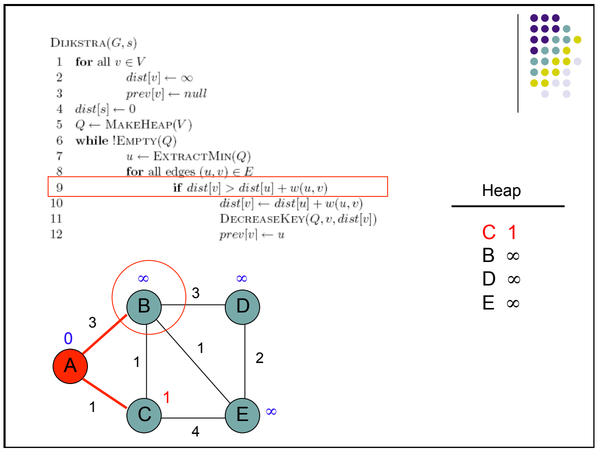


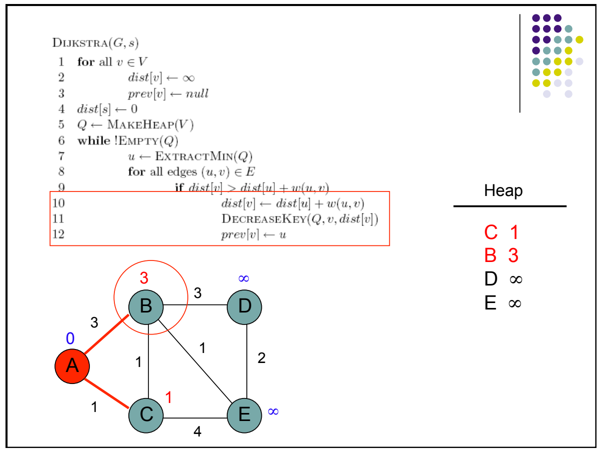

```
DIJKSTRA(G, s)
for all v ∈ V
    dist[v] ← ∞
    prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
    u ← EXTRACTMIN(Q)
    for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
            dist[v] ← dist[u] + w(u, v)
            DECREASEKEY(Q, v, dist[v])
            prev[v]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

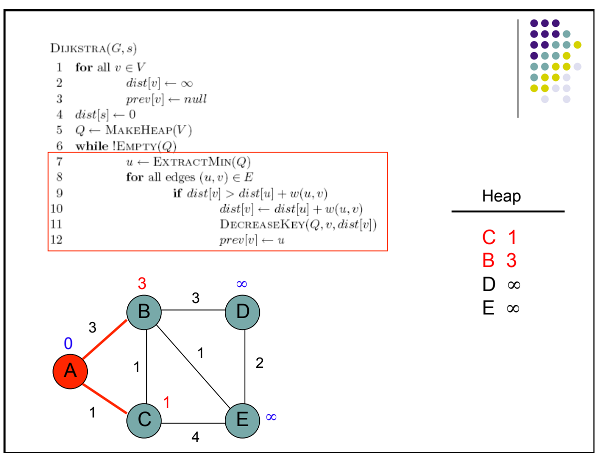
Heap

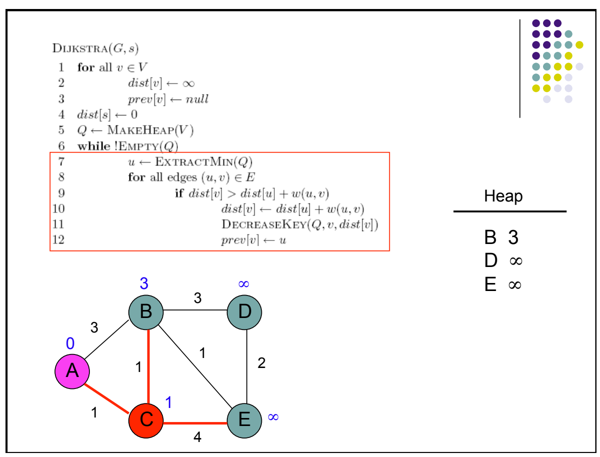
C 1
B ∞
D ∞
E ∞

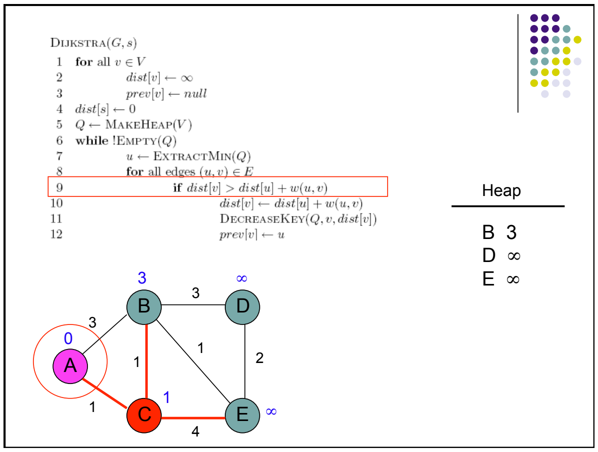


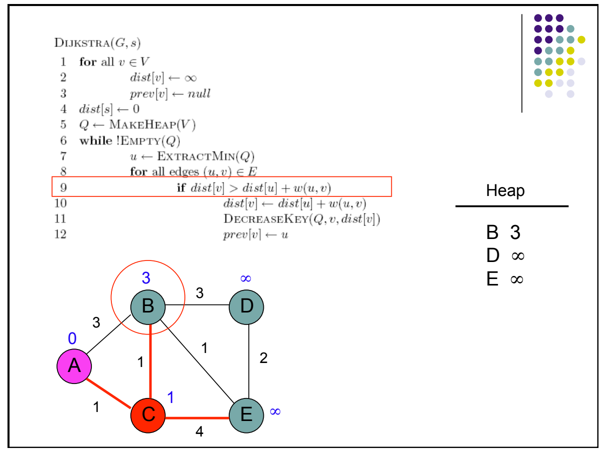

DIJKSTRA(G, s)
for all v ∈ V
dist[v] ← ∞
prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
u ← EXTRACTMIN(Q)
for all edges (u, v) ∈ E
if dist[v] > dist[u] + w(u, v)
dist[v] ← dist[u] + w(u, v)
DECREASEKEY(Q, v, dist[v])
prev[v] ← u
Heap
C 1
B ∞
D ∞
E ∞

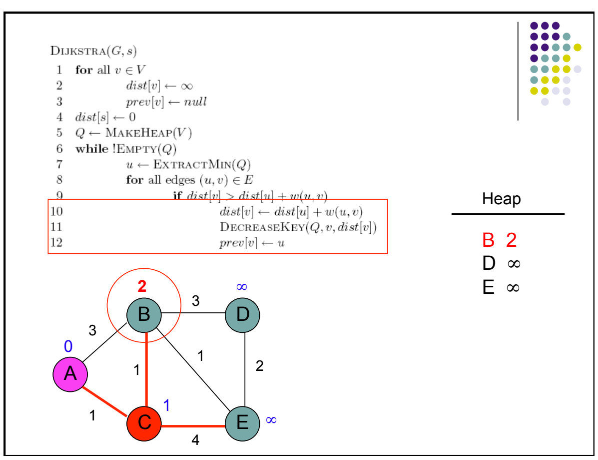

DIJKSTRA(G, s)
for all v ∈ V
dist[v] ← ∞
prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
u ← EXTRACTMIN(Q)
for all edges (u, v) ∈ E
if dist[v] > dist[u] + w(u, v)
dist[v] ← dist[u] + w(u, v)
DECREASEKEY(Q, v, dist[v])
prev[v] ← u
Heap
C 1
B 3
D ∞
E ∞

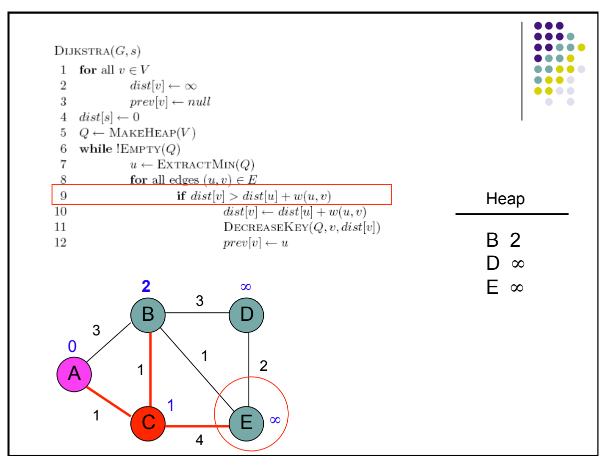

DIJKSTRA(G, s)
for all v ∈ V
dist[v] ← ∞
prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
u ← EXTRACTMIN(Q)
for all edges (u, v) ∈ E
if dist[v] > dist[u] + w(u, v)
dist[v] ← dist[u] + w(u, v)
