Algorithms and Data Structures for Data Science
Graph Algorithms

CS 277
Brad Solomon

April 17, 2024

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Exam Information

Exam 3 (4/23 — 4/25)

Covering all material up to last Wednesday (April 10th)

Final Exam (05/02 — 05/06)

Expected time: 1 hour exam in 1 hour, 50 minute time block

50 minute makeup exams *during* final exam time!

Submit topics or concepts you want reviewed

Google form linked through Prairielearn

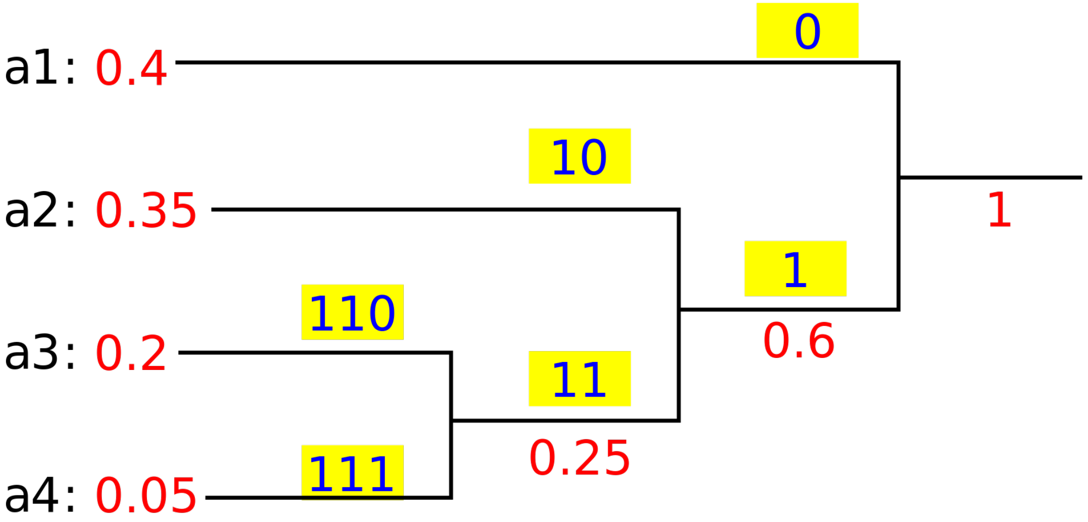
Learning Objectives

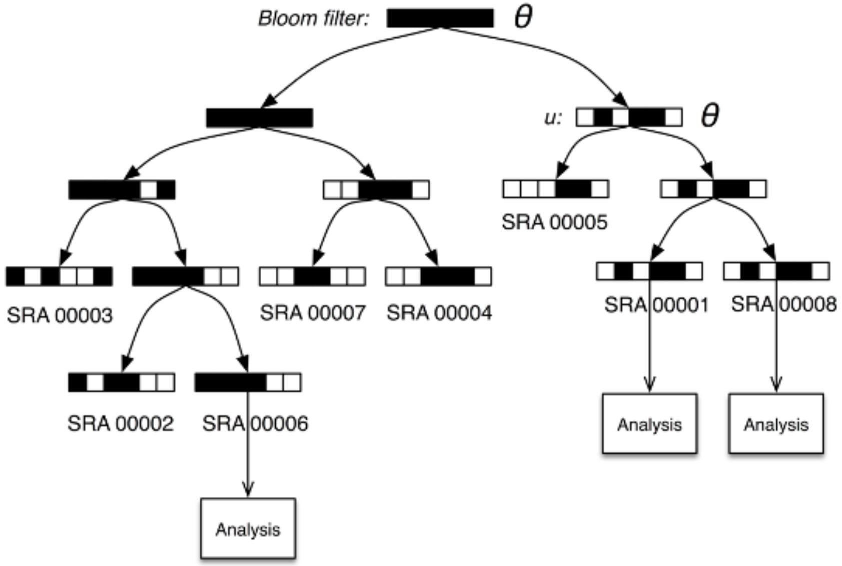
Review fundamentals of trees and graphs

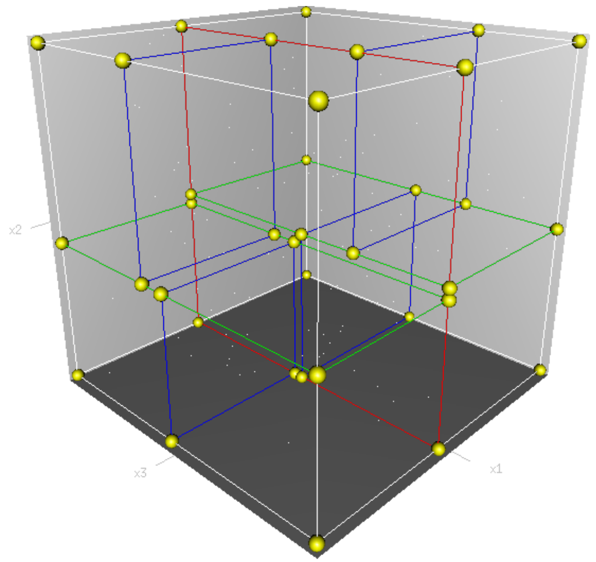
Introduce graph algorithms for SSSP and MST

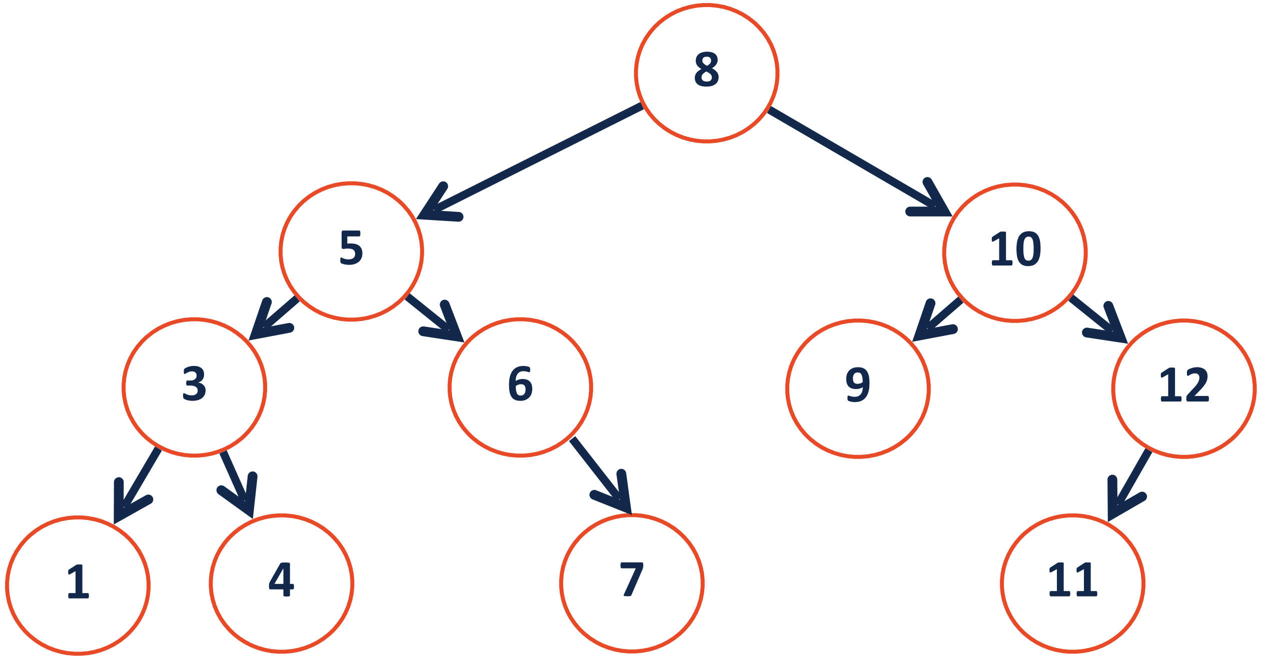
Practice analyzing the Big O of advanced algorithms

