MORE GRAPH ALGORITHMS

David Kauchak
CS 140 – Spring 2023

1

Admin

Assignment 8 (DP coding). How did it go?

Assignment 9, graph algorithms: use/modify existing algorithms

2

Connectedness

Given an undirected graph, for every node $u \in V$, can we reach all other nodes in the graph? Algorithm + running time

Run BFS or DFS-Visit (one pass) and mark nodes as we visit them. If we visit all nodes, return true, otherwise false.

Running time: $O(|V| + |E|)$

3

Strongly connected

Given a directed graph, can we reach any node v from any other node u?

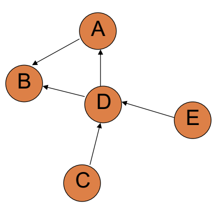
Can we do the same thing?

4

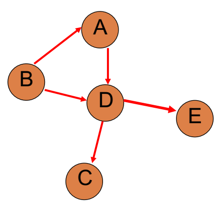
Transpose of a graph

Given a graph G, we can calculate the transpose of a graph G^R by reversing the direction of all the edges

G



G^R



Running time to calculate G^R? $\theta(|V| + |E|)$

5

Strongly connected

Strongly-Connected(G)
- Run DFS-Visit or BFS from some node u
- If not all nodes are visited: return false
- Create graph G^R
- Run DFS-Visit or BFS on G^R from node u
- If not all nodes are visited: return false
- return true

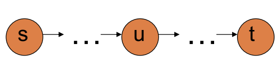
6

Is it correct?

What do we know after the first pass?
- Starting at u, we can reach every node

What do we know after the second pass?
- All nodes can reach u. Why?
- We can get from u to every node in G^R, therefore, if we reverse the edges (i.e. G), then we have a path from every node to u

Which means that any node can reach any other node. Given any two nodes s and t we can create a path through u



7

Runtime?

Strongly-Connected(G)	
- Run DFS-Visit or BFS from some node u	$O(\|V\| + \|E\|)$
- If not all nodes are visited: return false	$O(\|V\|)$
- Create graph G^R	$\theta(\|V\| + \|E\|)$
- Run DFS-Visit or BFS on G^R from node u	$O(\|V\| + \|E\|)$
- If not all nodes are visited: return false	$O(\|V\|)$
- return true	

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

8

Minimum spanning trees

What are they?

What do you remember about them?

What algorithms do you remember?

9

Minimum spanning trees

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

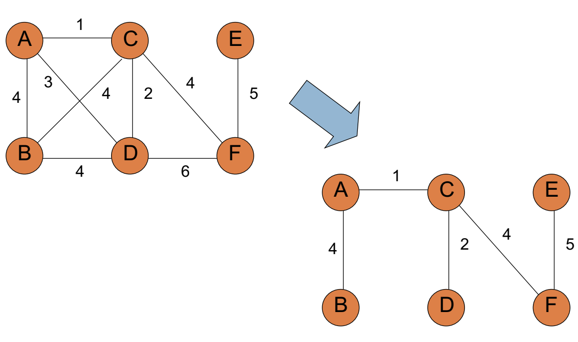
Input: An undirected, positive weight graph, G=(V,E)

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

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

10

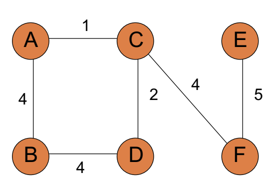
MST example



11

MSTs

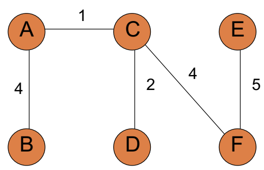
Can an MST have a cycle?



12

MSTs

Can an MST have a cycle?



13

Applications?

Connectivity

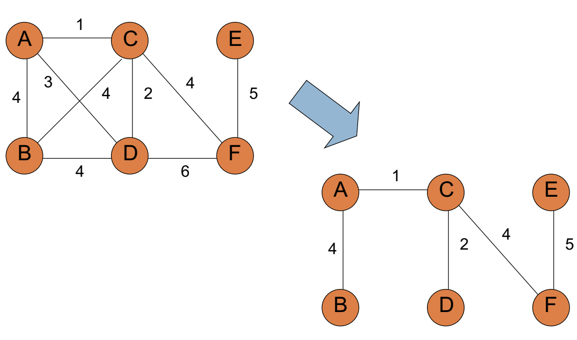
- Networks (e.g. communications)
- Circuit design/wiring

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

Traveling salesman problem?

14

Algorithm ideas?

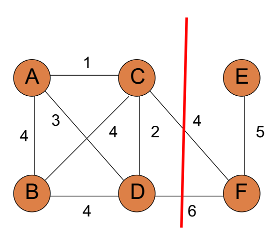


15

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

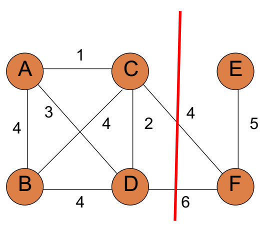


16

Minimum cut property

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

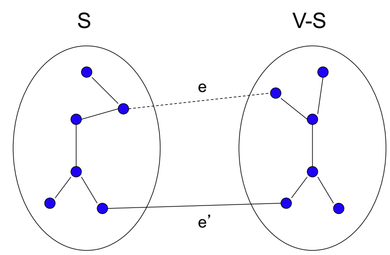
Prove this!



17

Minimum cut property

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

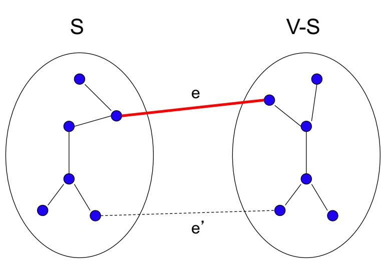


Consider an MST with edge e' that is not the minimum edge

18

Minimum cut property

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

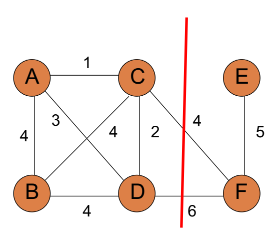


Using e instead of e', still connects the graph, but produces a tree with smaller weights

19

Minimum cut property

If the minimum cost edge that **crosses** the partition is not unique, then *some* minimum spanning tree contains edge e.



