# Data Structures
# MST 2

CS 225
Brad Solomon

November 1, 2024

A Big O
day

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

# Learning Objectives

Review the minimum spanning tree (with weights)

Review Kruskal's / Prim's MST Algorithms

Focus on determining Big O of complex pseudocode

Compare implementations under different conditions

# Summary: DFS and BFS

$|V| = n, |E| = m$

Both are **O(n+m)** traversals! They label every edge and every node

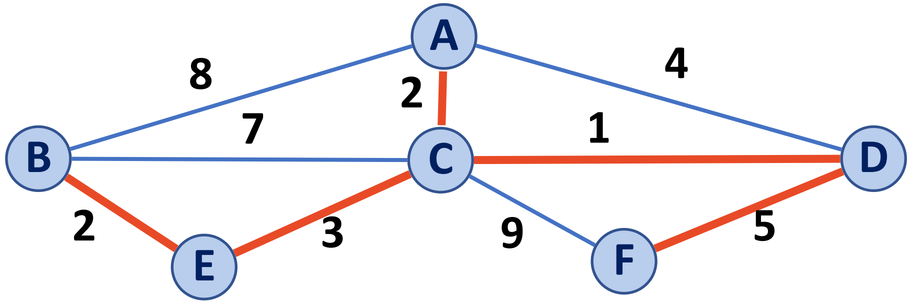
| **BFS** | **DFS** |
|---|---|
| Solves unweighted MST | Solves unweighted MST |
| Solves shortest path | |
| Solves cycle detection | Solves cycle detection |
| Memory bounded by width | Memory bounded by longest path |

