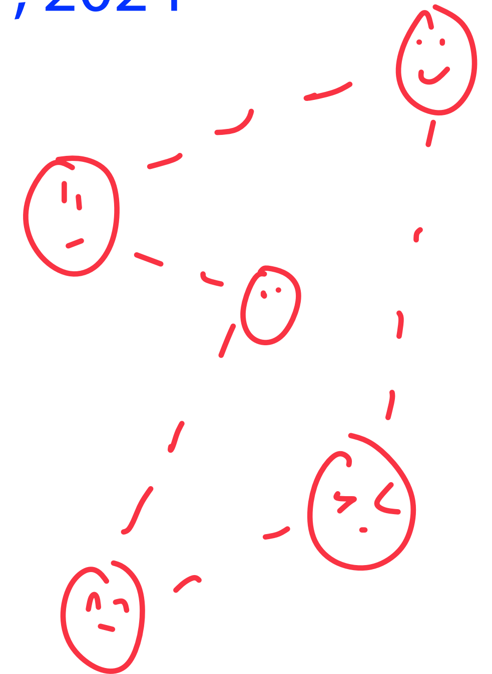
Algorithms and Data Structures for Data Science

Graph Algorithms

A tiny review of trees and...

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

AVL trees

Graphs

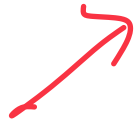
Recursion

Coding practice

Learning Objectives

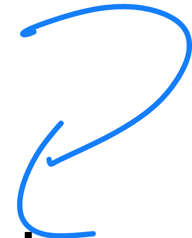
AVL tree
Huffman
KD tree

Outside of scope!
Red-Black tree



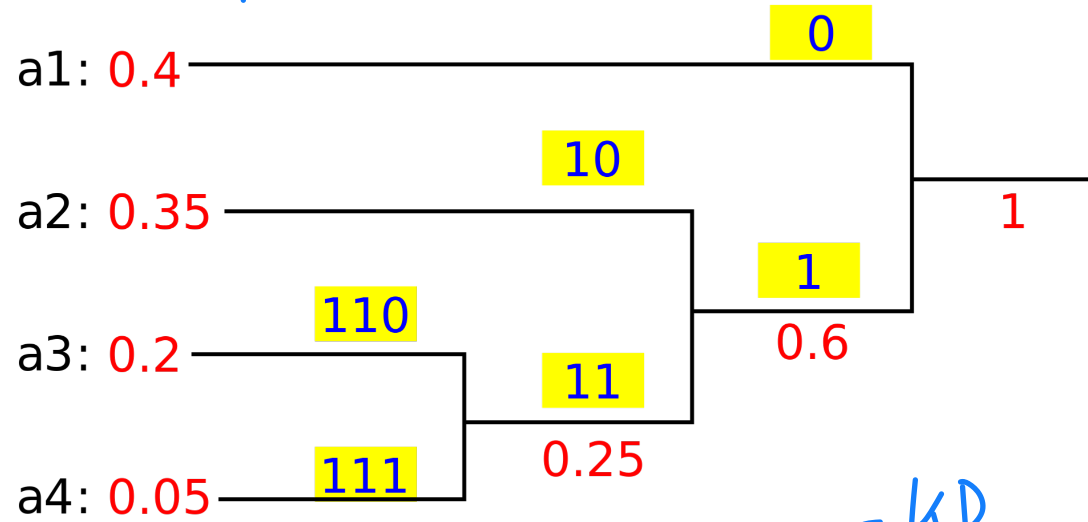
Review fundamentals of trees and graphs

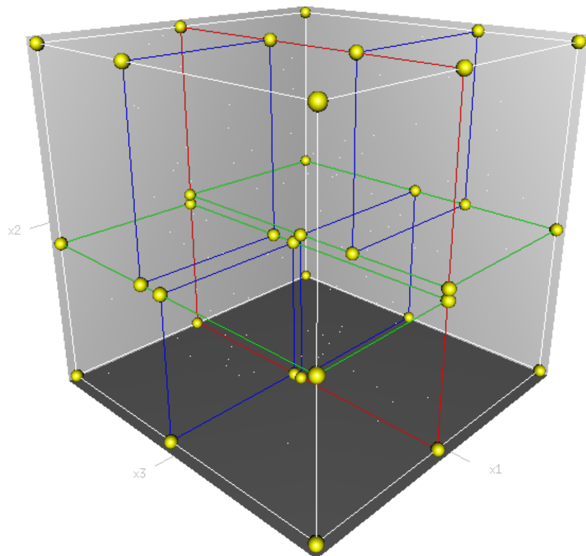
Introduce graph algorithms for SSSP and MST