20

Kruskal's algorithm

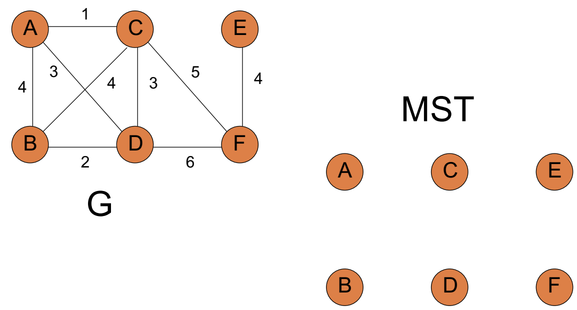
Given a partition S, let edge e be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge e.

```
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))
```

21

Kruskal's algorithm

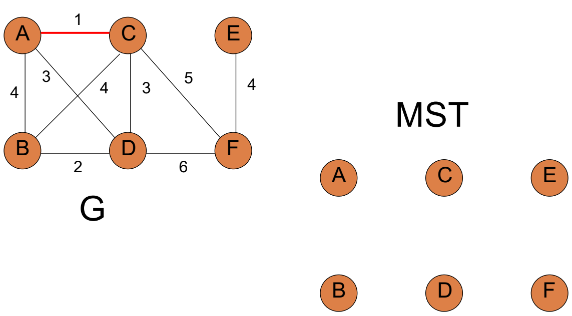
Add smallest edge that connects two sets not already connected



22

Kruskal's algorithm

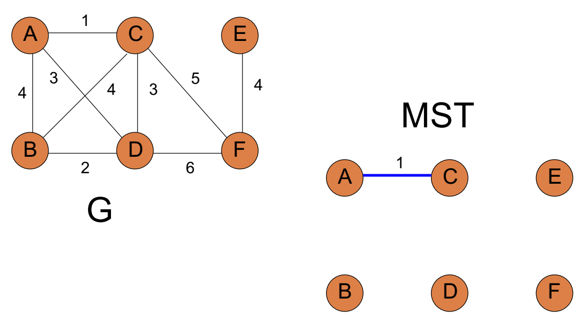
Add smallest edge that connects two sets not already connected



23

Kruskal's algorithm

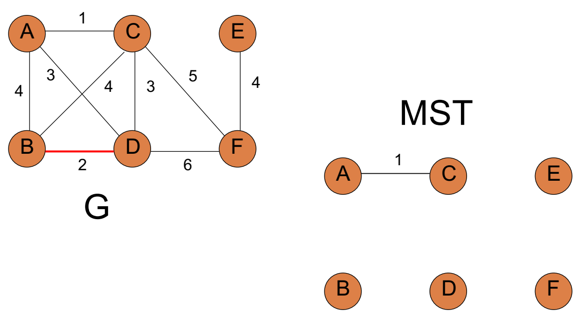
Add smallest edge that connects two sets not already connected



24

Kruskal's algorithm

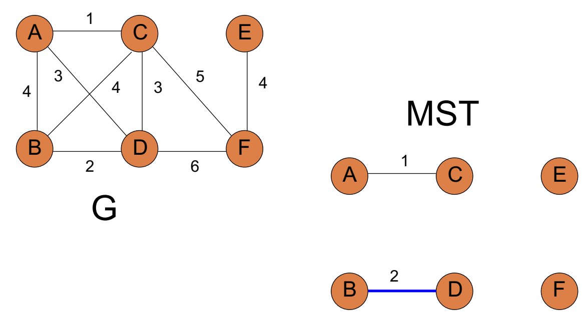
Add smallest edge that connects two sets not already connected



25

Kruskal's algorithm

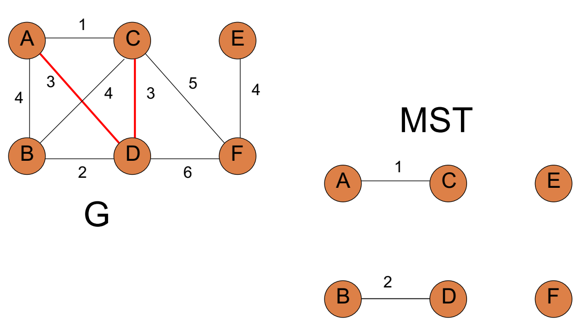
Add smallest edge that connects two sets not already connected



26

Kruskal's algorithm

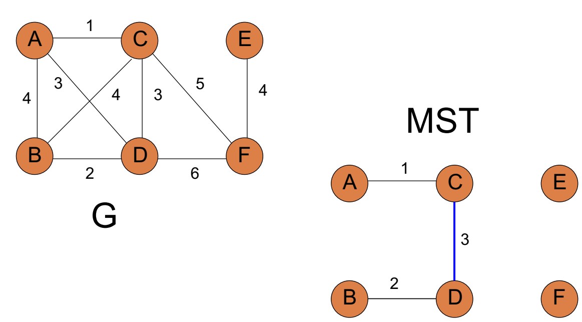
Add smallest edge that connects two sets not already connected