DECREASEKEY(Q, v, dist[v])
prev[v] ← u
Heap
C 1
B 3
D ∞
E ∞


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)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← null
4 dist[s] ← 0
5 Q ← MAKEHEAP(V)
6 while !EMPTY(Q)
7 u ← EXTRACTMIN(Q)
8 for all edges (u, v) ∈ E
9 if dist[v] > dist[u] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DECREASEKEY(Q, v, dist[v])
12 prev[v] ← u
Heap
B 3
D ∞
E ∞


DIJKSTRA(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← null
4 dist[s] ← 0
5 Q ← MAKEHEAP(V)
6 while !EMPTY(Q)
7 u ← EXTRACTMIN(Q)
8 for all edges (u, v) ∈ E
9 if dist[v] > dist[u] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DECREASEKEY(Q, v, dist[v])
12 prev[v] ← u
Heap
B 3
D ∞
E ∞

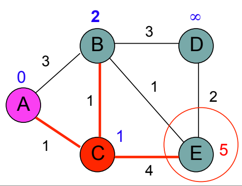

DIJKSTRA(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← null
4 dist[s] ← 0
5 Q ← MAKEHEAP(V)
6 while !EMPTY(Q)
7 u ← EXTRACTMIN(Q)
8 for all edges (u, v) ∈ E
9 if dist[v] > dist[u] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DECREASEKEY(Q, v, dist[v])
12 prev[v] ← u
Heap
B 2
D ∞
E ∞


DIJKSTRA(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← null
4 dist[s] ← 0
5 Q ← MAKEHEAP(V)
6 while !EMPTY(Q)
7 u ← EXTRACTMIN(Q)
8 for all edges (u, v) ∈ E
9 if dist[v] > dist[u] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DECREASEKEY(Q, v, dist[v])
12 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

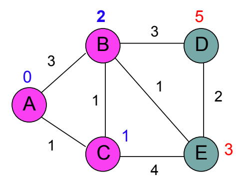
B 2
E 5
D ∞