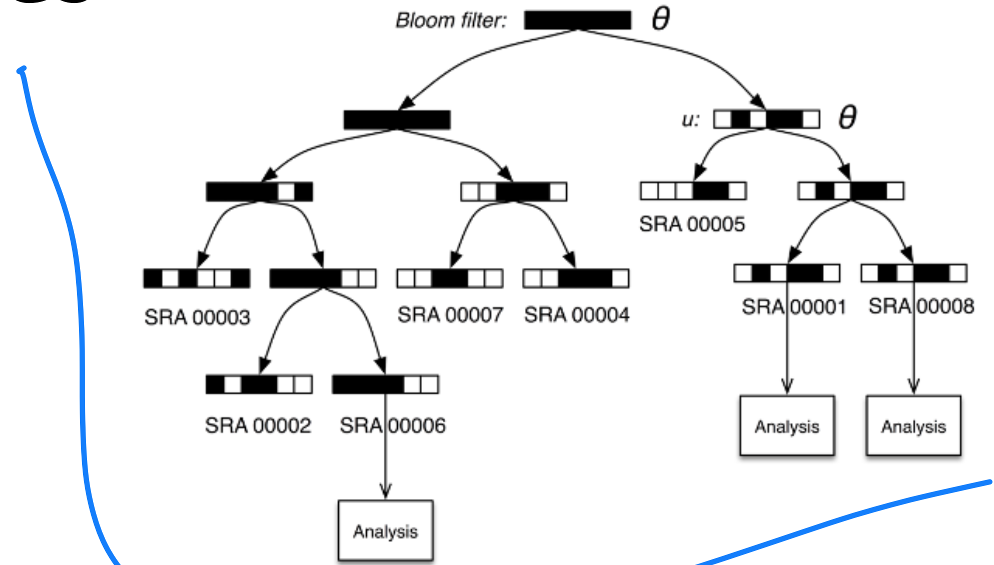


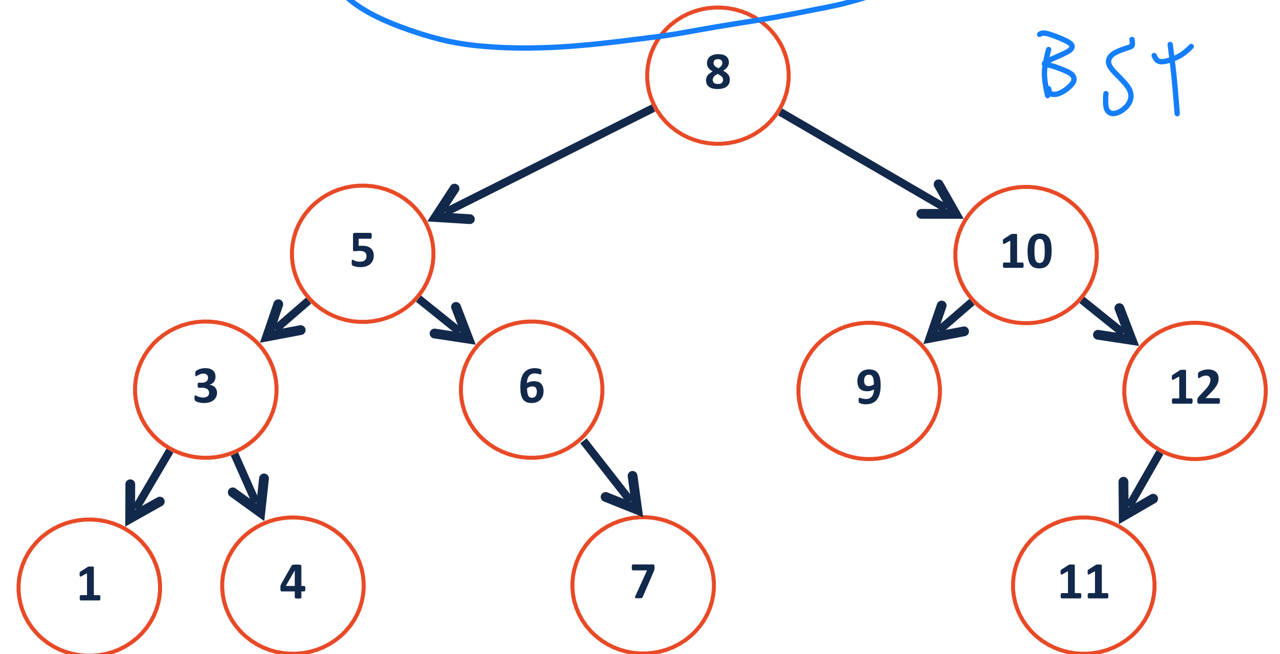
Practice analyzing the Big O of advanced algorithms