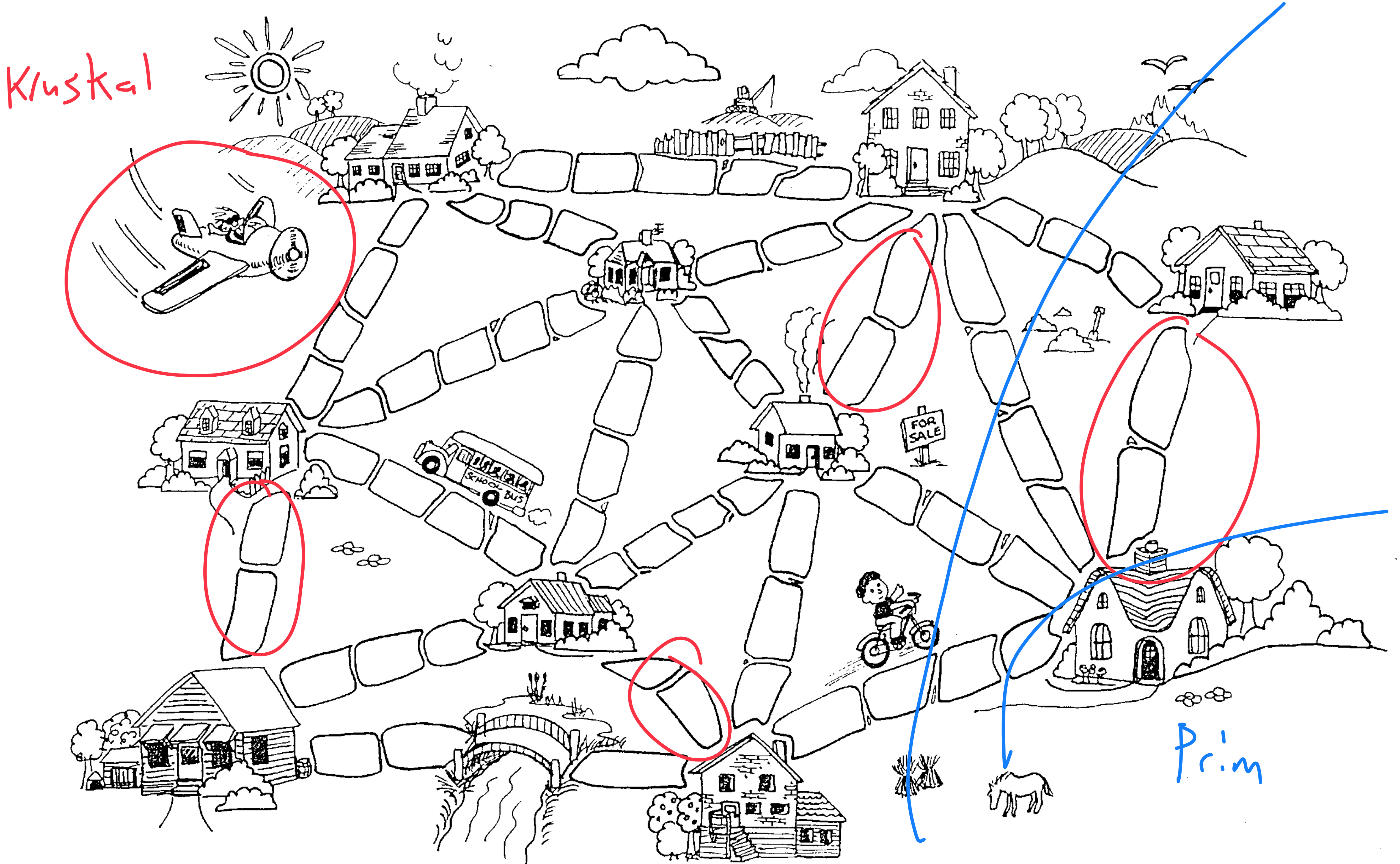
# Minimum Spanning Tree Algorithms

**Input:** Connected, undirected graph **G** with edge weights (unconstrained, but must be additive)

**Output:** A graph G’ with the following properties:

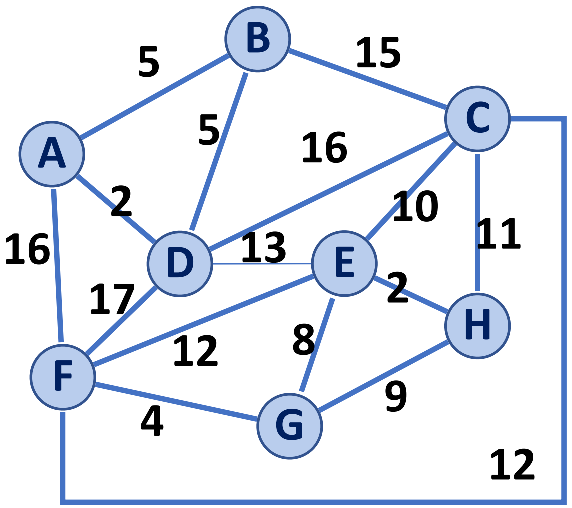
- G’ is a spanning graph of G
- G’ is a tree (connected, acyclic)
- G’ has a minimal total weight among all spanning trees

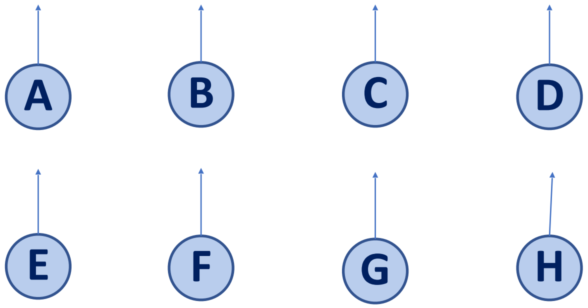



Kruskal
FOR SALE
SCHOOL BUS
Prim

# Kruskal's Algorithm

| |
|---|
| (A, D) |
| (E, H) |
| (F, G) |
| (A, B) |
| (B, D) |
| (G, E) |
| (G, H) |
| (E, C) |
| (C, H) |
| (E, F) |
| (F, C) |
| (D, E) |
| (B, C) |
| (C, D) |
| (A, F) |
| (D, F) |




1) Build a **priority queue** on edges

↳ min heap

↳ sorted list

2) Build a **disjoint set** on vertices

↳ ~O(1) time

3) Repeatedly find min edge

If edge connects two sets

Union and record edge

4) Stop after n-1 edges recorded

↳ everything in same set

# Kruskal's Algorithm

```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G.vertices()):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  Q.buildFromGraph(G.edges())

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
       T.addEdge(u, v)
       forest.union( forest.find(u),
                     forest.find(v) )

  return T
```

Output tree

1) Build a **priority queue** on edges

2) Build a **disjoint set** on vertices

3) Repeatedly find min edge

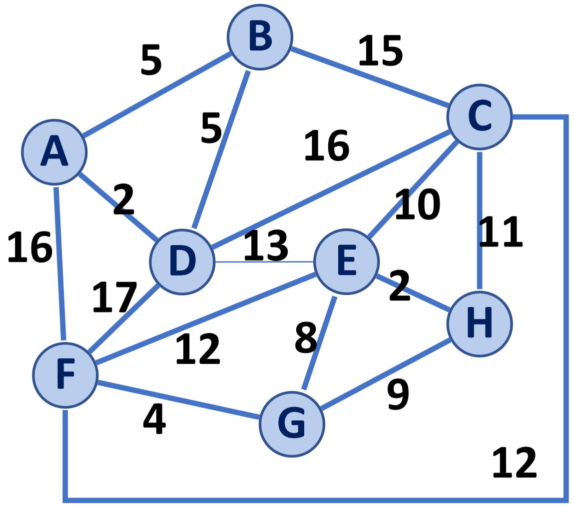
   If edge connects two sets

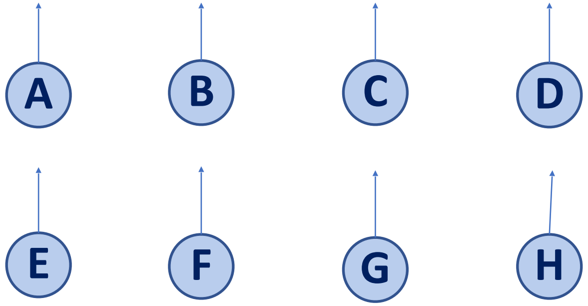
   Union and record edge

4) Stop after n-1 edges recorded

# Kruskal's Algorithm

- (A, D)
- (E, H)
- (F, G)
- (A, B)
- (B, D)
- (G, E)
- (G, H)
- (E, C)
- (C, H)
- (E, F)
- (F, C)
- (D, E)
- (B, C)
- (C, D)
- (A, F)
- (D, F)




```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G.vertices()):
    forest.makeSet(v)

  PriorityQueue Q     // min edge weight
  Q.buildFromGraph(G.edges())

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
       T.addEdge(u, v)
       forest.union( forest.find(u),
                     forest.find(v) )

  return T
```

# Kruskal's Algorithm

Big O?