```
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

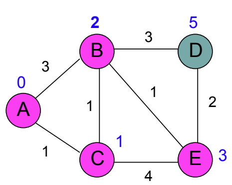
E 3
D 5




```
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
```

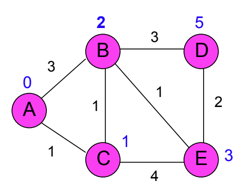
Heap

D 5



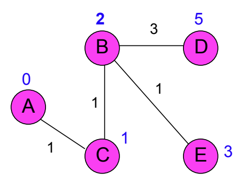

```
DIJKSTRA(G, s)
  for all v ∈ V
      dist[v] ← ∞
      prev[v] ← null
  dist[s] ← 0
  Q ← MAKEHEAP(V)
  while !EMPTY(Q)
      u ← EXTRACTMIN(Q)
      for all edges (u, v) ∈ E
          if dist[v] > dist[u] + w(u, v)
              dist[v] ← dist[u] + w(u, v)
              DECREASEKEY(Q, v, dist[v])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

proof?

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)
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
```

1 call to MakeHeap

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
```

|V| iterations

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
```

|V| calls

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
```

O(|E|) calls

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	|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|)

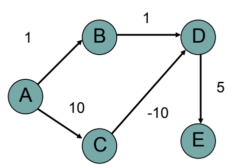
Is this an improvement? If |E| < |V|^2 / log |V|

Running time?

Depends on the heap implementation

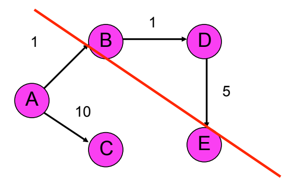
	1 MakeHeap	|V| ExtractMin	|E| DecreaseKey	Total
Array	O(|V|)	O(|V|^2)	O(|E|)	O(|V|^2)
Bin heap	O(|V|)	O(|V| log |V|)	O(|E| log |V|)	O((|V|+|E|) log |V|) O(|E| log |V|)
Fib heap	O(|V|)	O(|V| log |V|)	O(|E|)	O(|V| log |V| + |E|)

What about Dijkstra's on…?



What about Dijkstra's on…?

Dijkstra's algorithm only works for positive edge weights



Bounding the distance

Another invariant: For each vertex v, dist[v] is an upper bound on the actual shortest 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
```

Is this a valid invariant?

Bounding the distance

Another invariant: For each vertex v, dist[v] is an upper bound on the actual shortest distance

- start off at ∞
- only update the value if we find a shorter distance

An update procedure

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

Can we ever go wrong applying this update rule?

- We can apply this rule as many times as we want and will never underestimate dist[v]

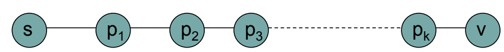
When will dist[v] be right?

- If u is along the shortest path to v and dist[u] is correct

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

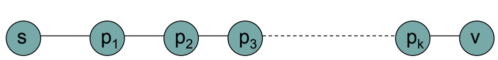
Consider the shortest path from s to v



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

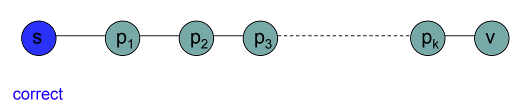
What happens if we update all of the vertices with the above update?



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

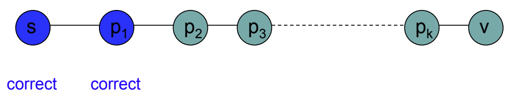
What happens if we update all of the vertices with the above update?



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

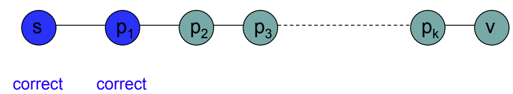
What happens if we update all of the vertices with the above update?



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

Does the order that we update the vertices matter?

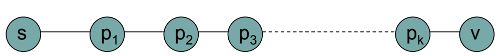


$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

How many times do we have to do this for vertex p_i to have the correct shortest path from s?

- i times

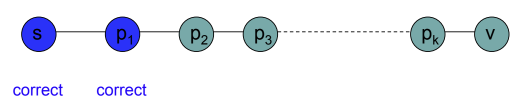


$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

How many times do we have to do this for vertex p_i to have the correct shortest path from s?

- i times

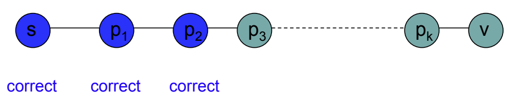


$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

How many times do we have to do this for vertex p_i to have the correct shortest path from s?

- i times

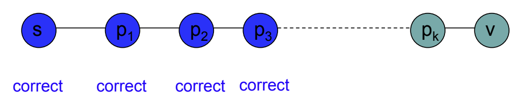


$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

How many times do we have to do this for vertex p_i to have the correct shortest path from s?

- i times

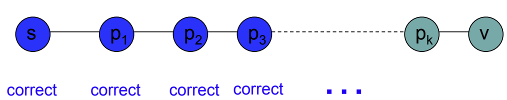


$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

How many times do we have to do this for vertex p_i to have the correct shortest path from s?

- i times

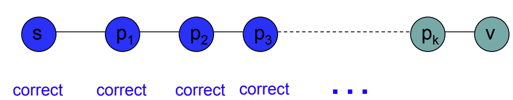


$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

What is the longest (vertex-wise) the path from s to any node v can be?