There are many *types* of trees









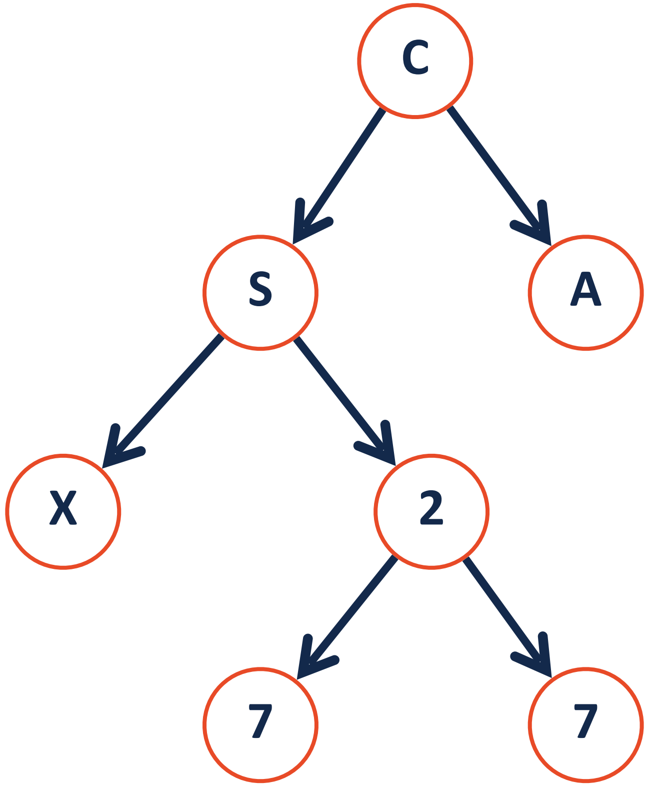
(Binary) Tree Recursion

A **binary tree** is a tree T such that:

$$T = None$$

or

$$T = treeNode(val, T_L, T_R)$$

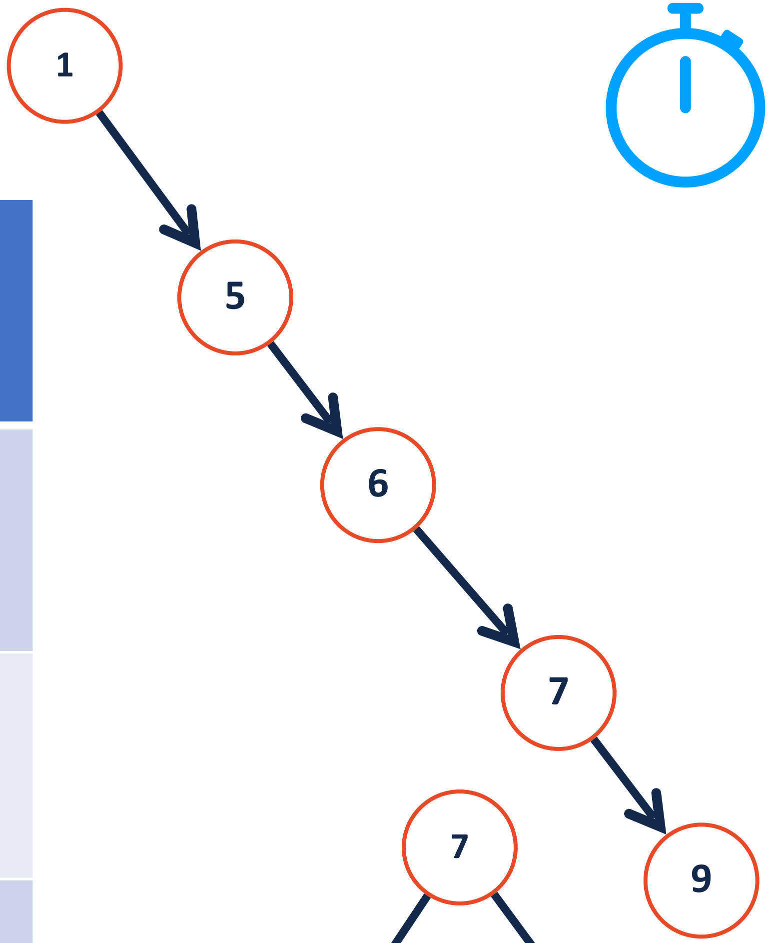


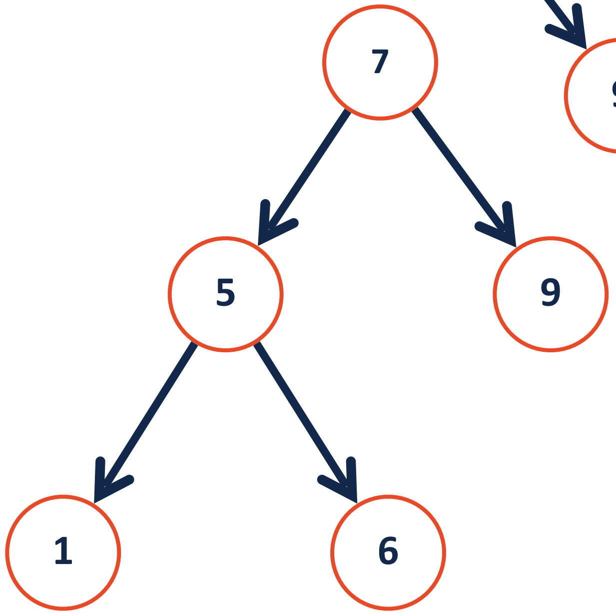

```
class treeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
class binaryTree:
    def __init__(self):
        self.root = None
```