|V| = n  |E| = m

```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G.vertices()):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  Q.buildFromGraph(G.edges())

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
      T.addEdge(u, v)
      forest.union( forest.find(u),
                    forest.find(v) )

  return T
```

O(n)

Heap: O(m)
Sorted list: O(m log m)

m ×

removeMin
Heap: O(log m)
Sorted list: O(1)

O(1) ← b/c path compression
Smart union
we have
inverse Ackerman

O(1)

# Kruskal's Algorithm

|V| = n    |E| = m

| Priority Queue: | Heap | Sorted Array |
|---|---|---|
| **Building** :7 | O(m) | O(m log m) |
| **Each removeMin** :12    m X | O(log m) | O(1) |

m + m log m   vs   m log m + m

why heap good?
↳ What if edge weight changes?

why sorted array good?
↳ Sorted array not destroyed when used = if we could use array later, this is better!

```
1  KruskalMST(G):
2    DisjointSets forest                              O(n)
3    foreach (Vertex v : G.vertices()):
4      forest.makeSet(v)
5
6    PriorityQueue Q    // min edge weight
7    Q.buildFromGraph(G.edges())   <-
8
9    Graph T = (V, {})
10                                     mx
11   while |T.edges()| < n-1:
12     Vertex (u, v) = Q.removeMin()   <-
13     if forest.find(u) != forest.find(v):       }
14       T.addEdge(u, v)                          }
15       forest.union( forest.find(u),            }  O(1)
16                     forest.find(v) )           }
17
18   return T
19
```

# Kruskal's Algorithm

$n-1 \le m \le n^2$ ← $\frac{n(n-1)}{2}$, $O(n^2)$

| Priority Queue: | Total Running Time |
|---|---|
| **Heap** | $O(n) + O(m) + O(m \log m)$ |
| **Sorted Array** | $O(n) + O(m \log m) + O(m)$ |

Unsorted array: $O(n)$, $O(1) + O(m^2)$
↳ Not good!

$O(\log m) \approx O(\log n)$

$\log n^2$

$2 \log n$

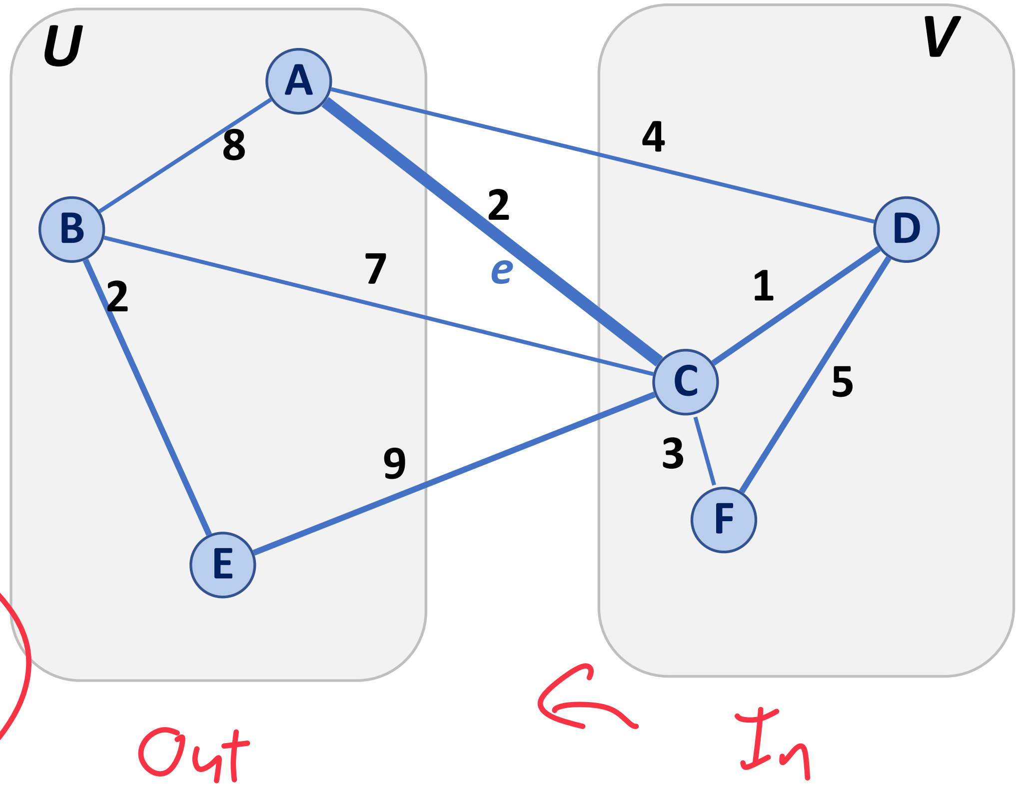
```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G.vertices()):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  Q.buildFromGraph(G.edges())

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
       T.addEdge(u, v)
       forest.union( forest.find(u),
                     forest.find(v) )

  return T
```

# Partition Property

Consider an arbitrary partition of the vertices on **G** into two subsets **U** and **V**.