- |V| - 1 edges/vertices



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

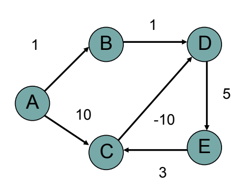
Bellman-Ford algorithm

```
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

Negative cycles

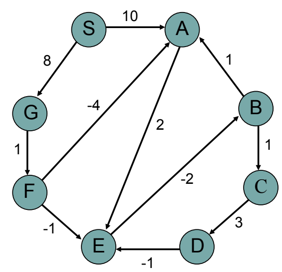
What is the shortest path from a to e?



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
```

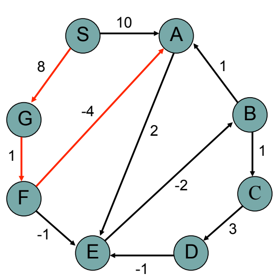
Bellman-Ford algorithm



How many edges is the shortest path from s to:

A:

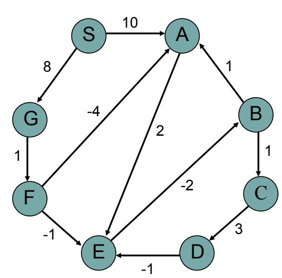
Bellman-Ford algorithm



How many edges is the shortest path from s to:

A: 3

Bellman-Ford algorithm

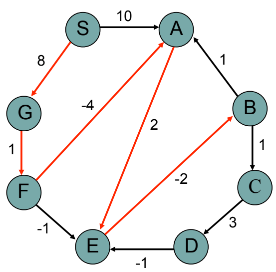


How many edges is the shortest path from s to:

A: 3

B:

Bellman-Ford algorithm

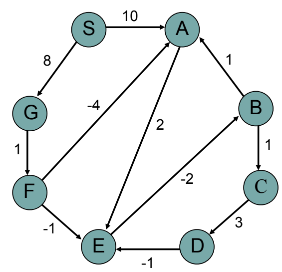


How many edges is the shortest path from s to:

A: 3

B: 5

Bellman-Ford algorithm



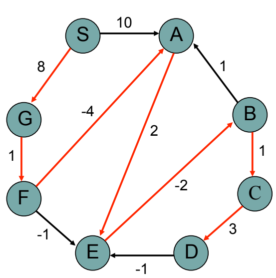
How many edges is the shortest path from s to:

A: 3

B: 5

D:

Bellman-Ford algorithm



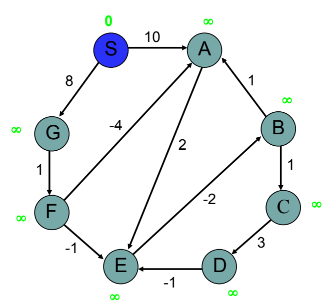
How many edges is the shortest path from s to:

A: 3

B: 5

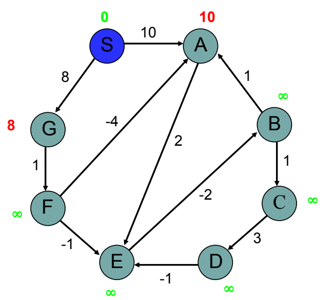
D: 7

Bellman-Ford algorithm



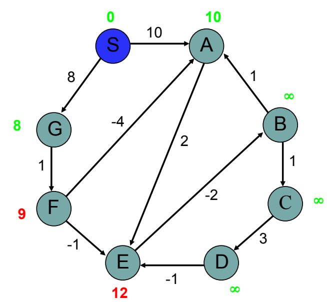
Iteration: 0

Bellman-Ford algorithm



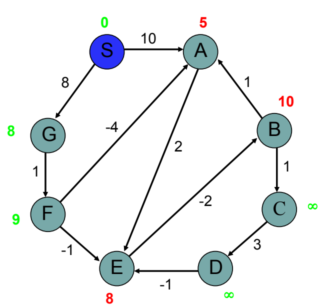
Iteration: 1

Bellman-Ford algorithm



Iteration: 2

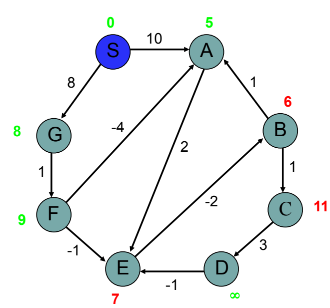
Bellman-Ford algorithm



Iteration: 3

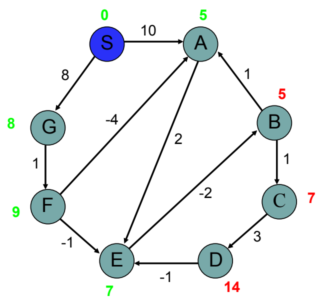
A has the correct distance and path

Bellman-Ford algorithm



Iteration: 4

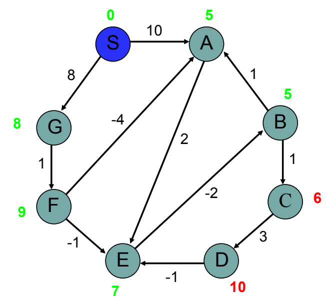
Bellman-Ford algorithm



Iteration: 5

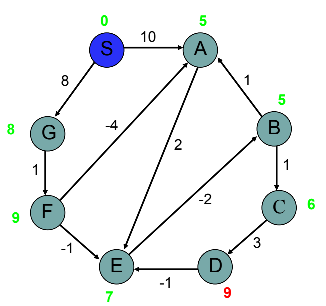
B has the correct distance and path

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

```
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
```

Can you modify the algorithm to run faster (in some circumstances)?