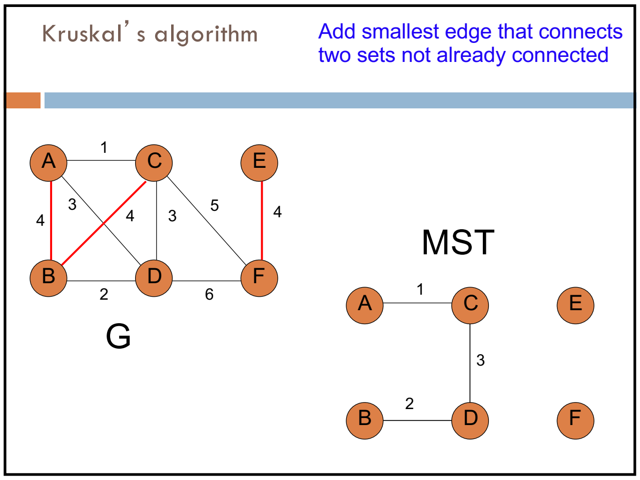
27

Kruskal's algorithm

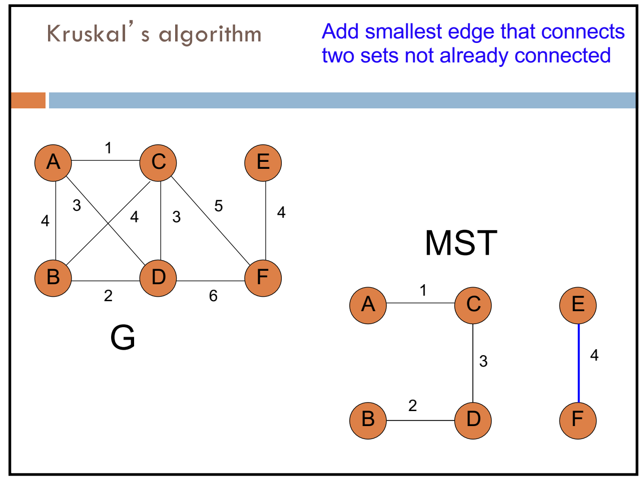
Add smallest edge that connects two sets not already connected



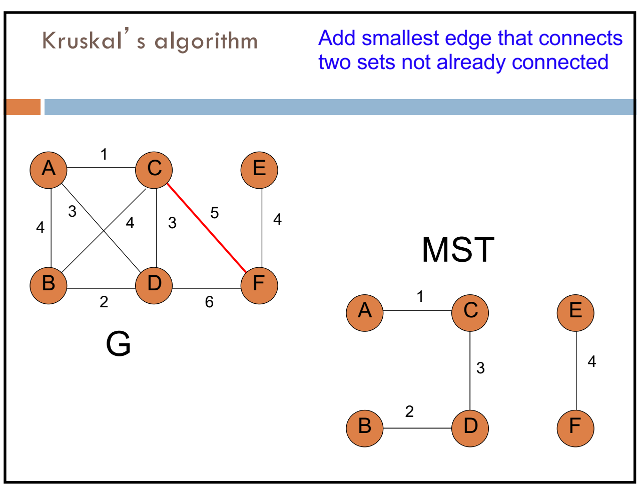
28


Kruskal's algorithm
Add smallest edge that connects
two sets not already connected
G
MST

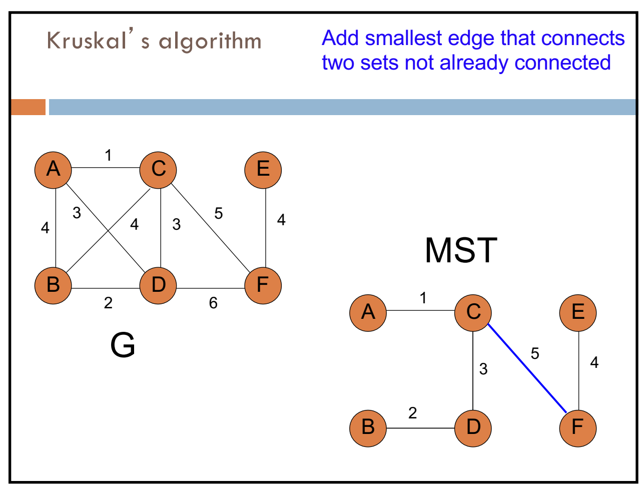
29


Kruskal's algorithm
Add smallest edge that connects
two sets not already connected
G
MST

30


Kruskal's algorithm
Add smallest edge that connects
two sets not already connected
G
MST

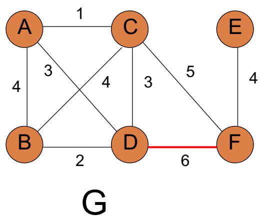
31

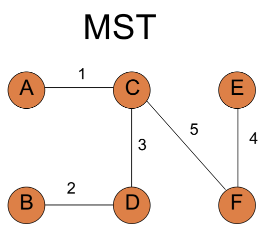

Kruskal's algorithm
Add smallest edge that connects
two sets not already connected
G
MST

32

Kruskal’s algorithm

Add smallest edge that connects two sets not already connected

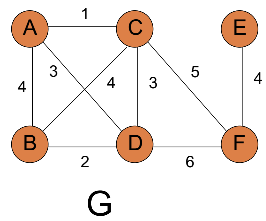




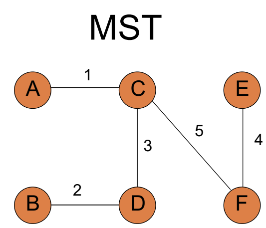
33

Kruskal’s algorithm

Add smallest edge that connects two sets not already connected



Done!



34

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))
```

35

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))
```

36

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

37

Disjoint set data structures

Represents a collection of one or more sets

Operations:

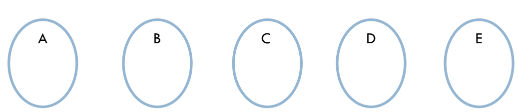
- MakeSet: Add a new value to the collections and make the value it's own set
- FindSet: Given a value, return the set the value is in
- Union: Merge two sets into a single set

38

Disjoint set data structure

MakeSet(A), MakeSet(B), MakeSet(C), MakeSet(D), MakeSet(E)

Disjoint Set

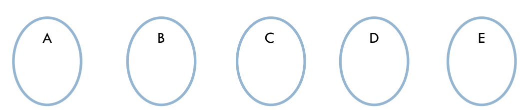


39

Disjoint set data structure

FindSet(A)?

Disjoint Set

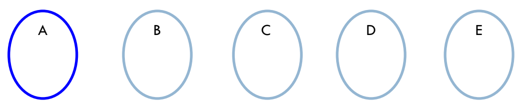


40

Disjoint set data structure

FindSet(A)?