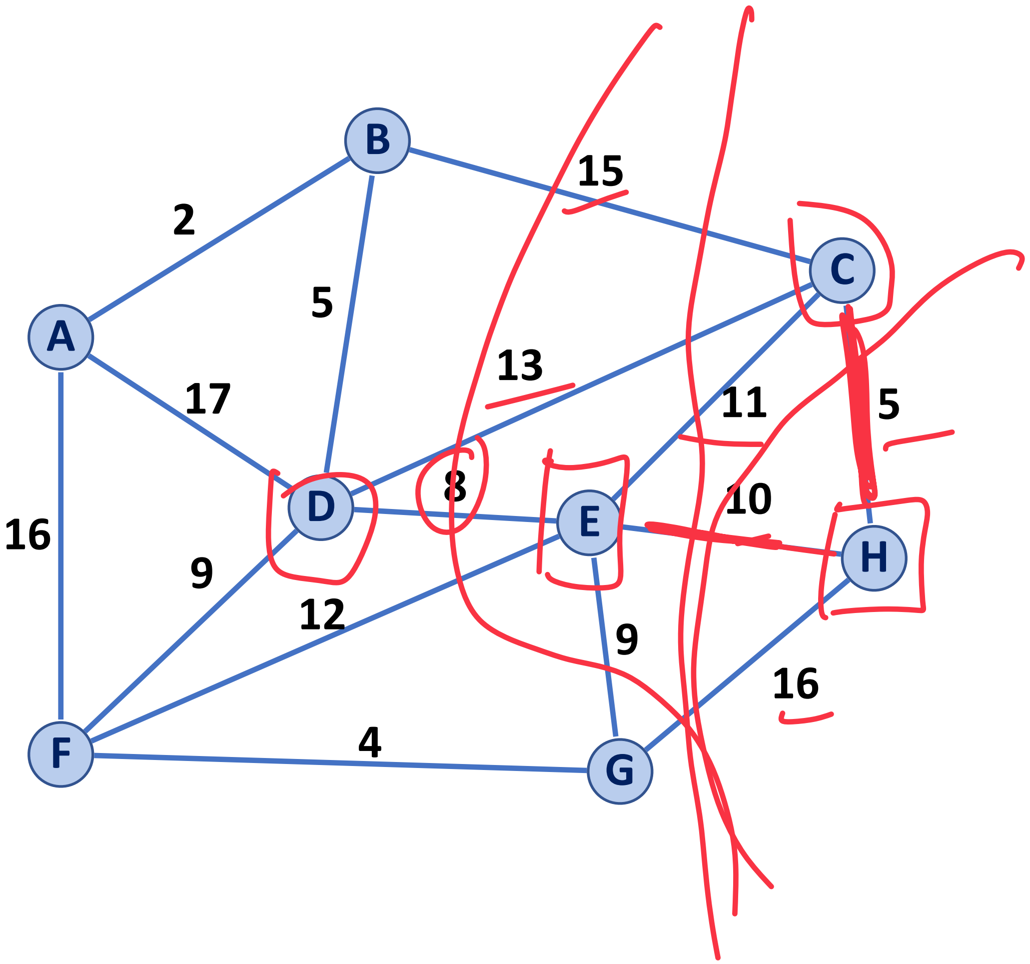
Let **e** be an edge of minimum weight across the partition.

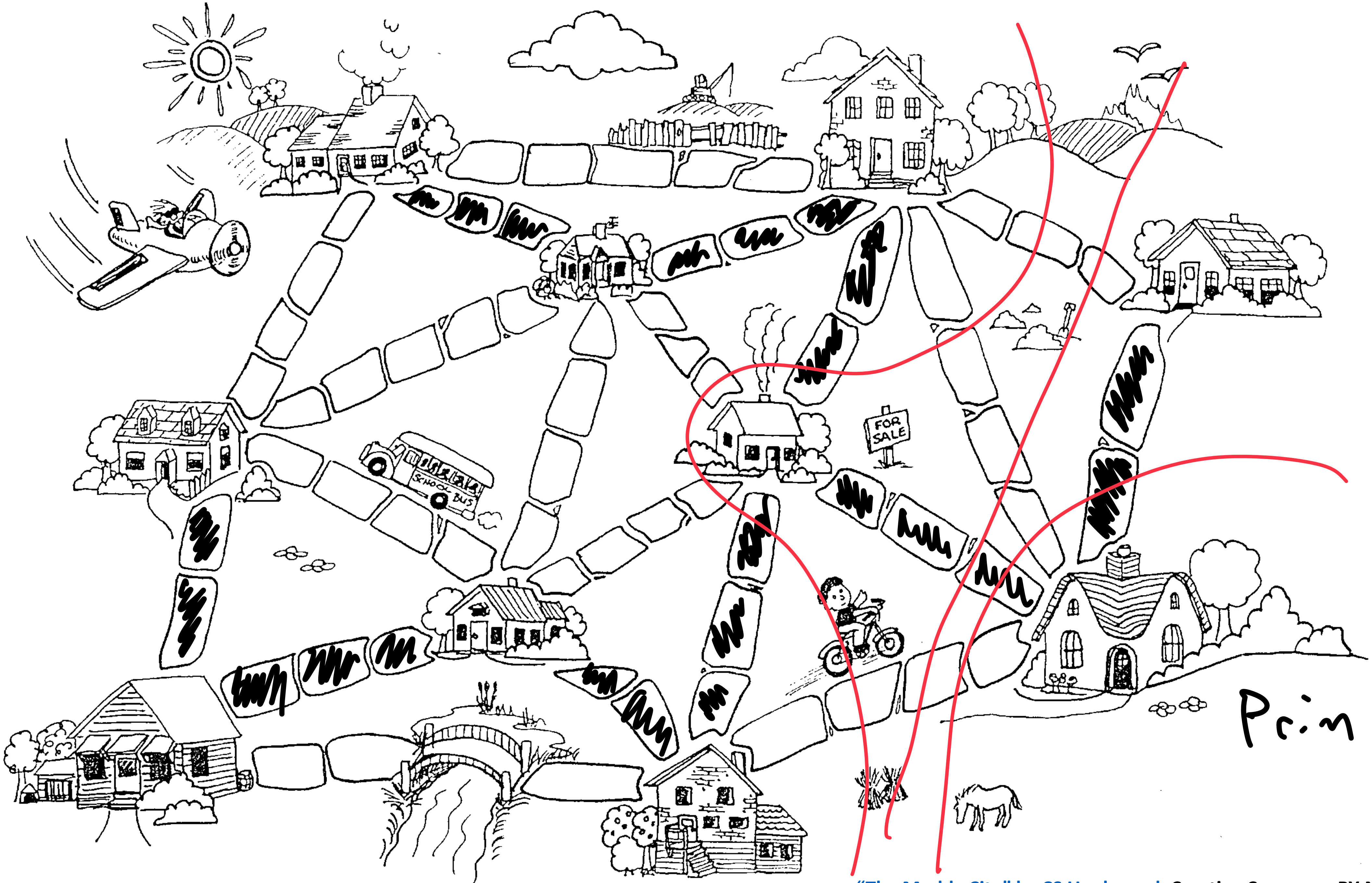
Then **e** is part of some minimum spanning tree.