Single source shortest paths

All of the shortest path algorithms we've looked at today are call "single source shortest paths" algorithms

Why?

All pairs shortest paths

Simple approach

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

Floyd-Warshall – $\Theta(|V|^3)$

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

Minimum spanning trees

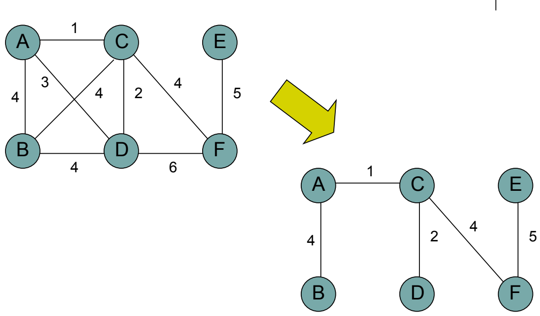
What is the lowest weight set of edges that connects all vertices of an undirected graph with positive weights

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

Output: A tree T=(V,E') where E' ⊆ E that minimizes

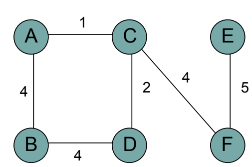
$$weight(T) = \sum_{e \in E'} w_e$$

MST example



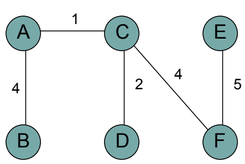
MSTs

Can an MST have a cycle?



MSTs

Can an MST have a cycle?



Applications?

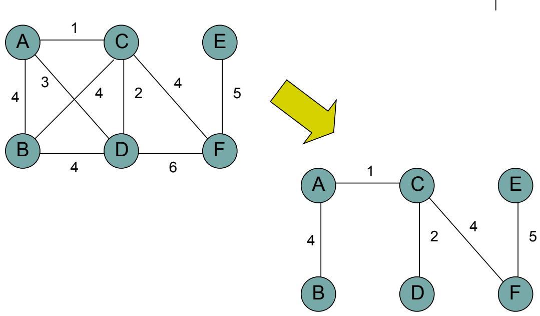
Connectivity

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

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

Traveling salesman problem?

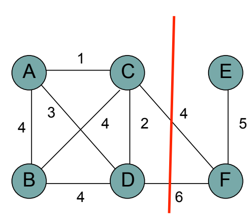
Algorithm ideas?



Cuts

A cut is a partitioning of the vertices into two sets S and V-S

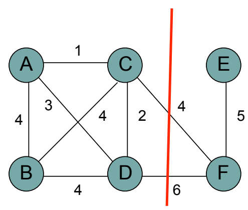
An edge "crosses" the cut if it connects a vertex u∈V and v∈V-S



Minimum cut property

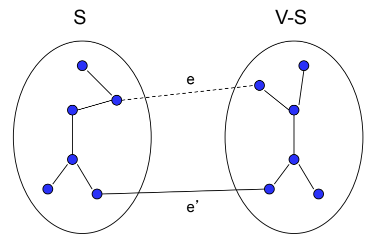
Given a partion S, let edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge *e*.

Prove this!



Minimum cut property

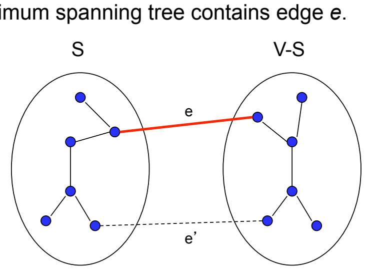
Given a partion S, let edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge *e*.



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

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*.



Using e instead of e', still connects the graph, but produces a tree with smaller weights

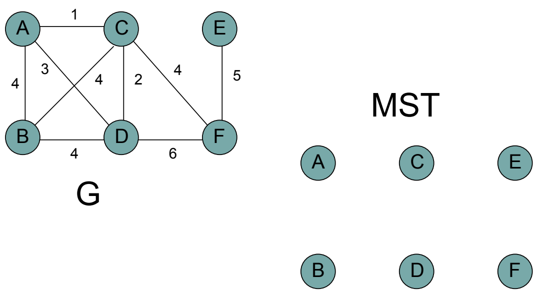
Kruskal's algorithm

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

```
KRUSKAL(G)
for all v ∈ V
        MAKESET(v)
T ← {}
sort the edges of E by weight
for all edges (u, v) ∈ E in increasing order of weight
        if FIND-SET(u) ≠ FIND-SET(v)
                add edge to T
                UNION(FIND-SET(u),FIND-SET(v))
```