Disjoint Set

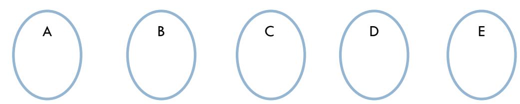


41

Disjoint set data structure

Union(FindSet(A), FindSet(E))

Disjoint Set

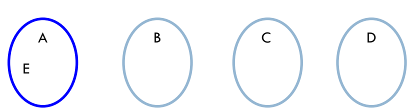


42

Disjoint set data structure

Union(FindSet(A), FindSet(E))

Disjoint Set

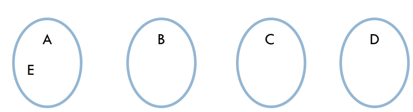


43

Disjoint set data structure

Union(FindSet(C), FindSet(D))

Disjoint Set

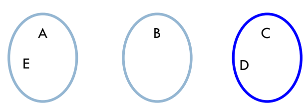


44

Disjoint set data structure

Union(FindSet(C), FindSet(D))

Disjoint Set

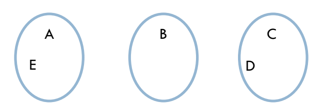


45

Disjoint set data structure

FindSet(D)?

Disjoint Set

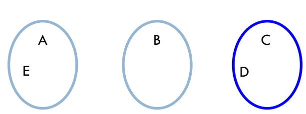


46

Disjoint set data structure

FindSet(D)?

Disjoint Set

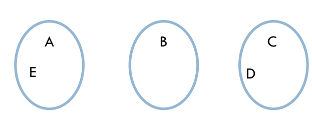


47

Disjoint set data structure

Union(FindSet(D), FindSet(B))

Disjoint Set



48

Disjoint set data structure

Union(FindSet(D), FindSet(B))

Disjoint Set

49

Disjoint set data structure

How would we implement it with a list of linked lists?

MakeSet?

FindSet?

Union?

Disjoint Set

50

Disjoint set

A B C D

51

Disjoint set: union

A B C D

52

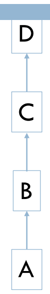
Disjoint set: union

53

Disjoint set: union

54

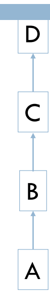
Disjoint set: union



Running time?

55

Disjoint set: union



O(1)

56

Disjoint set: find-set

Search each linked list

57

Disjoint set: find-set

Running time?

58

Disjoint set: find-set

O(n) -- n = number of things in set

59

Running time of Kruskal's

Disjoint set data structure

O(|E| log |E|) +

	MakeSet (V calls)	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|)

60

Prim's algorithm

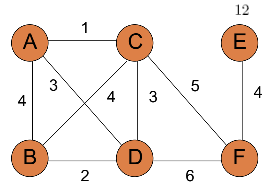
Start at some root node and build out the MST by adding the lowest weighted edge at the frontier

```
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
```

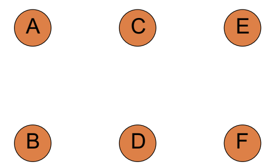
64

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
```



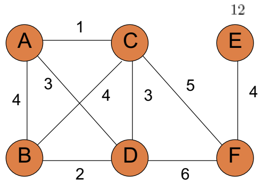
MST



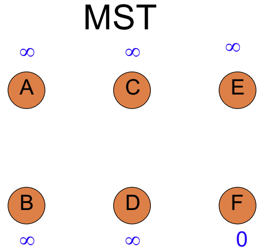
65

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
```