There are many *types* of trees

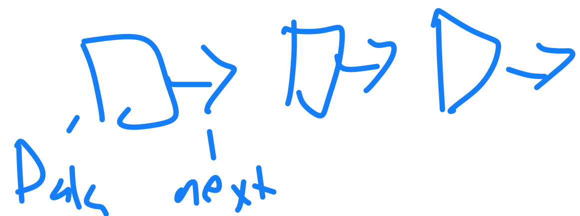








(Binary) Tree Recursion

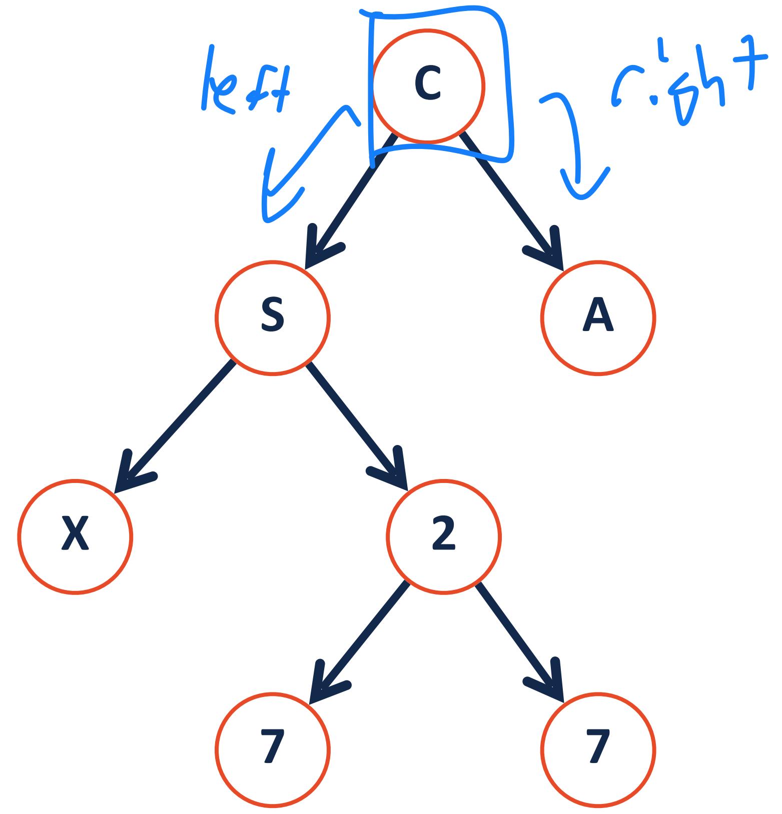


A **binary tree** is a tree T such that:

$$T = None$$

or

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




```
1 class treeNode:
2     def __init__(self, val, left=None, right=None):
3         self.val = val
4         self.left = left
5         self.right = right
```

```
1 class binaryTree:
2     def __init__(self):
3         self.root = None
4
5
```

Binary Search Tree (BST)

find(40)

A **BST** is a binary tree $T = treeNode(val, T_L, T_r)$ such that:

ordered

$$\forall n \in T_L,\ n\,.\,val < T\,.\,val$$

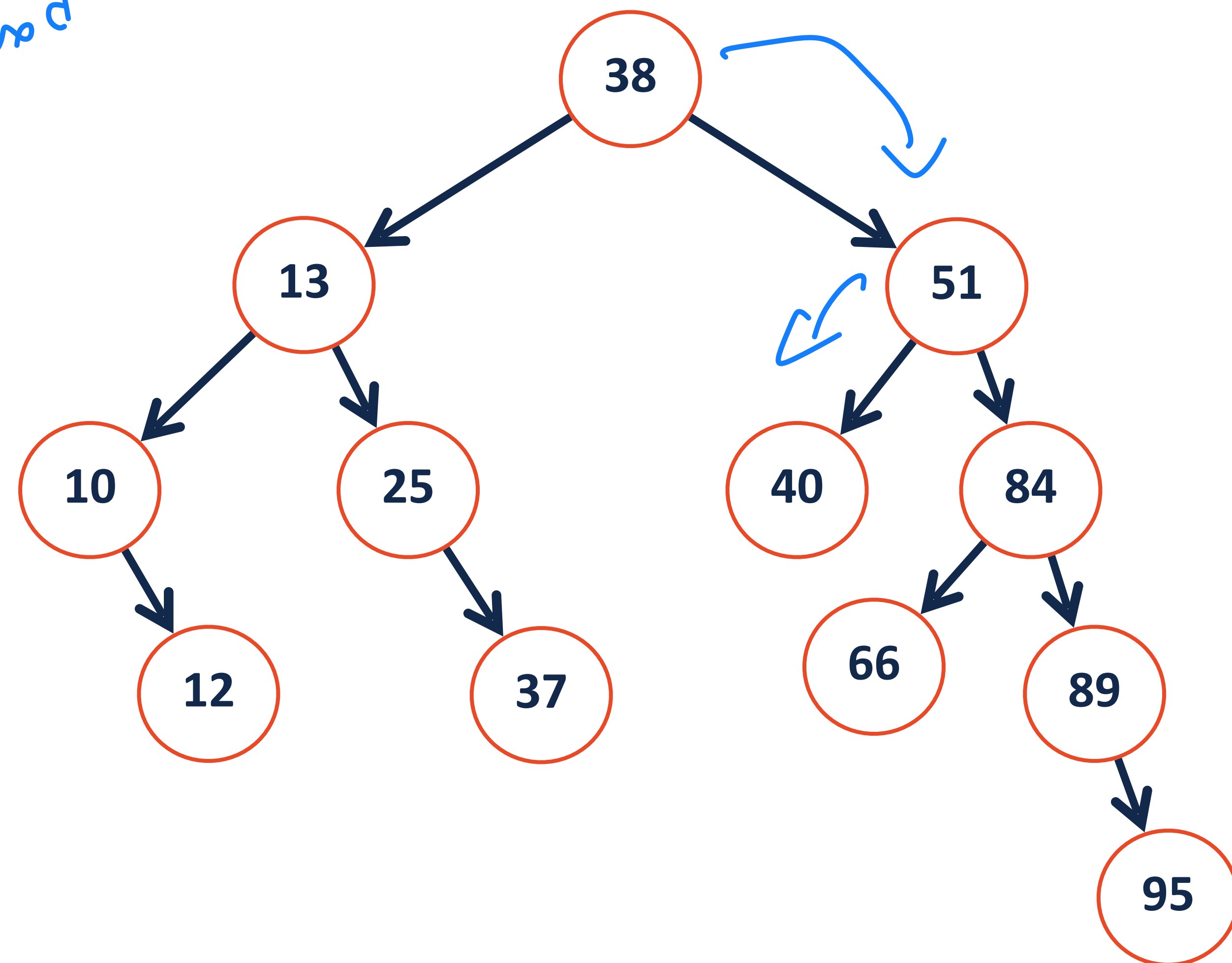
All values to left are

Smaller

$$\forall n \in T_R,\ n\,.\,val > T\,.\,val$$

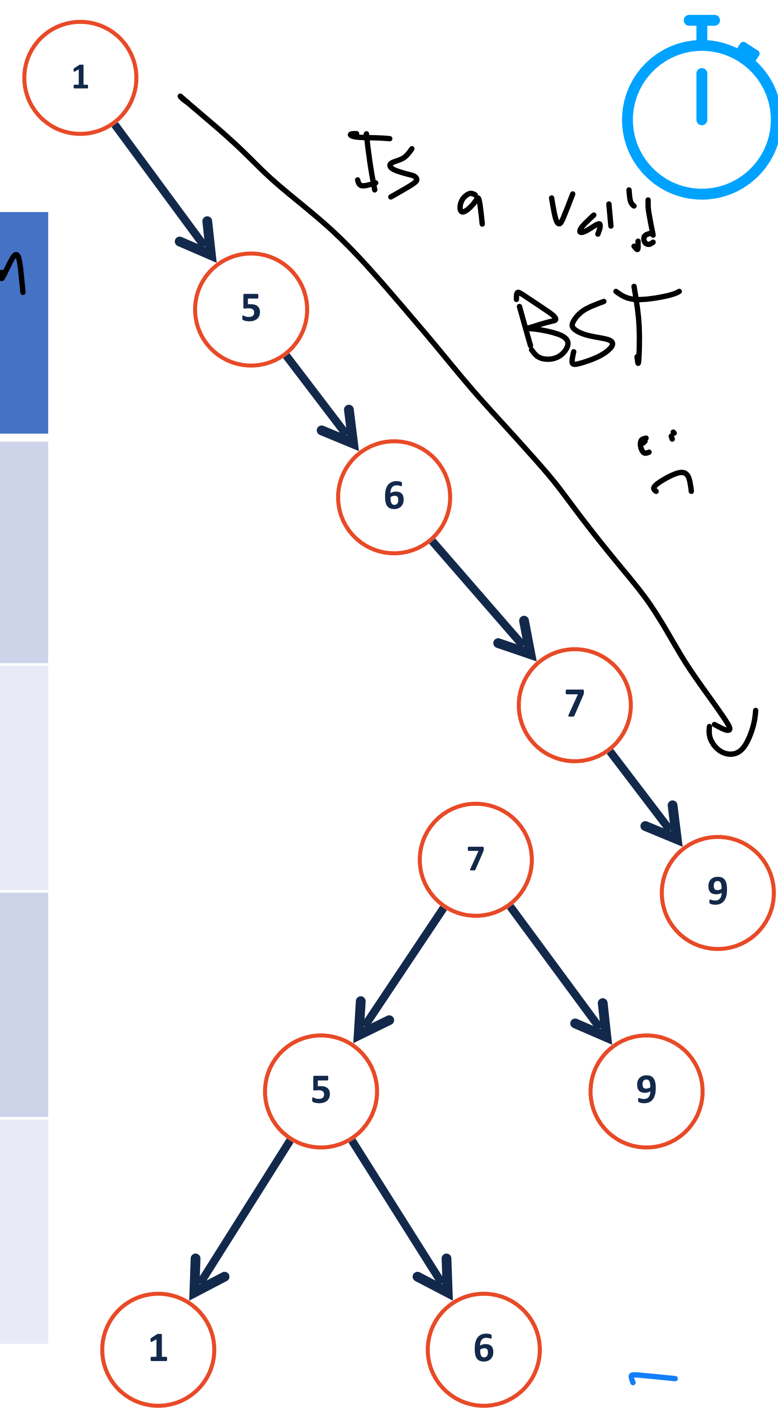
Right

Larger



Tree Efficiency

	worst case h = n **BST**	worst case h = log n **AVL Tree**
find	O(n) O(h) →	O(log n)
insert	O(n) →	O(log n)
delete	O(n) →	O(log n)
traverse	O(n)	O(n)



Graph ADT

has nodes = vertices and edges

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

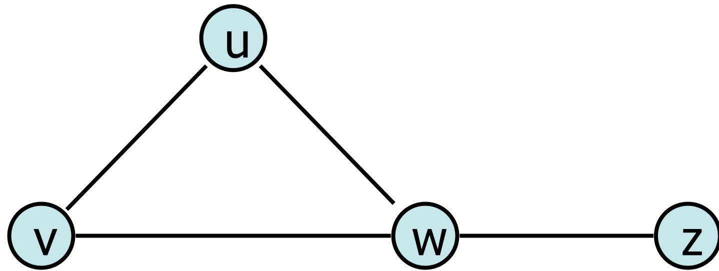
Remove

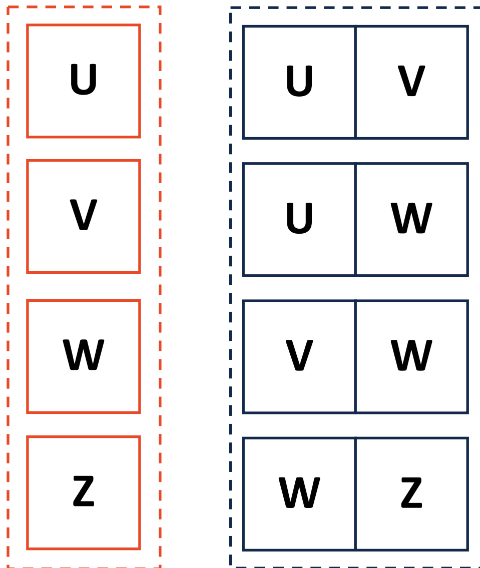
removeVertex(v) — removes a vertex from the graph