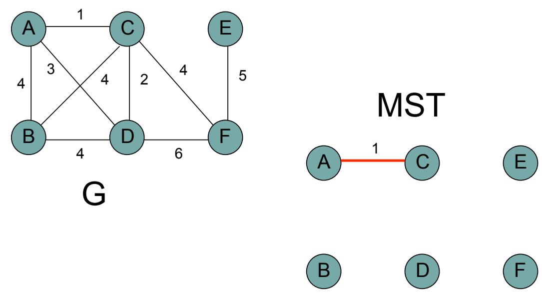
Kruskal's algorithm

Add smallest edge that connects
two sets not already connected



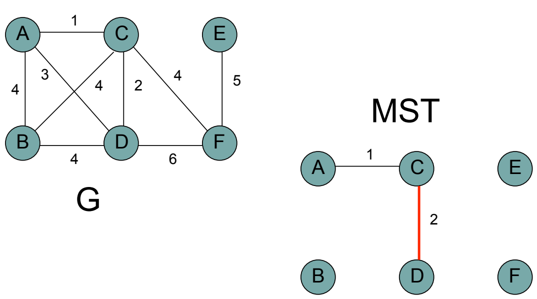
Kruskal's algorithm

Add smallest edge that connects
two sets not already connected



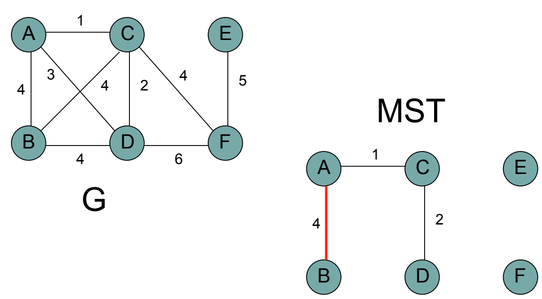
Kruskal's algorithm

Add smallest edge that connects
two sets not already connected



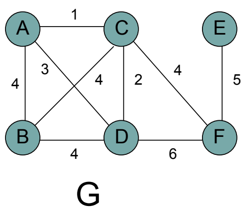
Kruskal's algorithm

Add smallest edge that connects
two sets not already connected

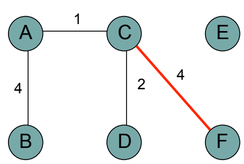


Kruskal's algorithm

Add smallest edge that connects two sets not already connected

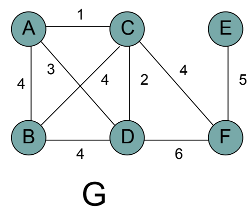


MST

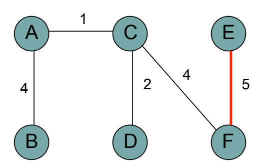


Kruskal's algorithm

Add smallest edge that connects two sets not already connected



MST



Correctness of Kruskal's

Never adds an edge that connects already connected vertices

Always adds lowest cost edge to connect two sets. By min cut property, that edge must be part of the MST

```
KRUSKAL(G)
for all v ∈ V
        MAKESET(v)
T ← {}
sort the edges of E by weight
for all edges (u, v) ∈ E in increasing order of weight
        if FIND-SET(u) ≠ FIND-SET(v)
                add edge to T
                UNION(FIND-SET(u),FIND-SET(v))
```

Running time of Kruskal's

```
KRUSKAL(G)
for all v ∈ V
        MAKESET(v)
T ← {}
sort the edges of E by weight
for all edges (u, v) ∈ E in increasing order of weight
        if FIND-SET(u) ≠ FIND-SET(v)
                add edge to T
                UNION(FIND-SET(u),FIND-SET(v))
```

Running time of Kruskal's

```
KRUSKAL(G)
  for all v ∈ V
      MAKESET(v)
  T ← {}
  sort the edges of E by weight
  for all edges (u, v) ∈ E in increasing order of weight
      if FIND-SET(u) ≠ FIND-SET(v)
          add edge to T
          UNION(FIND-SET(u),FIND-SET(v))
```

|V| calls to MakeSet

O(|E| log |E|)

2 |E| calls to FindSet

|V| calls to Union

Running time of Kruskal's

Disjoint set data structure

O(|E| log |E|) +

	MakeSet	FindSet |E| calls	Union |V| calls	Total
Linked lists	|V|	O(|V| |E|)	|V|	O(|V||E| + |E| log |E|) O(|V| |E|)
Linked lists + heuristics	|V|	O(|E| log |V|)	|V|	O(|E| log |V|+ |E| log |E|) O(|E| log |E|)

Prim'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
```

Prim'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
```

Prim'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
```

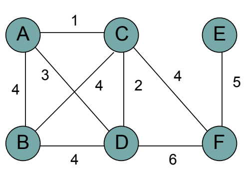
Prim's algorithm

Start at some root node and build out the MST by adding the lowest weighted edge at the frontier

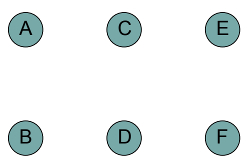
```
PRIM(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MAKEHEAP(key)
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))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