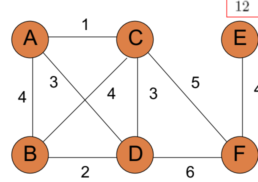
MST



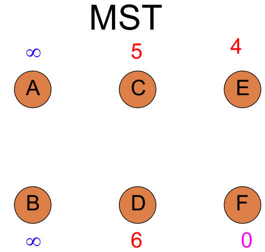
66

Prim's

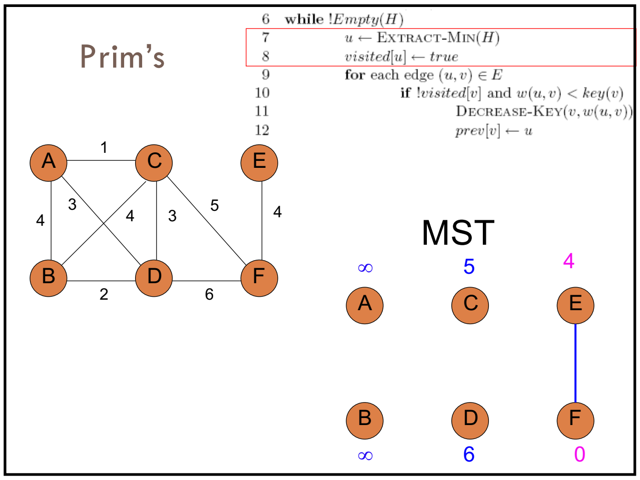
```
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```



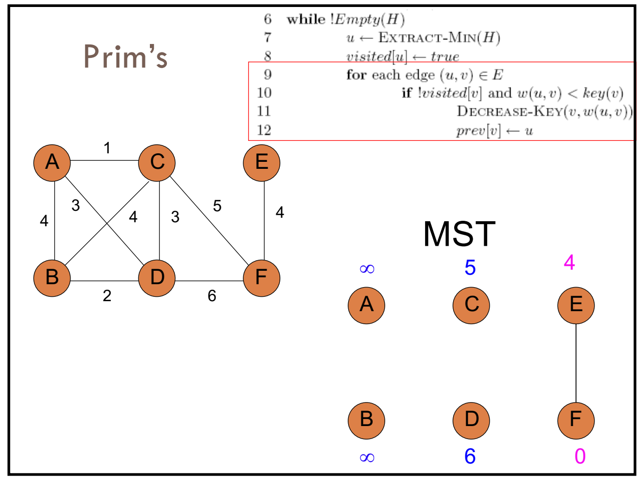
MST



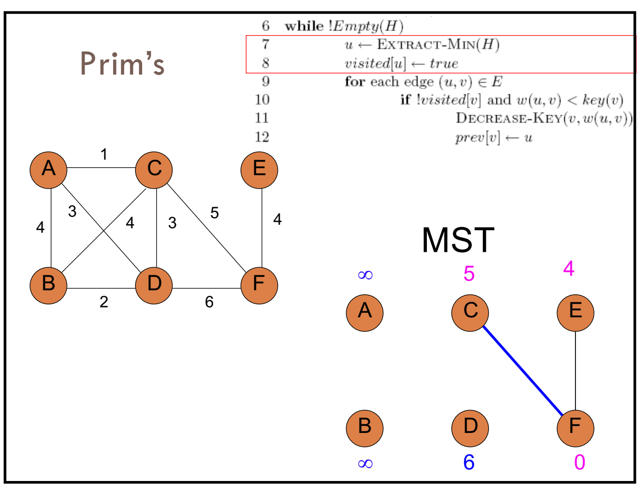
67


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

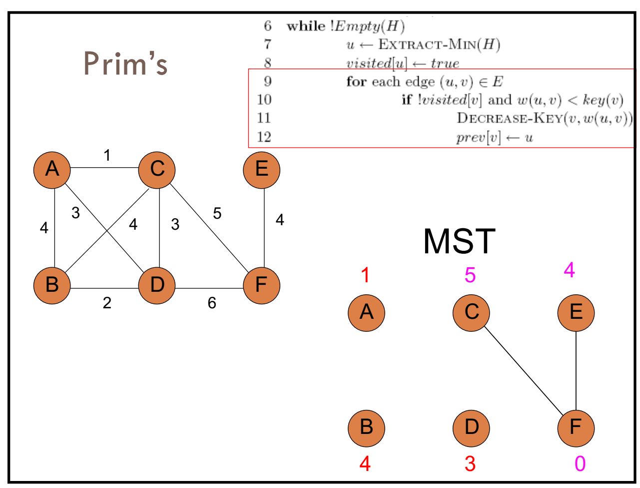
68


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

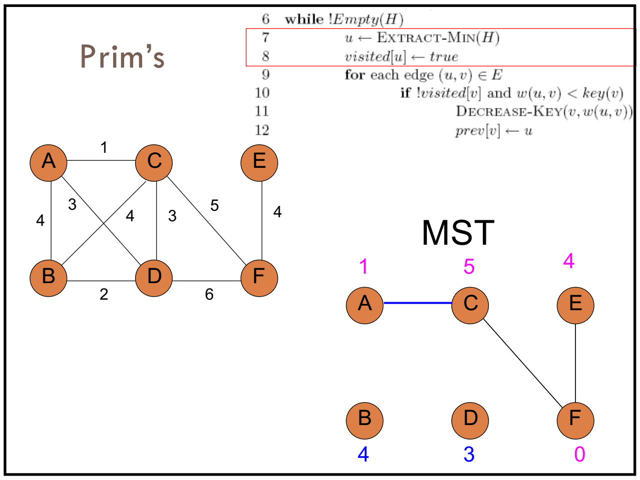
69


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

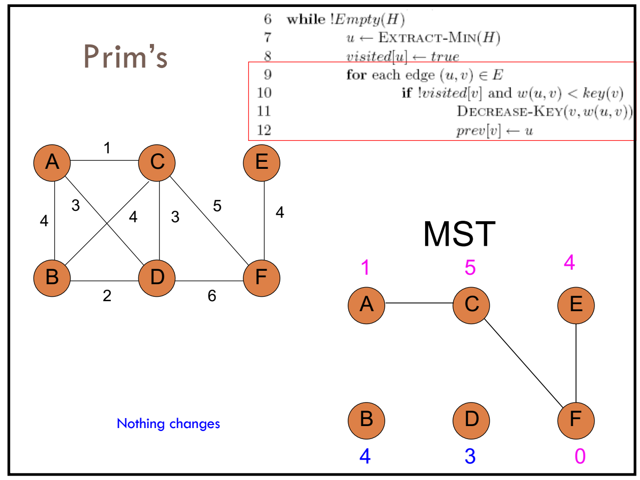
70


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

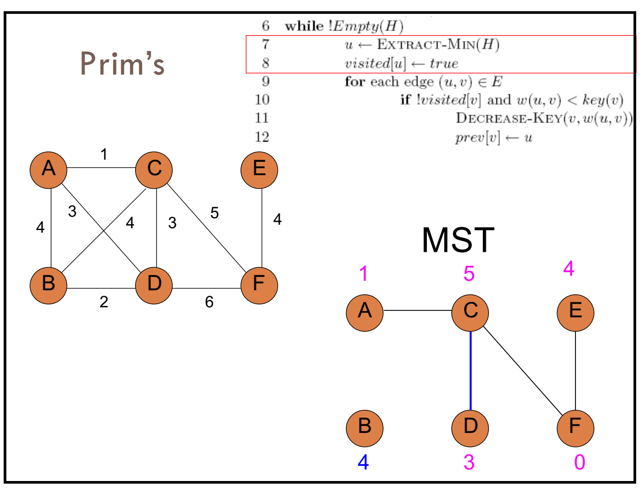
71


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

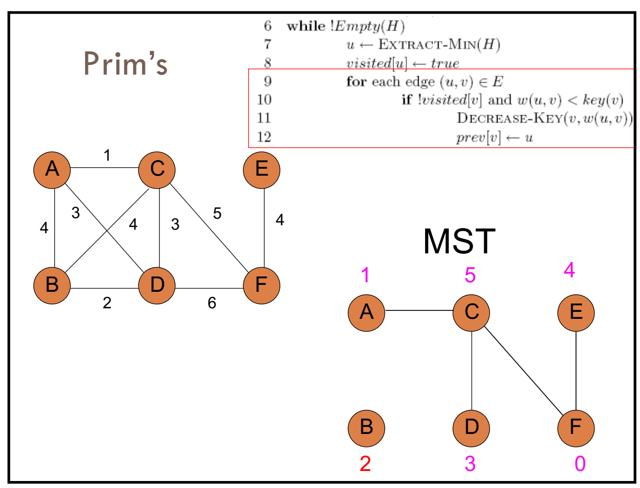
72


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
Nothing changes

73


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

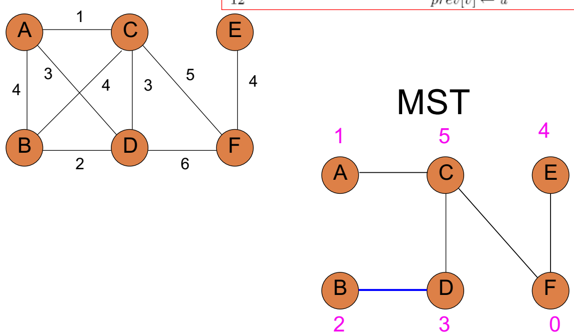
74


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

75

Prim's

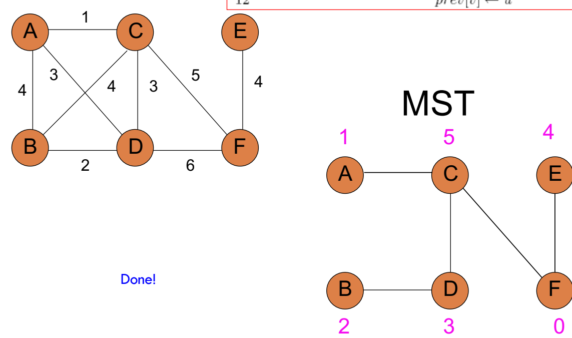
```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u,v) ∈ E
        if !visited[v] and w(u,v) < key(v)
            Decrease-Key(v, w(u,v))
            prev[v] ← u
```



76

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
```