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

Graph Implementation: Edge List

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





Vertex Storage:

An unordered list of vertices

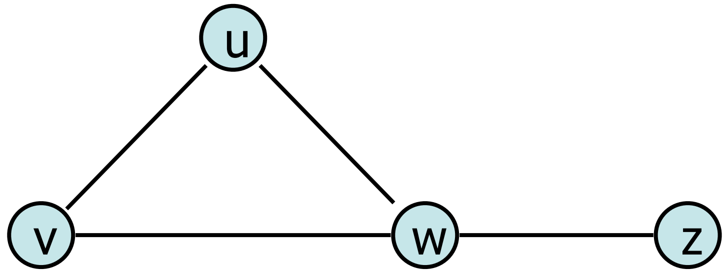
Edge Storage:

An unordered list of edges

+ Very efficient in space costs

+ Easy to add edges or vertices

Graph Implementation: Adjacency Matrix



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

Vertex Storage:

A dictionary of vertices storing index in matrix

Edge Storage:

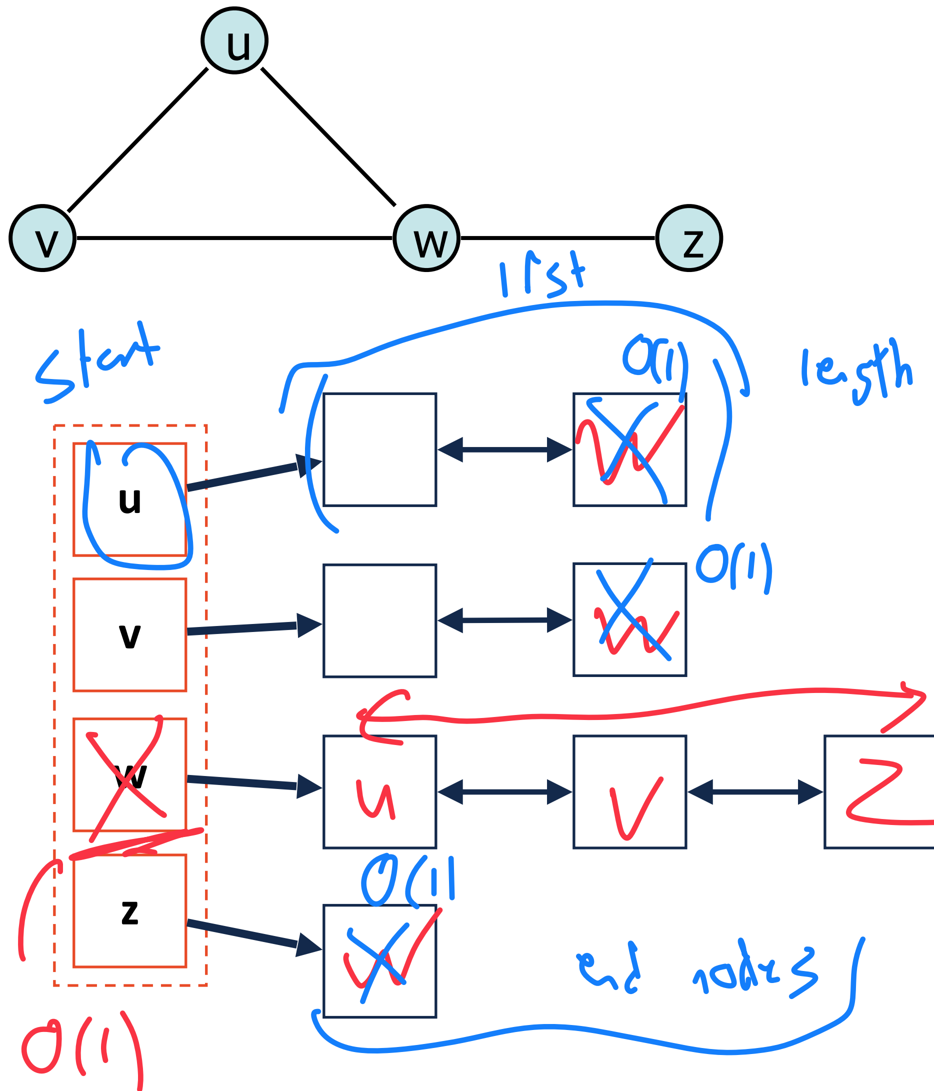
A matrix storing presence or absence of edges

+ Optimal way to detect if two vertices are neighbours

− Not space efficient (N^2 values always)

− Not easy to add vertices or remove them

Adjacency List



Vertex Storage:

The keys of the edge list dictionary

Edge Storage:

A dictionary storing endpoint vertices

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

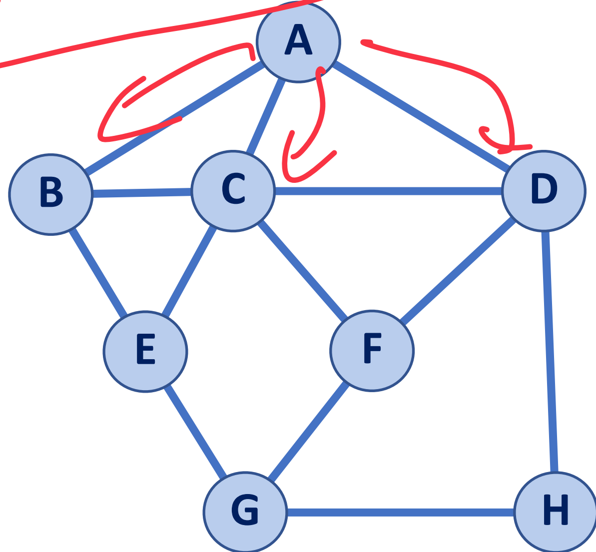
Expressed as O(f)	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n²	n+m
insertVertex(v)	1*	n*	1*
removeVertex(v)	n+m	n*	deg(v)
insertEdge(u, v)	1	1	1*
removeEdge(u, v)	m	1	min(deg(u), deg(v))
getEdges(v)	m	n	deg(v)
areAdjacent(u, v)	m	1	min(deg(u), deg(v))

Stores only edges that exist

Shortest Path

Hard:

simplest version:



breadth first traversal

finds shortest path from source to everything

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

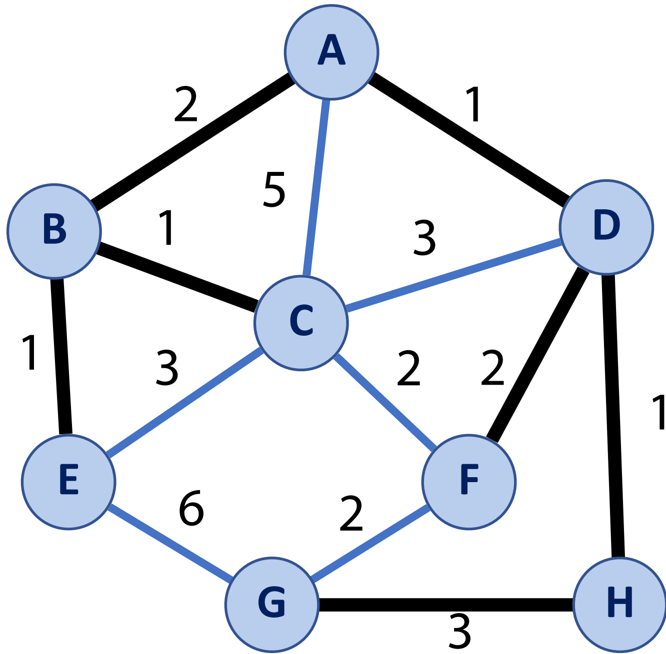
Most important piece

2+1= 3

1+3=4

	A:	B:	C:	D:	E:	F:	G:	H:
Dist	0	∞ 2	∞ 5 2	∞ 1	∞ 3	∞ 3	∞ 5	∞ 4 2
Prev	None	A	A D B	A	B	D	H	D

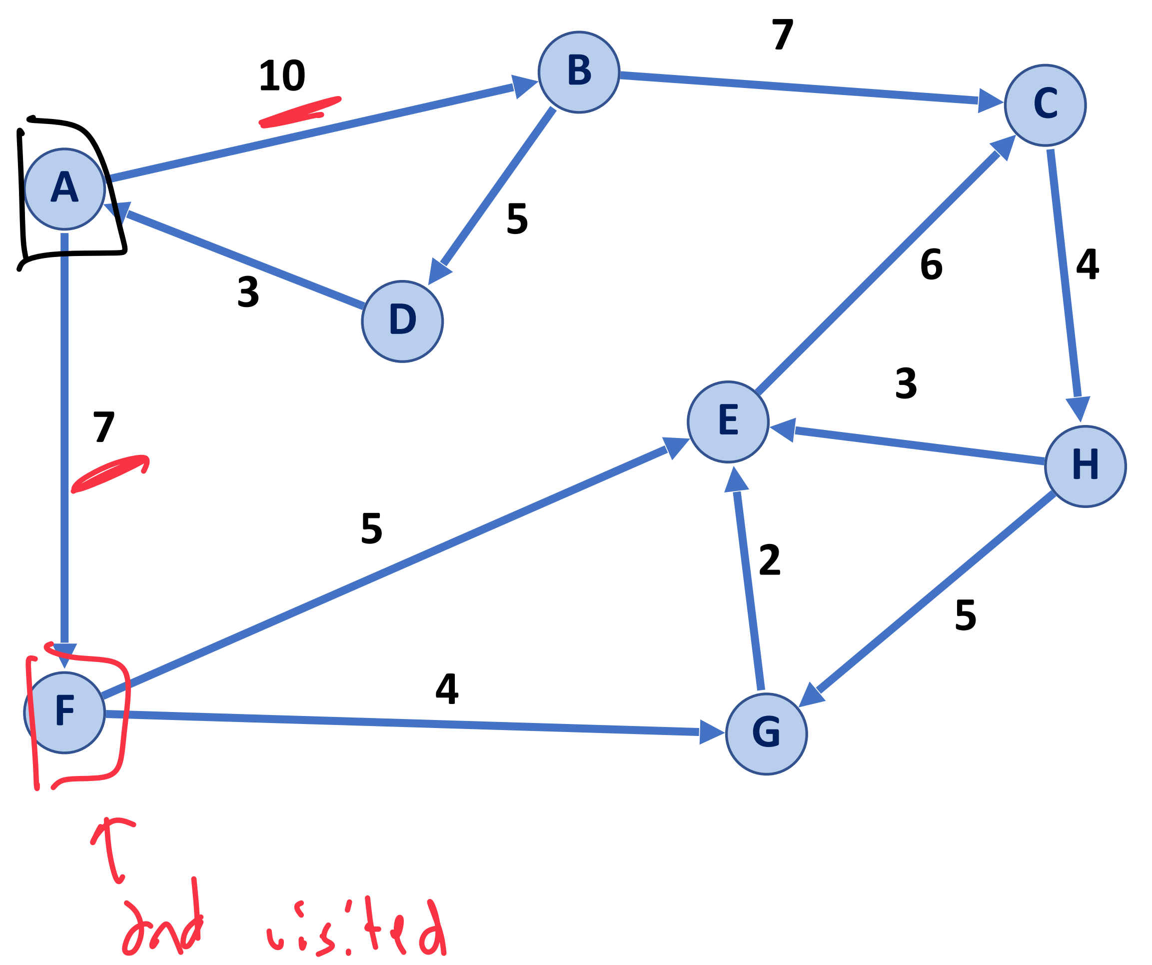
Dijkstras Shortest Path (Distances)