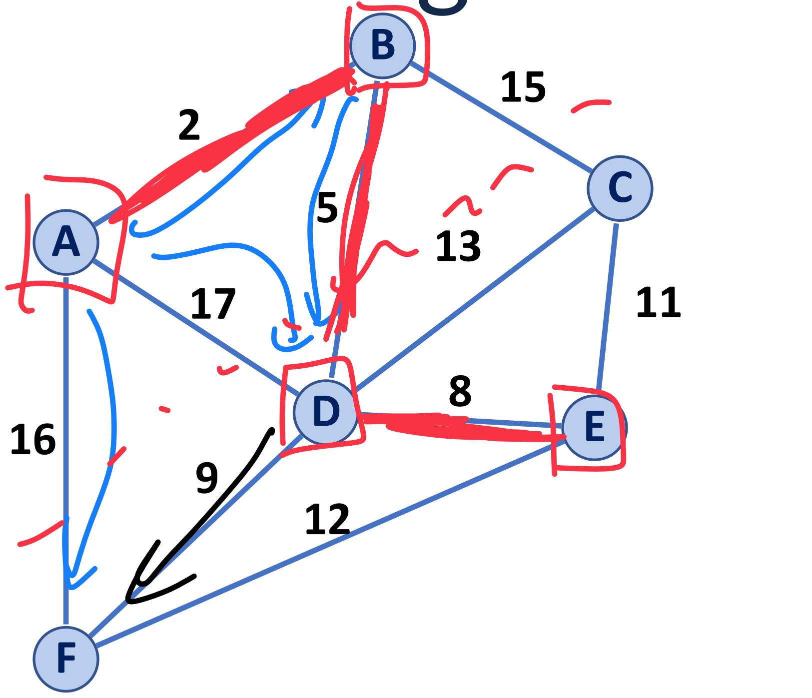
# Partition Property

The partition property suggests an algorithm:




SCHOOL BUS
FOR SALE
Prim

# Prim’s Algorithm



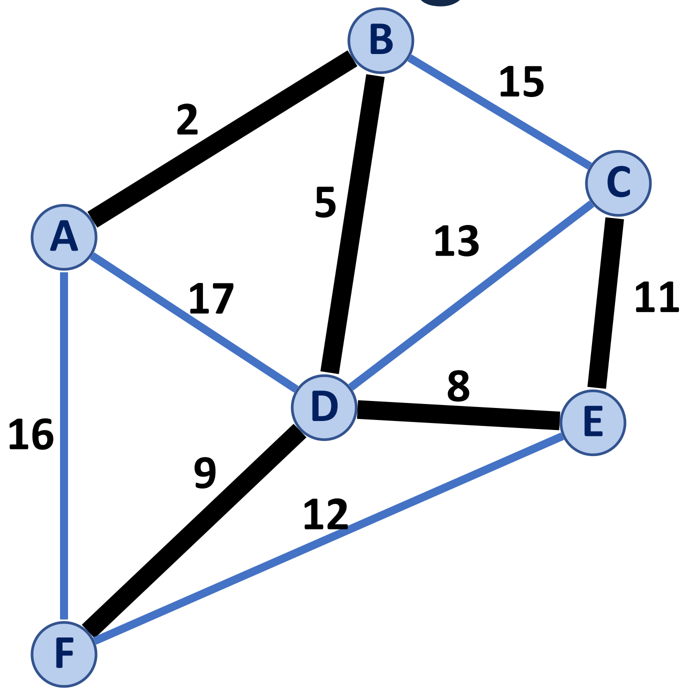

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 0 | ~~∞~~ 2,A | ~~∞~~ ~~15,B~~ ~~13,D~~ 11,E | ~~∞~~ ~~17,A~~ 5,B | ~~∞~~ 8,D | ~~∞~~ ~~16,A~~ 9,D |

```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q   // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

  return T
```