77

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

78

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
```

79

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
```

$\Theta(|V|)$

1 call to MakeHeap

|V| calls to Extract-Min

|E| calls to Decrease-Key

80

Running time of Prim's

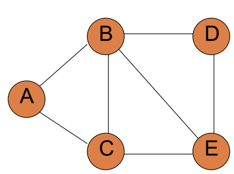
	1 MakeHeap	|V| ExtractMin	|E| DecreaseKey	Total
Array	θ(|V|)	$O(\|V\|^2)$	O(|E|)	$O(\|V\|^2)$
Bin heap	θ(|V|)	O(|V| log |V|)	O(|E| log |V|)	O((|V|+|E|) log |V|) O(|E| log |V|)
Fib heap	θ(|V|)	O(|V| log |V|)	O(|E|)	**O(|V| log |V| + |E|)**

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

81

Shortest paths

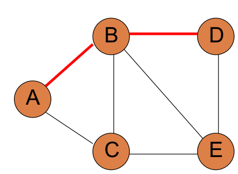
What is the shortest path from a to d?



82

Shortest paths

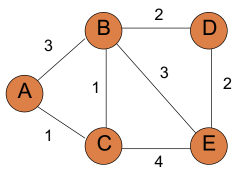
BFS



83

Shortest paths

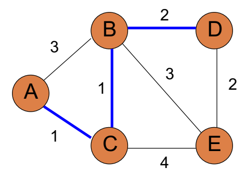
What is the shortest path from a to d?



84

Shortest paths

What is the shortest path from a to d?



85

Shortest path algorithms?

86

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
```

What is dist?

What is prev?

How does it work?

What is the run-time?

How do we get the shortest path?

87

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
```

88

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
```

89

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
```

90

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
```

91

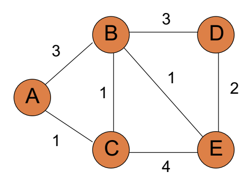
Dijkstra's algorithm

```
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
```

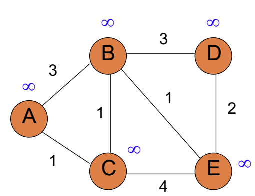
92

```
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
```



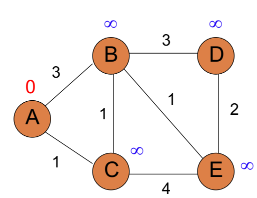
93