Tree Efficiency

	BST	AVL Tree
find		
insert		
delete		
traverse		





Graph ADT

Find

getVertices() — return the list of vertices in a graph

getEdges(v) — return the list of edges that touch the vertex v

areAdjacent(u, v) — returns a bool based on if an edge from u to v exists

Insert

insertVertex(v) — adds a vertex to the graph

insertEdge(u, v) — adds an edge to the graph

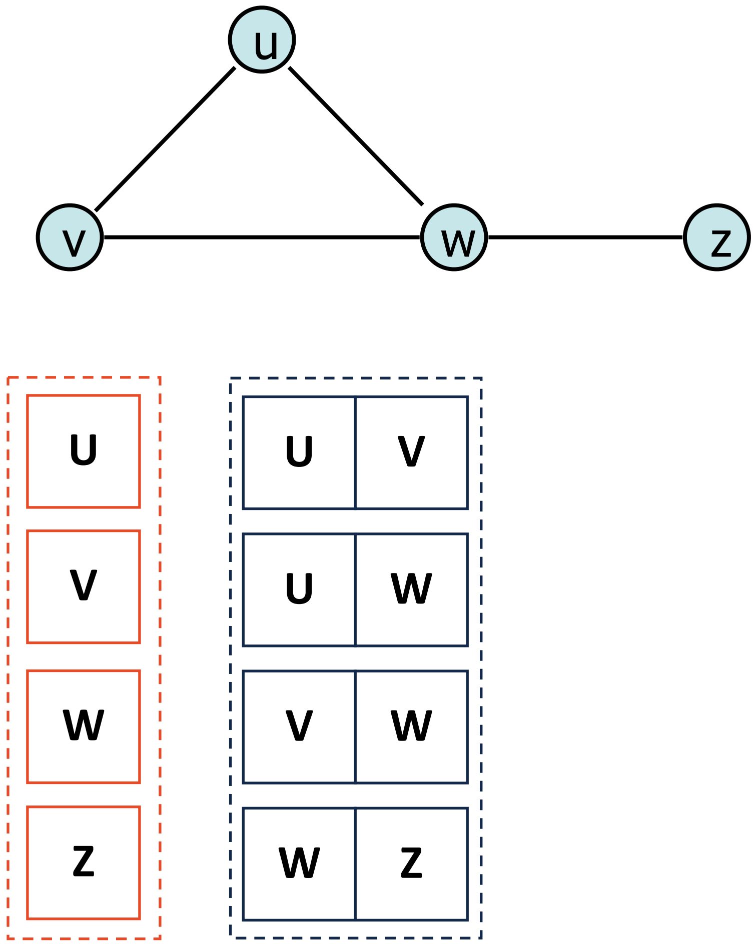
Remove

removeVertex(v) — removes a vertex from the graph

removeEdge(u, v) — removes an edge from the graph

Graph Implementation: Edge List

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



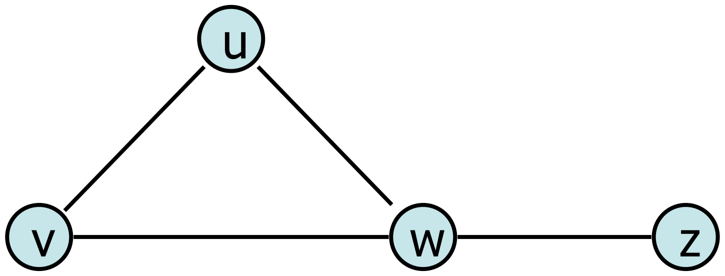
Vertex Storage:

An unordered list of vertices

Edge Storage:

An unordered list of edges

Graph Implementation: Adjacency Matrix



Vertex Storage:

A dictionary of vertices storing index in matrix

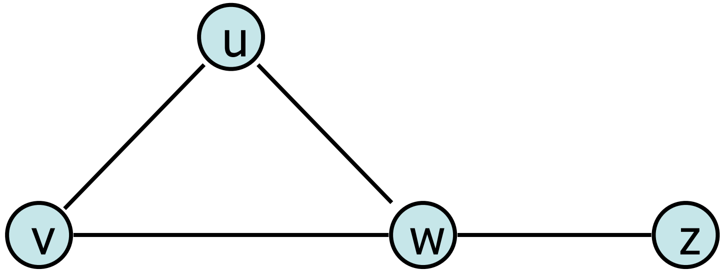
Edge Storage:

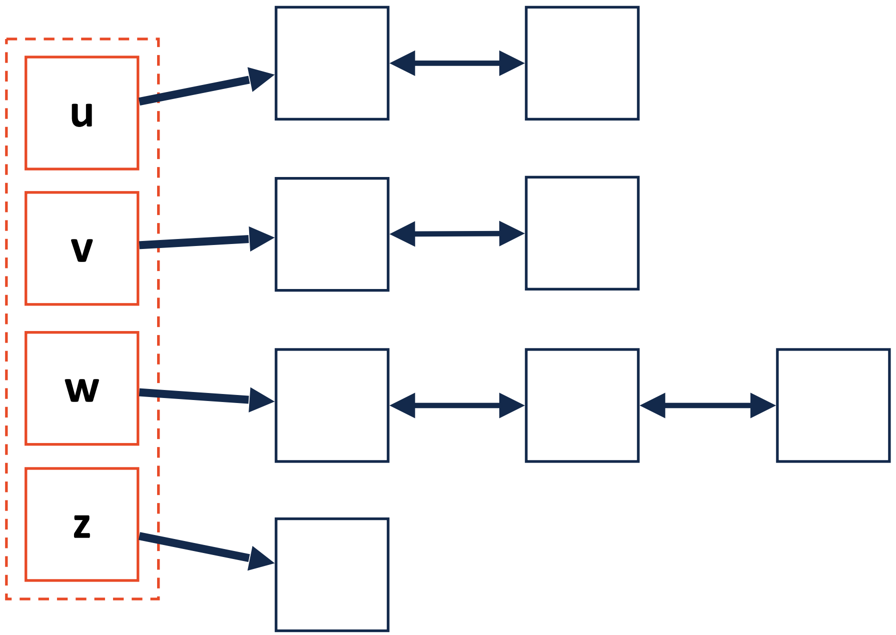
A matrix storing presence or absence of edges

U	0
V	1
W	2
Z	3

	u	v	w	z
u	0	1	1	0
v	1	0	1	0
w	1	1	0	1
z	0	0	1	0

Adjacency List





Vertex Storage:

The keys of the edge list dictionary

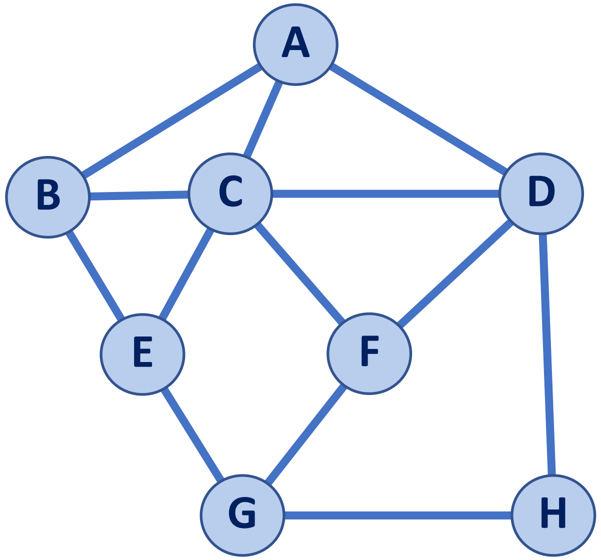
Edge Storage:

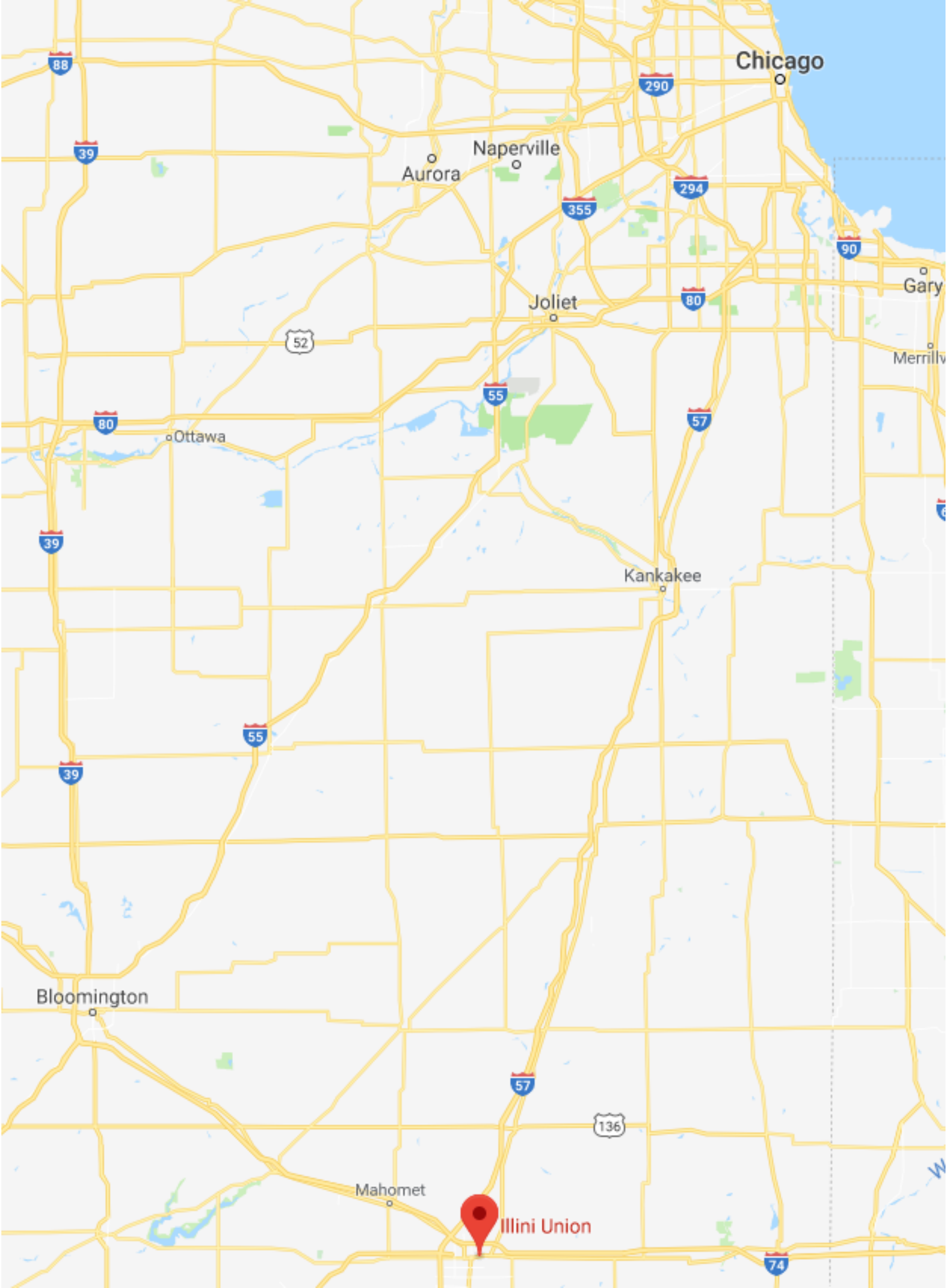
A dictionary storing endpoint vertices

|V|= n, |E|= m

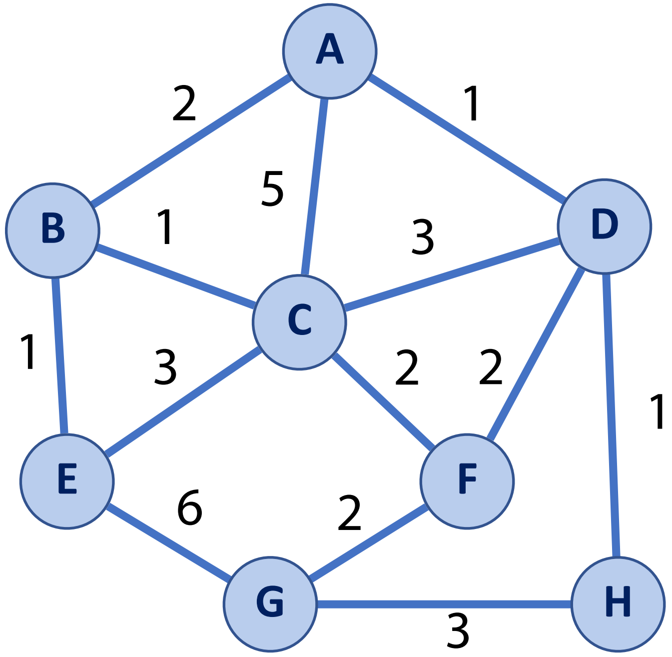
Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1*	n*	1*
removeVertex(v)	n+m	n*	deg(v)
insertEdge(u, v)	1	1	1*
removeEdge(u, v)	m	1	min(deg(u), deg(v))
getEdges(v)	m	n	deg(v)
areAdjacent(u, v)	m	1	min(deg(u), deg(v))

Shortest Path





Dijkstras Shortest Path (Distances)