Init

update all neighbors if new smaller edge

# Prim’s Algorithm




| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 0, — | 2, A | 11, E | 5, B | 8, D | 9, D |

```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q   // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

  return T
```

# Prim's Big O

```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

O(n)

min heap or unsorted array

?? Depend on implementation

n x

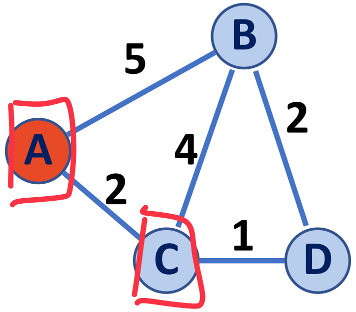
Adjacency matrix or Adjacency list

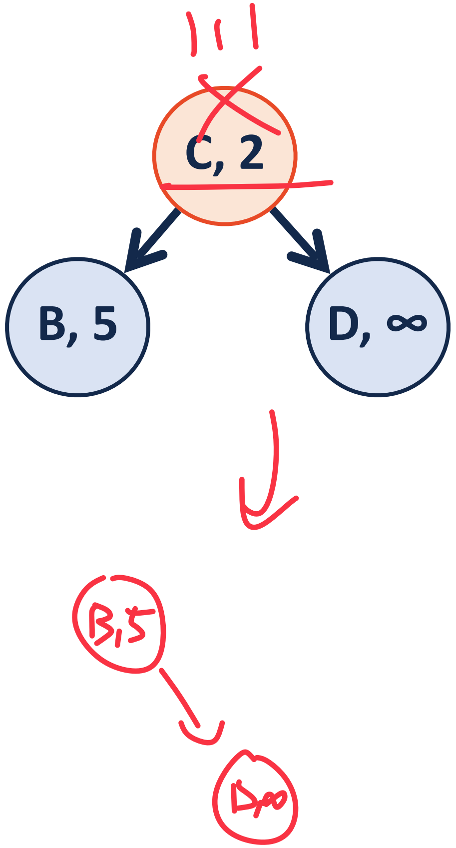
O(n²)

O(m)

$\sum deg(v) = 2|E|$

| A | B | C | D |
|---|---|---|---|
| 0 | 5 | 2 | ∞ |

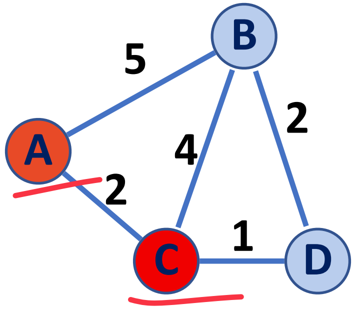


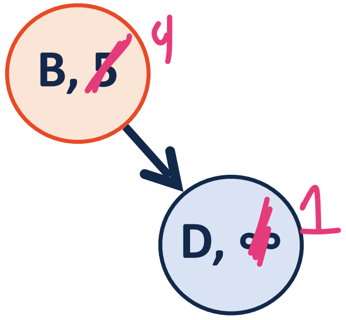




```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T            // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()   ← O(log n)
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

| | Adj. Matrix | Adj. List |
|---|---|---|
| **Heap** | O(n) + O(n log n) + O(n^2) + ______________ | O(n) + O(n log n) + O(m) + ______________ |

| A | B | C | D |
|---|---|---|---|
| 0 | ~~5~~ 4 | 2, A | ~~∞~~ 1 |




```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

n x

1) change value O(1)

2) heapify up()

Adj Matrix

O(1)

$m \le n^2$

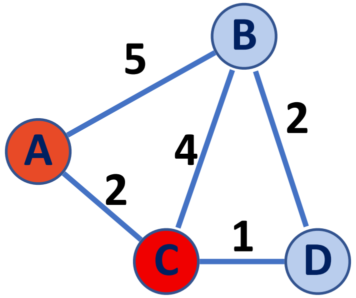
$n^2 \log n$

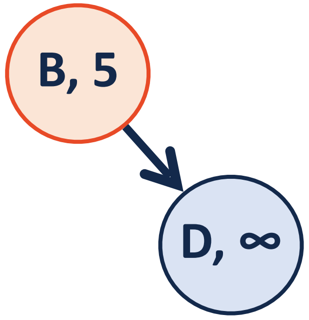
$\sum_v \deg(v) = m$ x

O(log n)

| | Adj. Matrix | Adj. List |
|---|---|---|
| **Heap** | O(n) + O(n log n) + O(n^2) + O(m log n) | O(n) + O(n log n) + O(m) + O(m log n) |

| A | B | C | D |
|---|---|---|---|
| 0 | 5 | 2, A | ∞ |




```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

1) Change minheap value

2) HeapifyUp()

O(n)

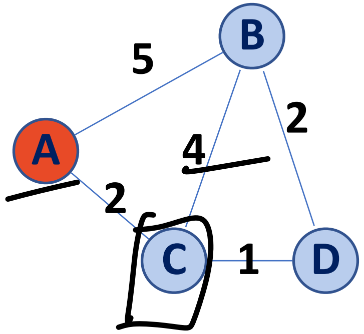
O(n)

O(n²) << O(m)

Only happens for edges deg(v)

| | Adj. Matrix | Adj. List |
|---|---|---|
| **Heap** | O(n) + O(n log n) + O(n^2) + O(m log n) | O(n) + O(n log n) + O(m) + O(m log n) |