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

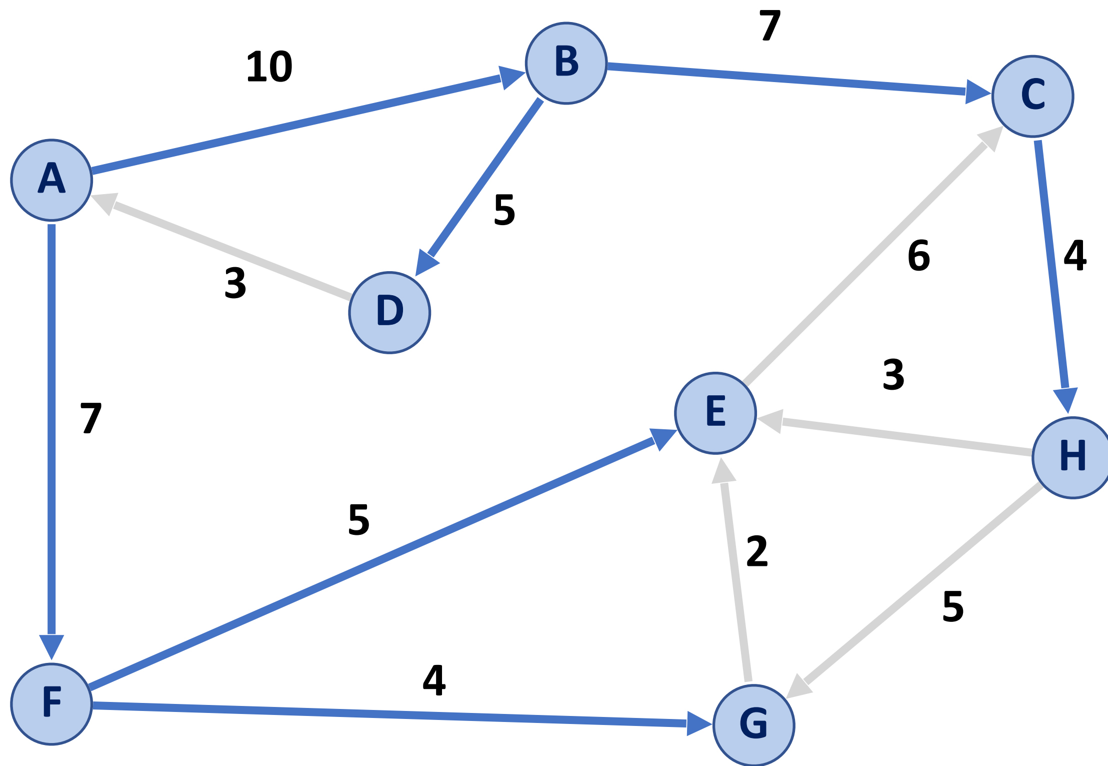
Dist	**A:** 0	**B:** 2	**C:** 3	**D:** 1	**E:** 3	**F:** 3	**G:** 5	**H:** 2
Prev	None	A	B	A	B	D	H	G

Dijkstras Shortest Path (Full Paths)



Vertex	Distance	Previous
A	0	None
B	~~∞~~ 10	A
C	∞	
D	∞	
E	∞	
F	~~∞~~ 7	A
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