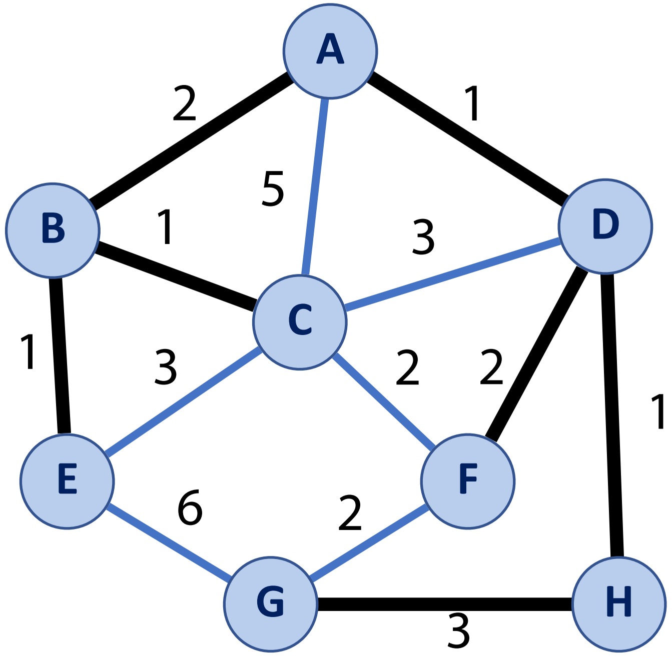


1) Given a start vertex, initialize algorithm:
   - Each vertex has previous and distance
   - Set all distances to ∞ (and source to 0)
2) While there exists an unvisited vertex:
   - Visit the current nearest vertex
   - Update distances based on current edges

Dist **A:** **B:** **C:** **D:** **E:** **F:** **G:** **H:**

Prev

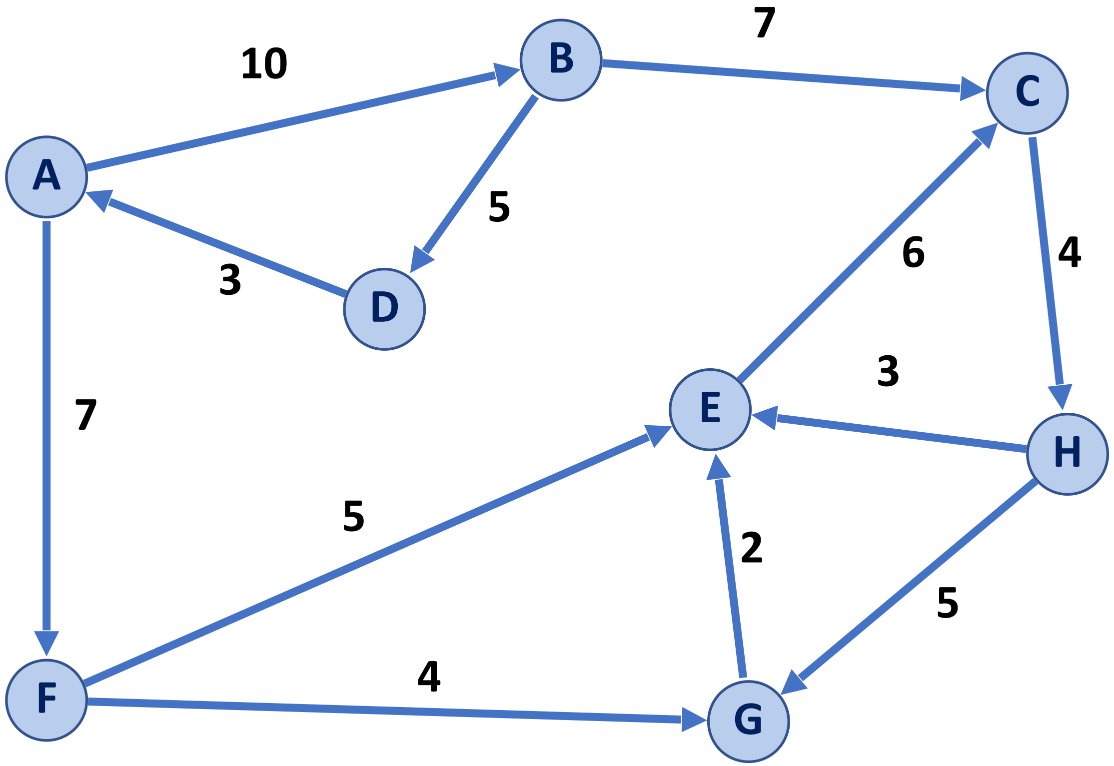
Dijkstras Shortest Path (Distances)



1) Given a start vertex, initialize algorithm:
   - Each vertex has previous and distance
   - Set all distances to ∞ (and source to 0)
2) While there exists an unvisited vertex:
   - Visit the current nearest vertex
   - Update distances based on current edges

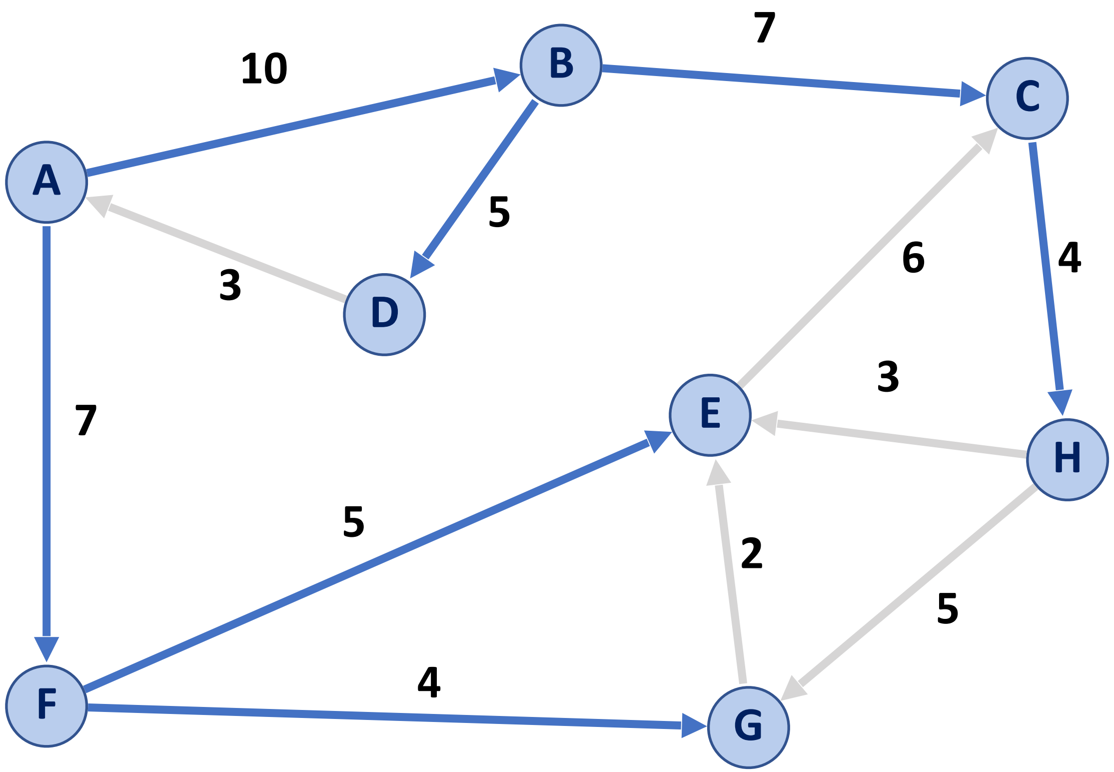
Dist	**A:** 0	**B:** 2	**C:** 3	**D:** 1	**E:** 3	**F:** 3	**G:** 5	**H:** 2
Prev	None	A	B	A	B	D	H	G