```
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
```



94

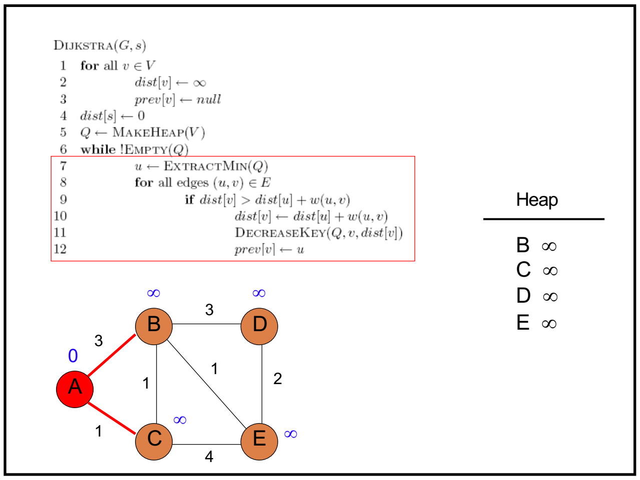
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                      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



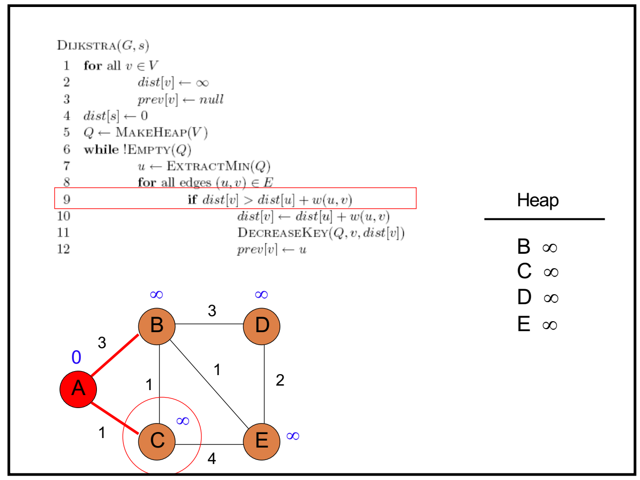
Heap

A 0
B ∞
C ∞
D ∞
E ∞

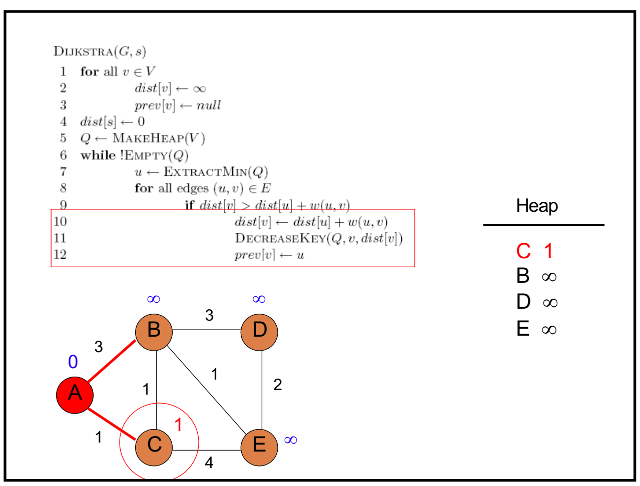
95


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 ∞

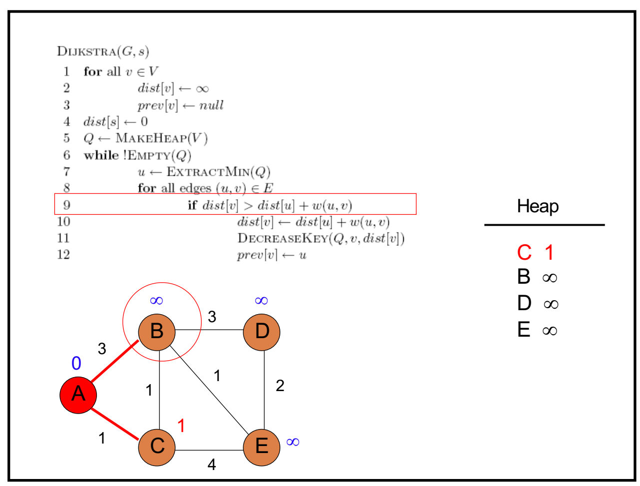
96


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 ∞

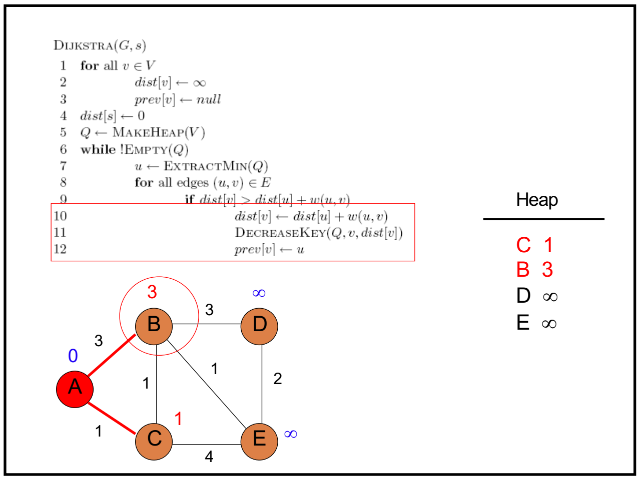
97


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 ∞

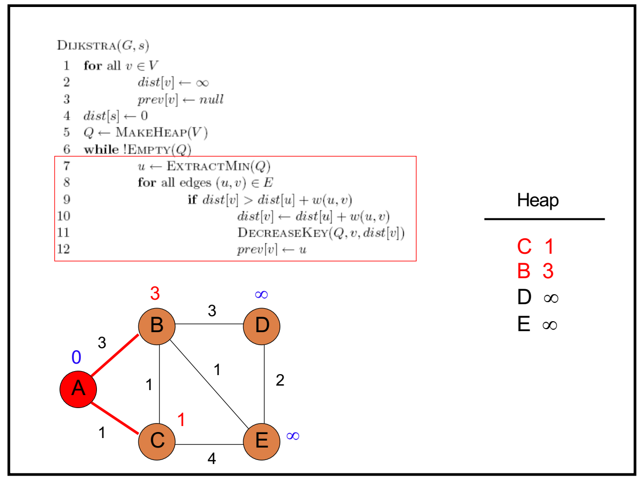
98


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 ∞

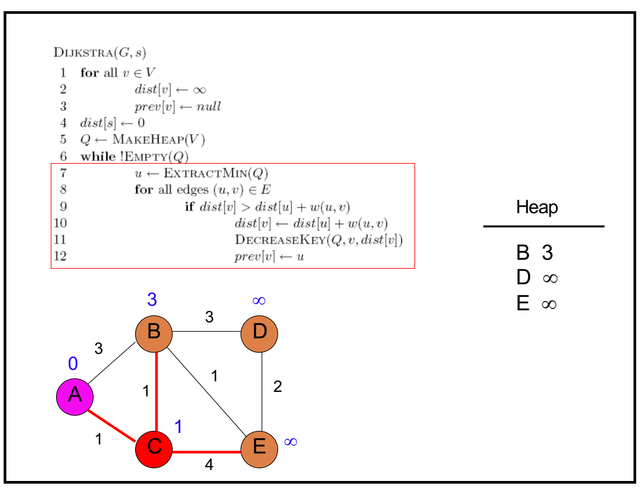
99



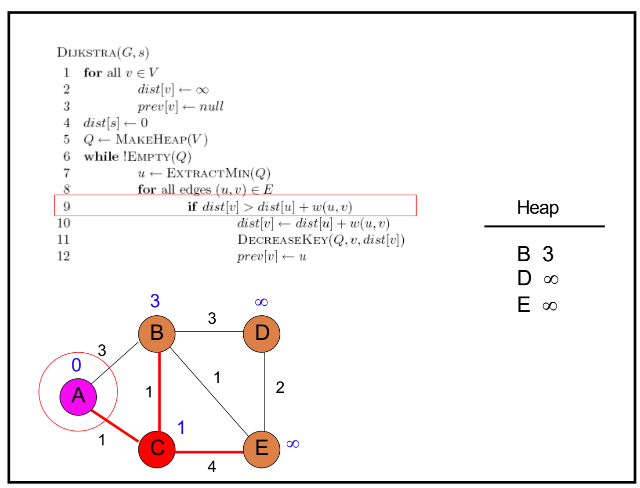
100



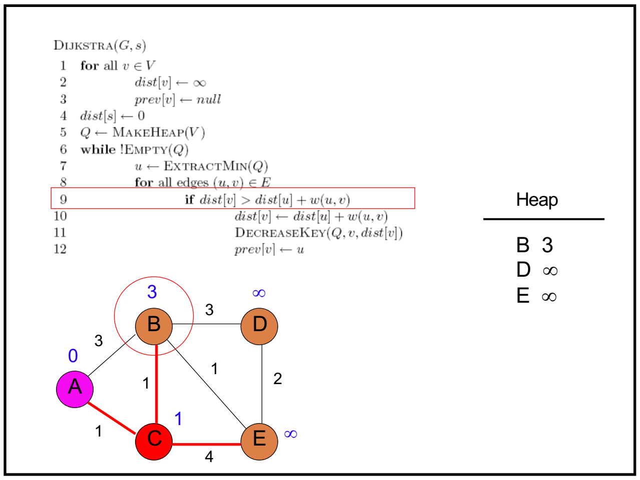
101



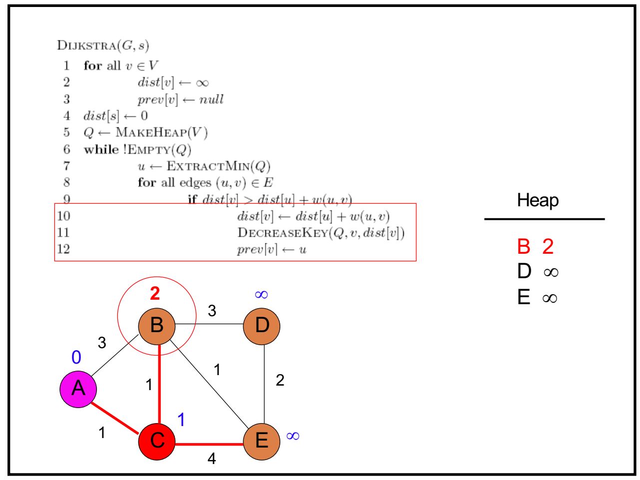
102



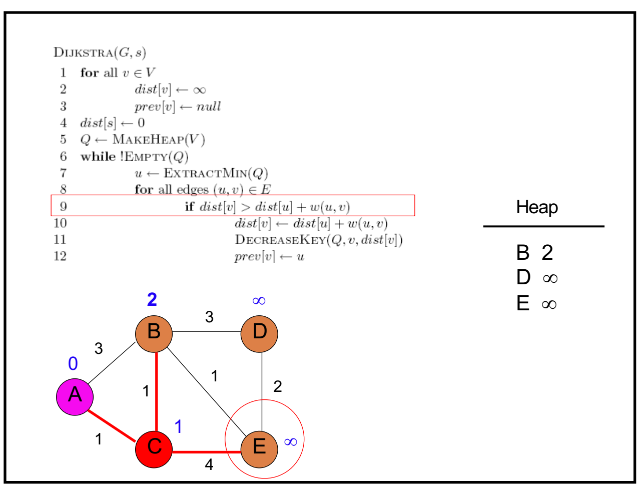
103



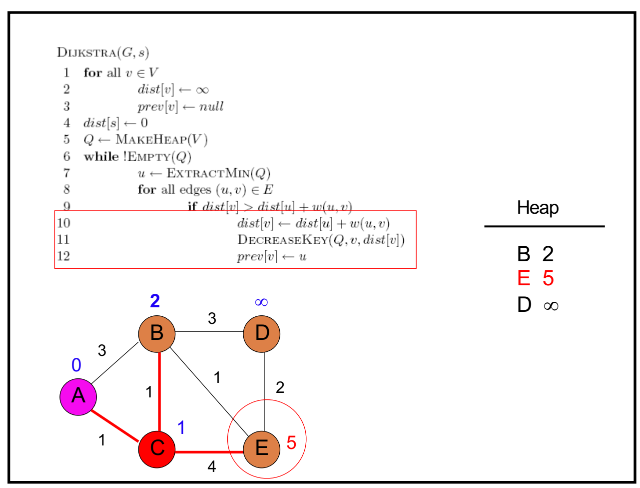
104



105



106

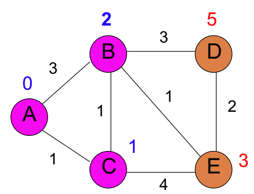


107