(A, 0)
(D, ∞)
(C, 2)
(B, 5)




```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

| | Adj. Matrix | Adj. List |
|---|---|---|
| Heap | $O(n^2 + m \lg(n))$ | $O(n \lg(n) + m \lg(n))$ |
| Unsorted Array | $O(n^2)$ | $O(n^2)$ |

# Prim's Algorithm

Sparse Graph: $n \sim m$

↳ heap is better

Dense Graph: $m \sim n^2$

↳ unsorted array better

```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

$n-1 \le m \le n^2$

$m = n^2$

| | Adj. Matrix | Adj. List |
|---|---|---|
| **Heap** | $O(n^2 + m \lg(n))$ → $n^2 \log n$ | sparse $O(n \log n)$ $O(n \lg(n) + m \lg(n))$ Dense $n^2 \log n$ |
| **Unsorted Array** | $O(n^2)$ | $m = n$ $O(n^2)$ |

# MST Algorithm Runtime:

Kruskal's Algorithm:
**O(n + m log (n) )**

Prim's Algorithm:
**O(n log(n) + m log (n) )**

Sparse Graph:

Dense Graph:

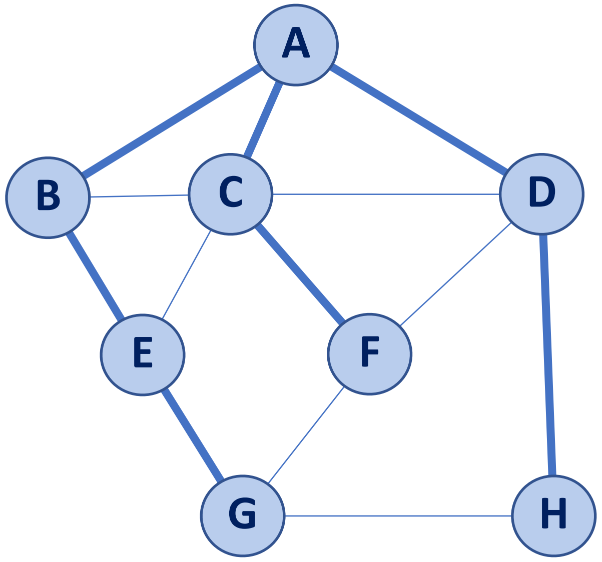
# Suppose I have a new heap:

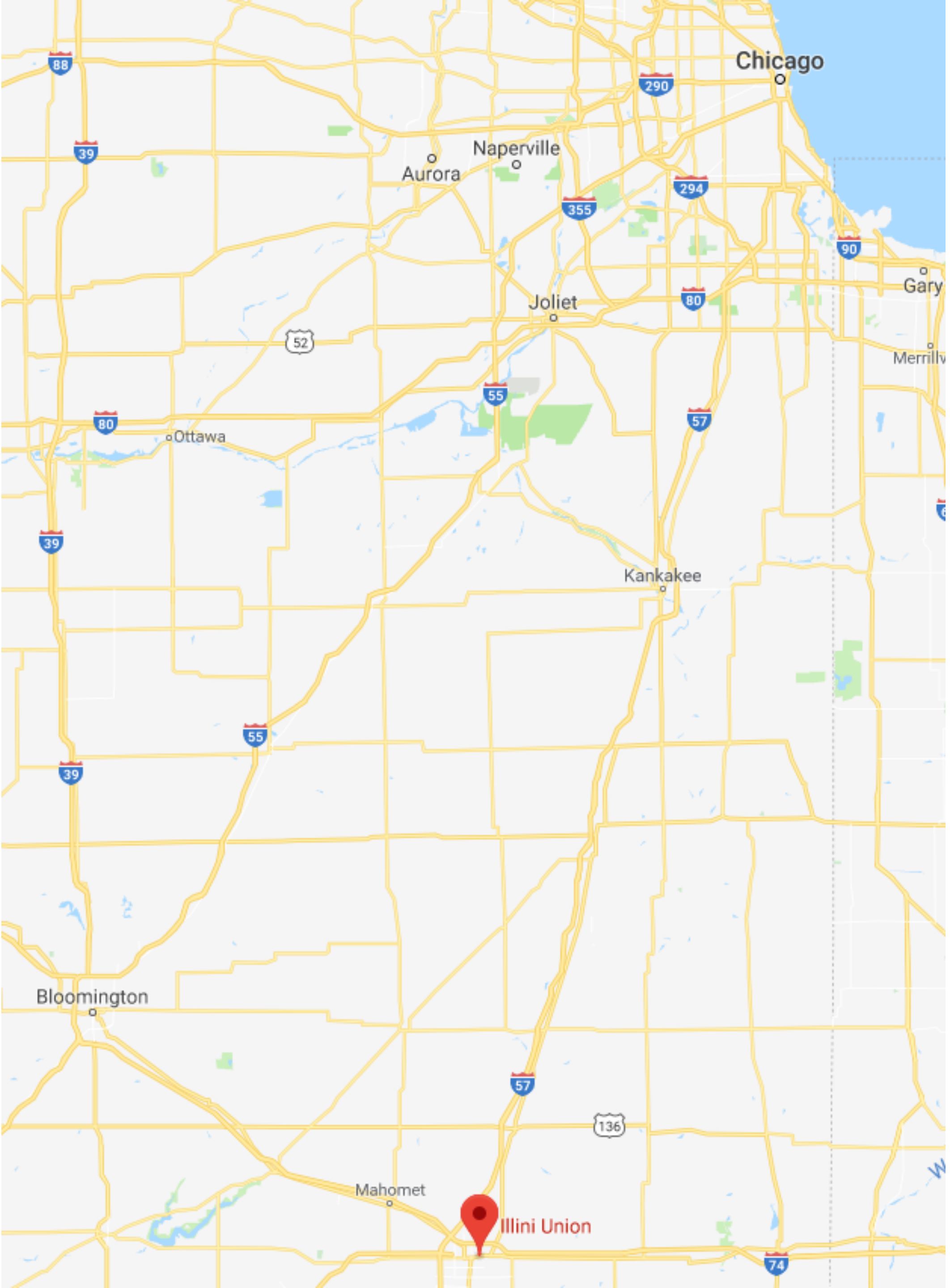
| | Binary Heap | Fibonacci Heap |
|---|---|---|
| **Remove Min** | O( lg(n) ) | O( lg(n) ) |
| **Decrease Key** | O( lg(n) ) | O(1)* |