Prim's

```
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
```

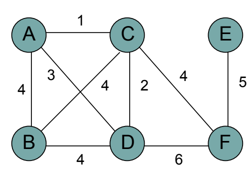


MST

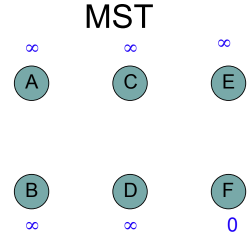


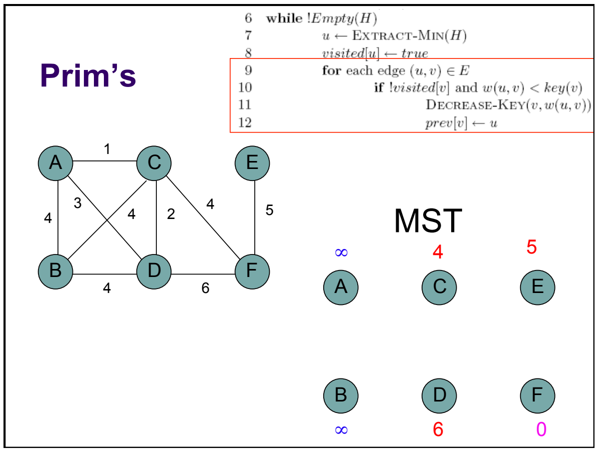
Prim's

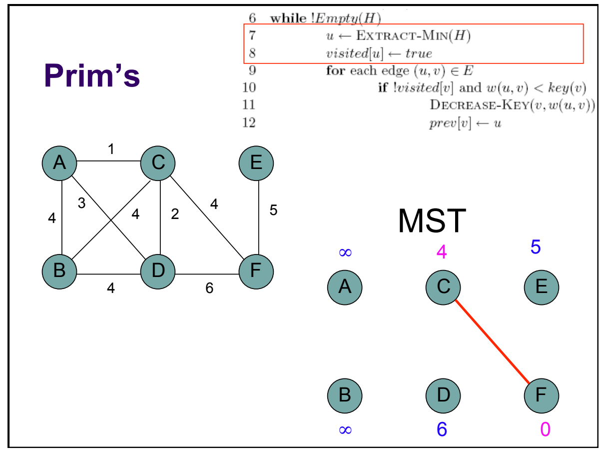
```
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

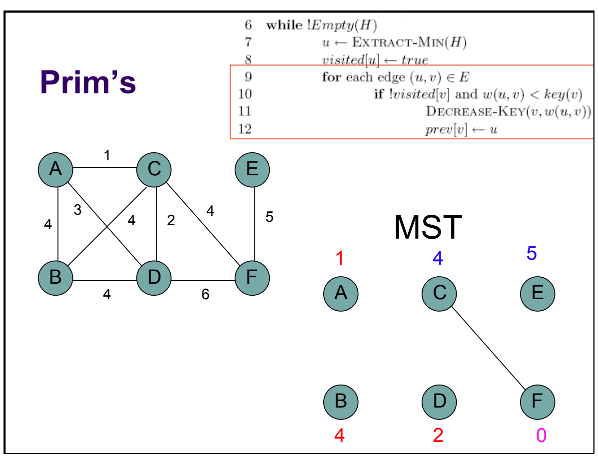


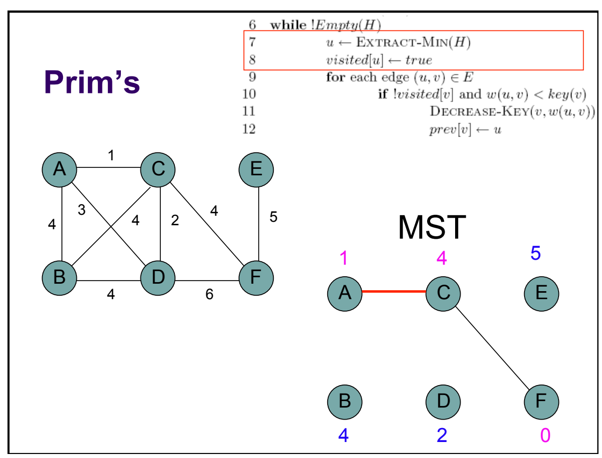
MST

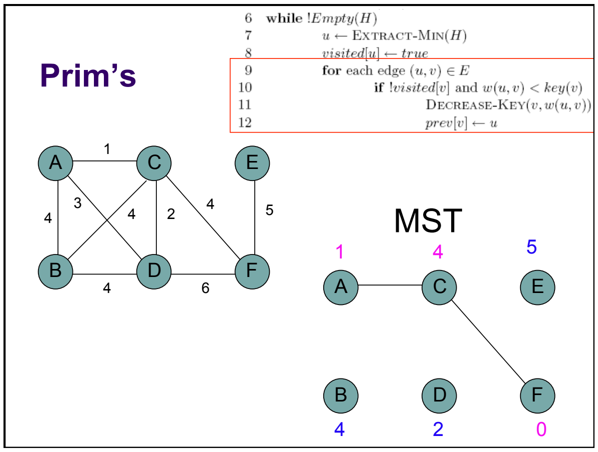


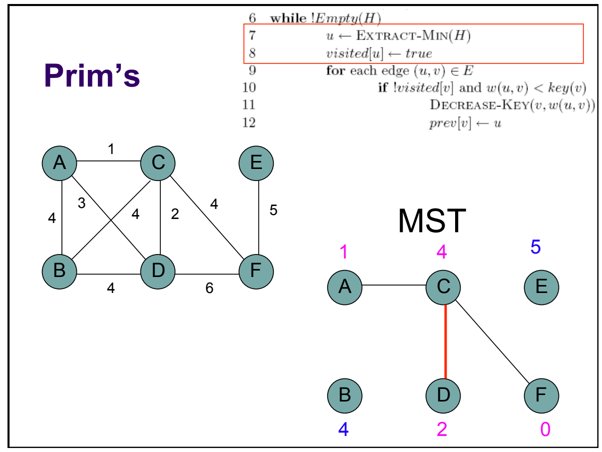

Prim's
while !Empty(H)
u ← Extract-Min(H)
visited[u] ← true
for each edge (u, v) ∈ E
if !visited[v] and w(u, v) < key(v)
Decrease-Key(v, w(u, v))
prev[v] ← u
A B C D E F
1 3 4 4 2 4 5 4 6
MST
∞ 4 5
A C E
B D F
∞ 6 0

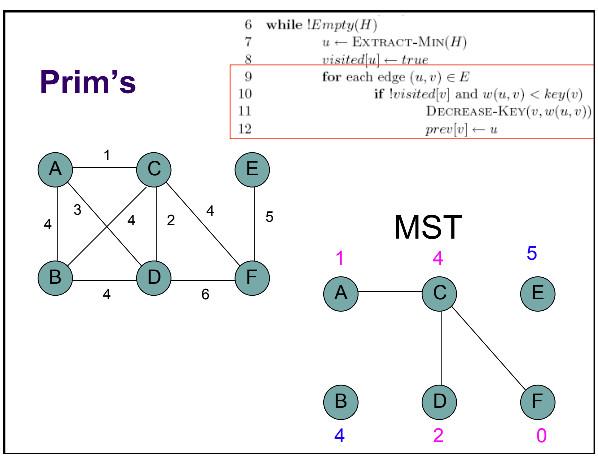

Prim's
while !Empty(H)
u ← Extract-Min(H)
visited[u] ← true
for each edge (u, v) ∈ E
if !visited[v] and w(u, v) < key(v)
Decrease-Key(v, w(u, v))
prev[v] ← u
A B C D E F
1 3 4 4 2 4 5 4 6
MST
∞ 4 5
A C E
B D F
∞ 6 0

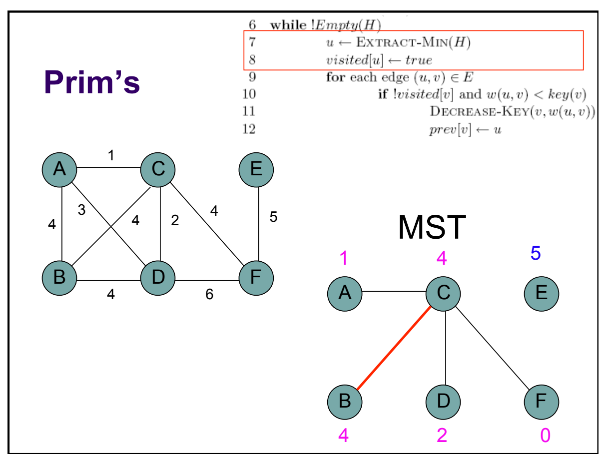

Prim's
while !Empty(H)
u ← Extract-Min(H)
visited[u] ← true
for each edge (u, v) ∈ E
if !visited[v] and w(u, v) < key(v)
Decrease-Key(v, w(u, v))
prev[v] ← u
A B C D E F
1 3 4 4 2 4 5 4 6
MST
1 4 5
A C E
B D F
4 2 0


Prim's
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
A B C D E F
1 3 4 4 2 4 5 4 6
MST
1 4 5
A C E
B D F
4 2 0


Prim's
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
A
C
E
B
D
F
1
3
4
4
2
4
5
4
6
MST
1
4
5
A
C
E
B
D
F
4
2
0

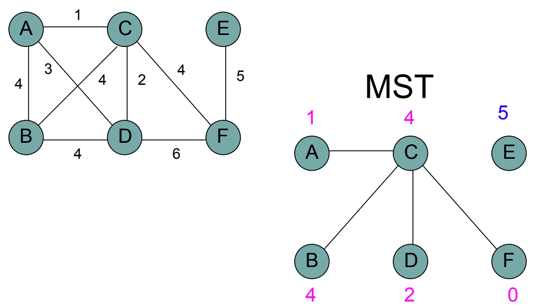

Prim's
while !Empty(H)
u ← Extract-Min(H)
visited[u] ← true