```
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

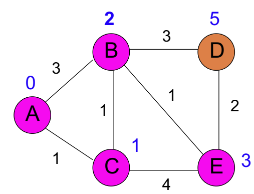


108

```
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

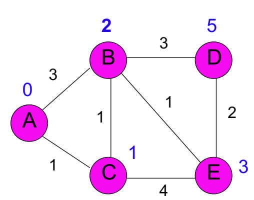
D 5



109

```
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
```

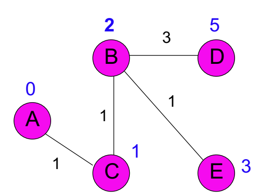
Heap



110

```
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

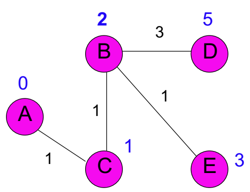


Prev

111

```
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



Prev

How do we get the actual paths?

112

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?

113

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

114

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
```

115

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

116

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

117

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

118

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

119

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|)

120

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 $|E| < |V|^2 / \log |V|$

121

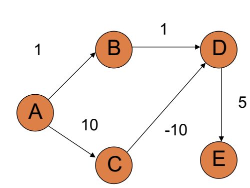
Running time?

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|)

122

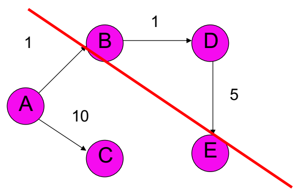
What about Dijkstra's on…?



123

What about Dijkstra's on…?

Dijkstra's algorithm only works for positive edge weights



124

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

We relied on having positive edge weights for correctness!

125

Dijkstra's vs Prim's

```
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(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
```

126