Dijkstras Shortest Path (Full Paths)



Vertex	Distance	Previous
A		
B		
C		
D		
E		
F		
G		
H		

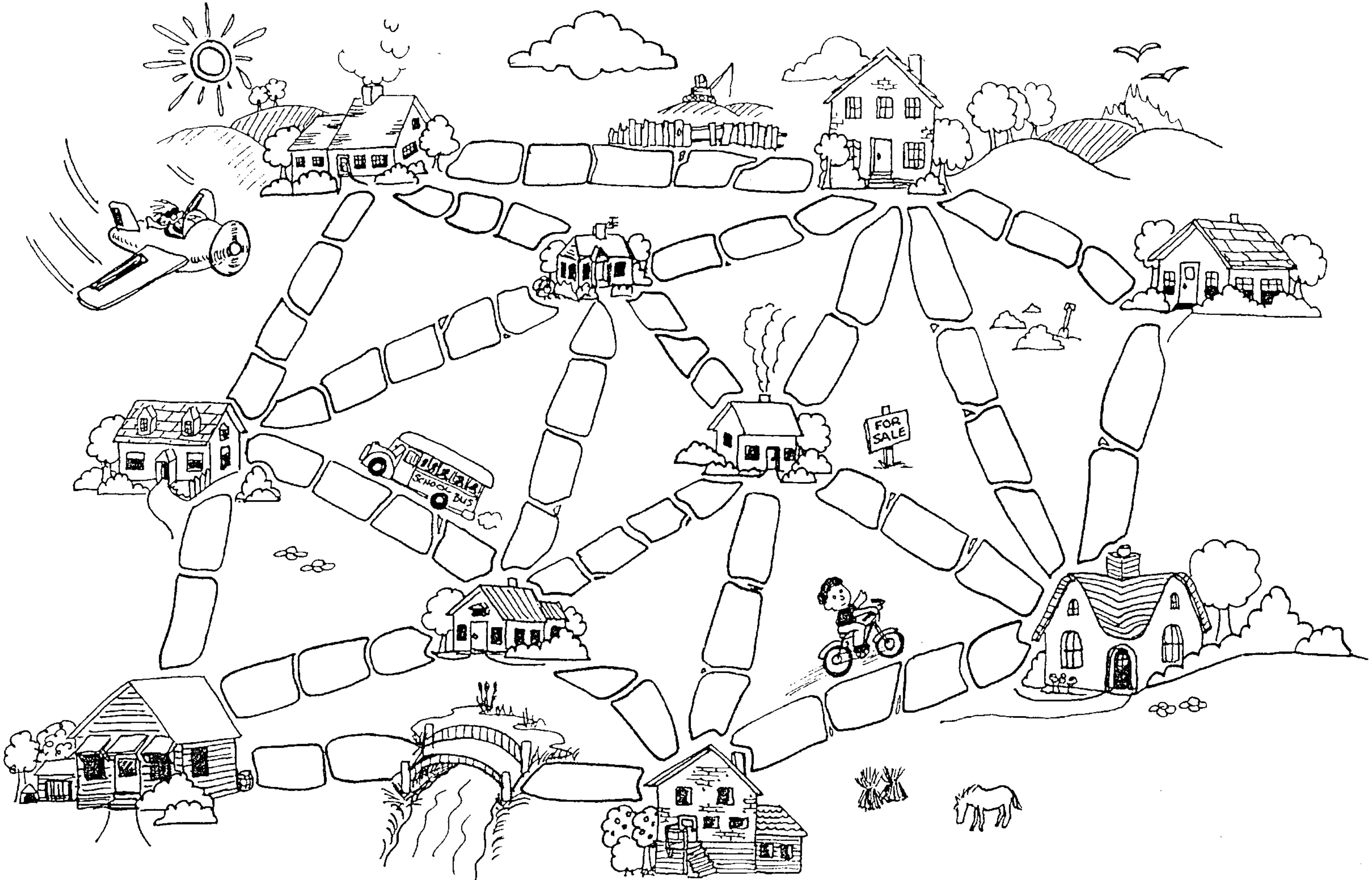
Dijkstras Shortest Path (Full Paths)



Vertex	Distance	Previous
A	0	None
B	10	A
C	17	B
D	15	B
E	12	F
F	7	A
G	11	F
H	21	C

Dijkstras in NetworkX

```
nx.shortest_path(G, source, target)
```


FOR SALE
SCHOOL BUS

Minimum Spanning Tree

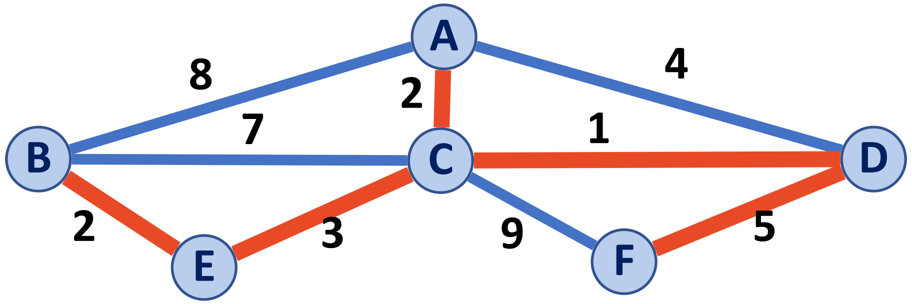
Input: Connected, undirected graph G with positive edge weights

Output: A graph G' with the following properties:

G' is a **spanning graph** of G — all vertices are connected and included

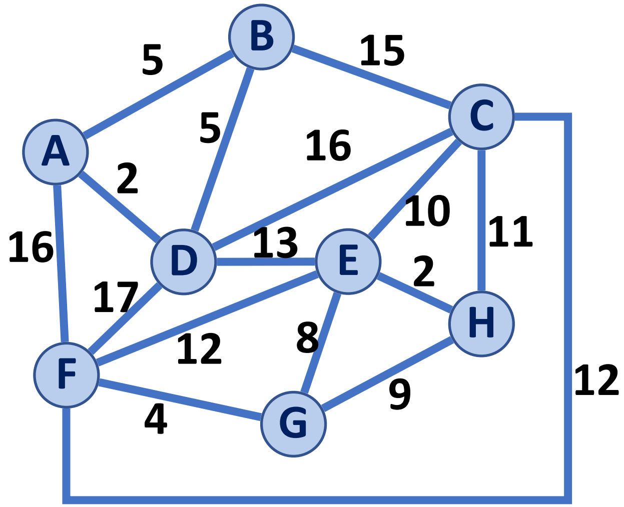
G' is acyclic

G' has a minimal total weight among all possible spanning trees

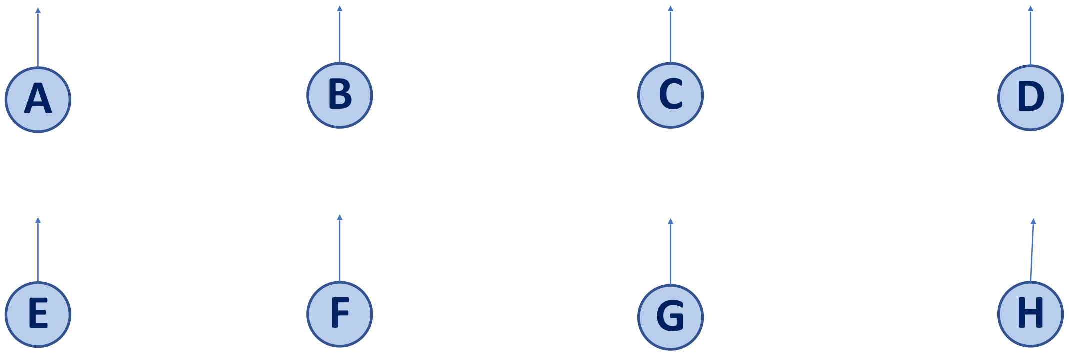


Kruskal's Algorithm

(A, D, 2)
(E, H, 2)
(F, G, 4)
(A, B, 5)
(B, D, 5)
(G, E, 8)
(G, H, 9)
(E, C, 10)
(C, H, 11)
(E, F, 12)
(F, C, 12)
(D, E, 13)
(B, C, 15)
(C, D, 16)
(A, F, 16)
(D, F, 17)



1. Initialize sorted edge list and empty MST
2. Set each vertex as its own partition
3. Find the minimum edge connecting two partitions, add it to MST
4. Merge the two partitions
5. Repeat steps 3-4 until |V| - 1 edges found



Kruskal's Algorithm

1. Initialize sorted edge list and empty MST

2. Set each vertex as its own partition

3. Find the min edge between two partitions and add it to MST

4. Merge the two partitions

5. Repeat steps 3-4 until |V| - 1 edges found

```
def kruskal(G):
    sortedEdgeList = sortEdges(G)
    outEdges = []

    part = {}
    belong = {}
    i = 0
    for v in G.nodes():
        belong[v]=i
        part[i]=set(v)
        i+=1

    i=0
    numV = len(G.nodes())
    while len(outEdges) < numV - 1:

        u, v, w = sortedEdgeList[i]
        i+=1
        x = belong[u]
        y = belong[v]
        if x!=y:
            outEdges.append( (u, v, w))
            part[x]=part[x].union(part[y])
            for t in part[y]:
                belong[t]=x
            part.pop(y)

    return outEdges
```

Kruskal Runtime

Let $|V| = n$ and $|E| = m$

1. Initialize a partition for each vertex

2. Build a sorted array of edges

3. Get the minimum valid edge

4. Merge partitions and add to MST

```
def kruskal(G):
    sortedEdgeList = sortEdges(G)
    outEdges = []

    part = {}
    belong = {}
    i = 0
    for v in G.nodes():
        belong[v]=i
        part[i]=set(v)
        i+=1

    i=0
    numV = len(G.nodes())
    while len(outEdges) < numV - 1:

        u, v, w = sortedEdgeList[i]
        i+=1
        x = belong[u]
        y = belong[v]
        if x!=y:
            outEdges.append( (u, v, w))
            part[x]=part[x].union(part[y])
            for t in part[y]:
                belong[t]=x
            part.pop(y)

    return outEdges
```

Kruskal Runtime

Let $|V| = n$ and $|E| = m$

1. Initialize a partition for each vertex

   $O(n)$

2. Build a sorted array of edges

   $O(m \log m)$

3. Get the minimum valid edge

   $O(1)$

4. Merge partitions and add to MST

   $O(m)$ **total****

```
def kruskal(G):
    sortedEdgeList = sortEdges(G)
    outEdges = []

    part = {}
    belong = {}
    i = 0
    for v in G.nodes():
        belong[v]=i
        part[i]=set(v)
        i+=1

    i=0
    numV = len(G.nodes())
    while len(outEdges) < numV - 1:

        u, v, w = sortedEdgeList[i]
        i+=1
        x = belong[u]
        y = belong[v]
        if x!=y:
            outEdges.append( (u, v, w))
            part[x]=part[x].union(part[y])
            for t in part[y]:
                belong[t]=x
            part.pop(y)

    return outEdges
```

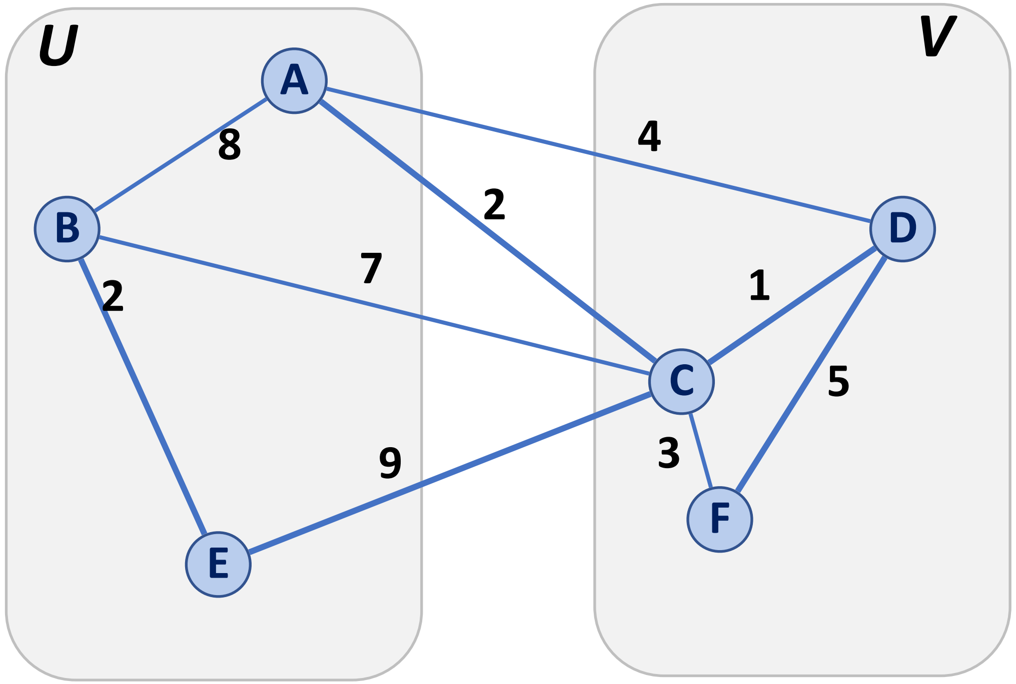
MST in NetworkX

```
nx.minimum_spanning_tree(G, weight, algorithm=‘kruskal’)
```