**What's the updated running time?**

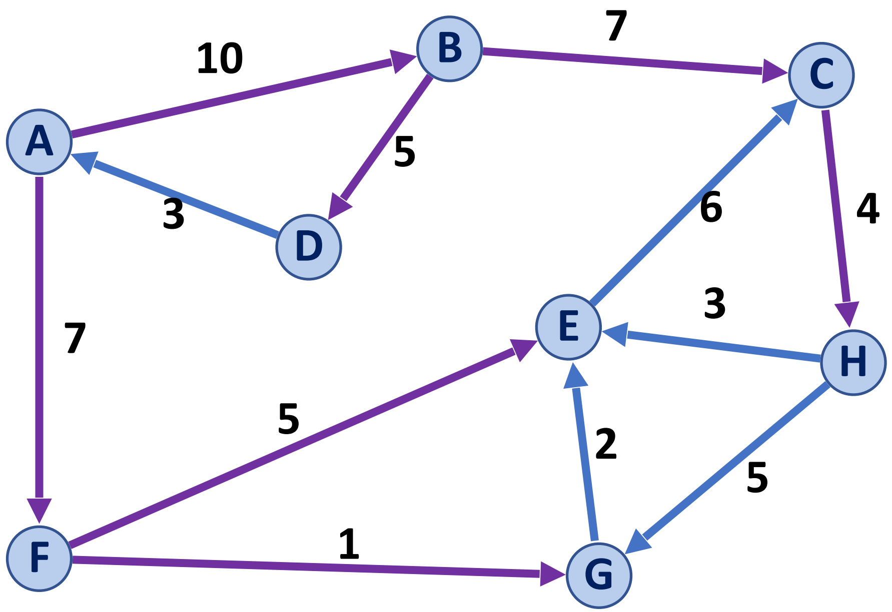
```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

# Shortest Path



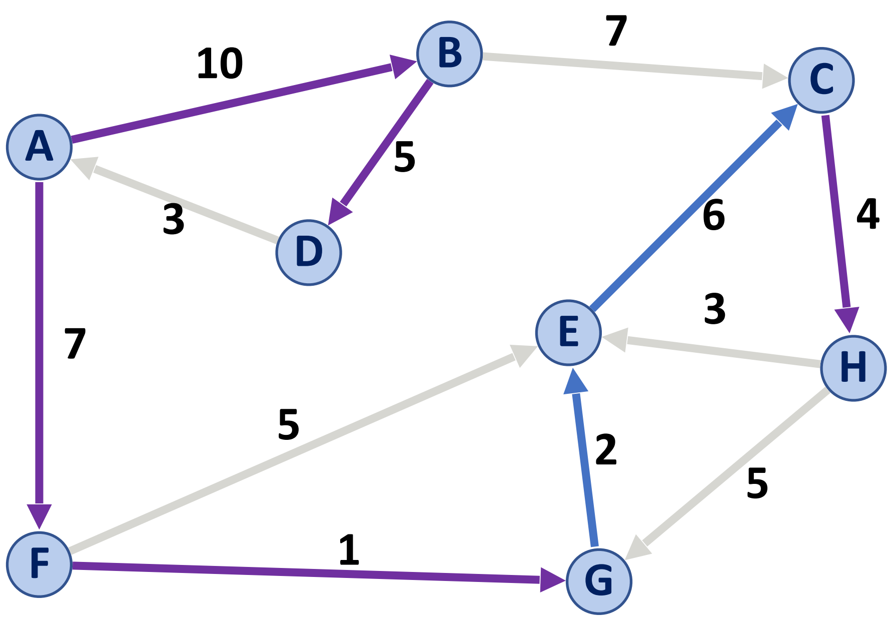
# Dijkstra's Algorithm (SSSP)




```
DijkstraSSSP(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if ________________ < d[v]:
        d[v] = ________________
        p[v] = u
```

| A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|
| -- | | | | | | | |
| 0 | | | | | | | |

# Dijkstra's Algorithm (SSSP)




```
DijkstraSSSP(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = u
```

| A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|
| -- | A | E | B | G | A | F | C |
| 0 | 10 | 16 | 15 | 10 | 7 | 8 | 20 |

# Dijkstra's Algorithm (SSSP)

What is the running time of Dijkstra's Algorithm?

```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m

  return T
```