for each edge (u, v) ∈ E
if !visited[v] and w(u, v) < key(v)
Decrease-Key(v, w(u, v))
prev[v] ← u
A
C
E
B
D
F
1
3
4
4
2
4
5
4
6
MST
1
4
5
A
C
E
B
D
F
4
2
0


Prim's
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
A
C
E
B
D
F
1
3
4
4
2
4
5
4
6
MST
1
4
5
A
C
E
B
D
F
4
2
0


Prim's
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
A
C
E
B
D
F
1
3
4
4
2
4
5
4
6
MST
1
4
5
A
C
E
B
D
F
4
2
0

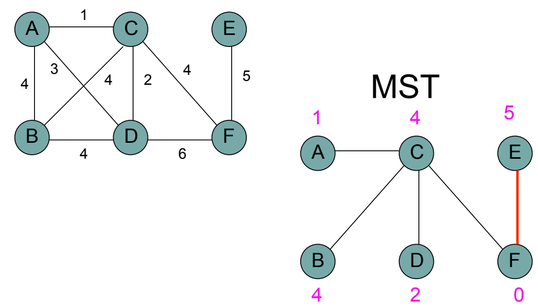
Prim's

while $!Empty(H)$
  $u \leftarrow$ EXTRACT-MIN(H)
  $visited[u] \leftarrow true$
  for each edge $(u, v) \in E$
    if $!visited[v]$ and $w(u, v) < key(v)$
      DECREASE-KEY($v, w(u, v)$)
      $prev[v] \leftarrow u$



Prim's

while $!Empty(H)$
  $u \leftarrow$ EXTRACT-MIN(H)
  $visited[u] \leftarrow true$
  for each edge $(u, v) \in E$
    if $!visited[v]$ and $w(u, v) < key(v)$
      DECREASE-KEY($v, w(u, v)$)
      $prev[v] \leftarrow u$



Correctness of Prim's?

Can we use the min-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*.

Let S be the set of vertices visited so far

The only time we add a new edge is if it's the lowest weight edge from S to V-S

Running time of Prim's

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

Running time of Prim's

```
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
```

Θ(|V|)

Θ(|V|)

|V| calls to Extract-Min

|E| calls to Decrease-Key

Running time of Prim's

Same as Dijksta's algorithm

	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|)
Fib heap	O(|V|)	O(|V| log |V|)	O(|E|)	**O(|V| log |V| + |E|)**

Kruskal's: O(|E| log |E|)