Partition Property

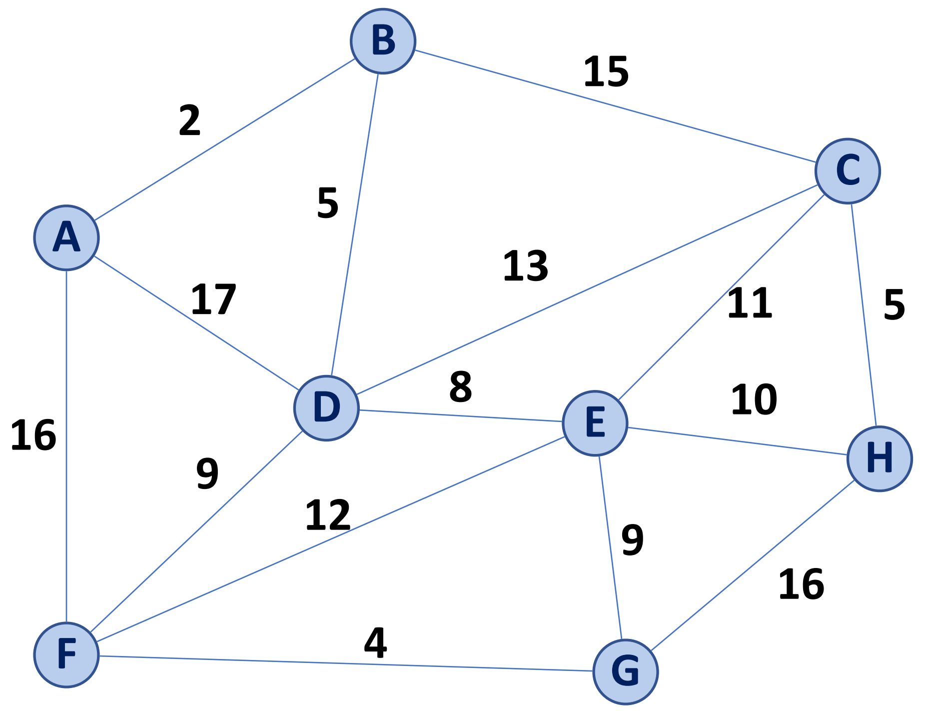
Consider an arbitrary partition of the vertices on **G** into two subsets **U** and **V**

If **e** is an edge of minimum weight across the partition, then there exists a minimum spanning tree containing **e**

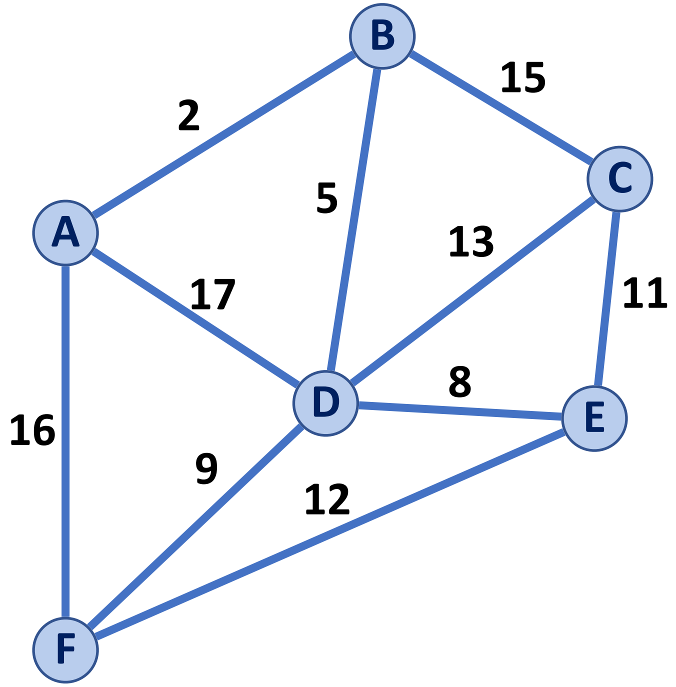


Partition Property

The partition property suggests an algorithm for MST:



Prim's Algorithm



1) Create data structs to store distances

   Create data structs to store previous

2) Initialize all distances to ∞ (and source to 0)

3) Find the **min edge** between 'in' and 'out'

   Add the vertex this edge links to to 'in'

   Update distances between 'in' and 'out'

   (The updated distances are **single edges**)

4) Repeat (3) once for every vertex

Dist	**A:**	**B:**	**C:**	**D:**	**E:**	**F:**
Prev						

Prim’s Runtime

Let $|V| = n$ and $|E| = m$

1. Initialize distances

2. Get min valid edge

3. Add edge to MST

4. Update all distances

```
def prim(G, start):
    outEdges=[]
    dist = {}
    prev = {}
    inGroup = set()

    for v in G.nodes():
        dist[v] = float("inf")
    dist[start]=0
    prev[start]=None

    for _ in range(len(G.nodes()):
        v = minOutEdge(dist, inGroup)
        inGroup.add(v)
        if prev[v]!=None:
            outEdges.append( (prev[v], v, dist[v]) )

        for u in nx.neighbors(G, v):
            weight = G[u][v][‘weight’]
            if u not in inGroup and dist[u] > weight:
                dist[u]=weight
                prev[u]=v
    return outEdges

```

Prim’s Runtime

Let $|V| = n$ and $|E| = m$

1. Initialize distances

   $O(n)$

2. Get min valid edge

   $O(n)$

3. Add edge to MST

   $O(1)$

4. Update all distances

   $O(n)$

(steps 2–4) x $O(n)$

```
def prim(G, start):
    outEdges=[]
    dist = {}
    prev = {}
    inGroup = set()

    for v in G.nodes():
        dist[v] = float("inf")
    dist[start]=0
    prev[start]=None

    for _ in range(len(G.nodes()):
        v = minOutEdge(dist, inGroup)
        inGroup.add(v)
        if prev[v]!=None:
            outEdges.append( (prev[v], v, dist[v]) )

        for u in nx.neighbors(G, v):
            weight = G[u][v][‘weight’]
            if u not in inGroup and dist[u] > weight:
                dist[u]=weight
                prev[u]=v
    return outEdges

```

MST Algorithm Runtimes

Kruskal's Algorithm:

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

Prim's Algorithm:

$O(n^2)$

How does n and m relate (assuming graph is connected)?

MST Algorithm Runtimes

Kruskal's Algorithm:

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

Prim's Algorithm:

$O(n^2)$

Sparse Graph ($m \approx n$):

Dense Graph ($m \approx n^2$):

Fibonacci Heap MST Runtimes

Kruskal's Algorithm:

$O(m \log n)$

Prim's Algorithm:

$O(n \log n + m)$

Sparse Graph ($m \approx n$):

Dense Graph ($m \approx n^2$):

Final Takeaway: Memorizing Big O is not as important as understanding ***why***