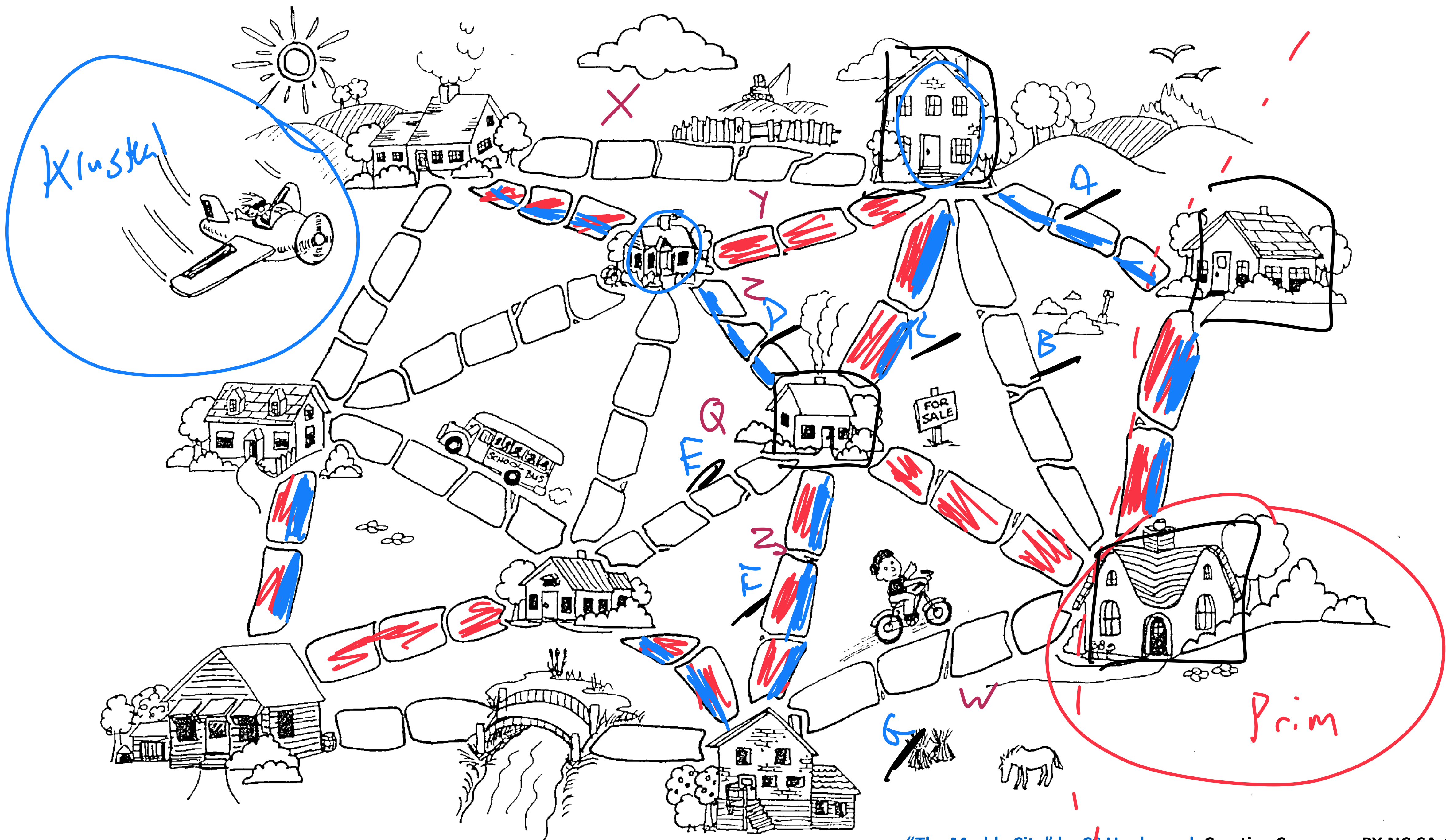
Dijkstras in NetworkX

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

A X

(G, source)

↳ Path for everything as a dictionary!



"The Muddy City" by CS Unplugged, Creative Commons BY-NC-SA 4.0

Minimum Spanning Tree

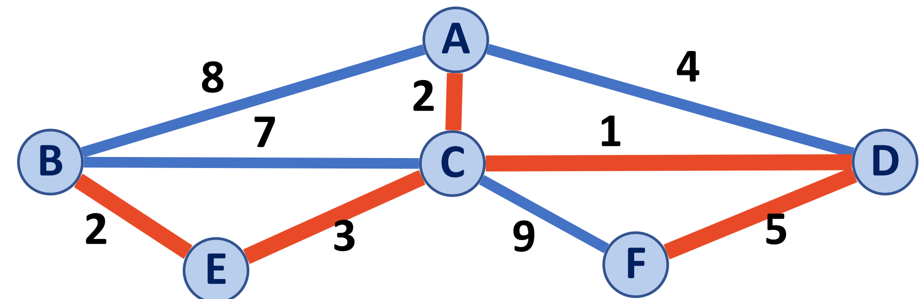
Input: Connected, undirected graph G with positive edge weights

Output: A graph G' with the following properties:

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

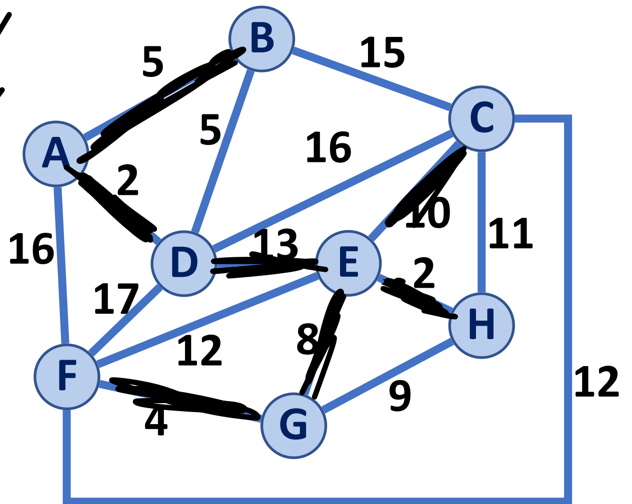
G' is acyclic

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

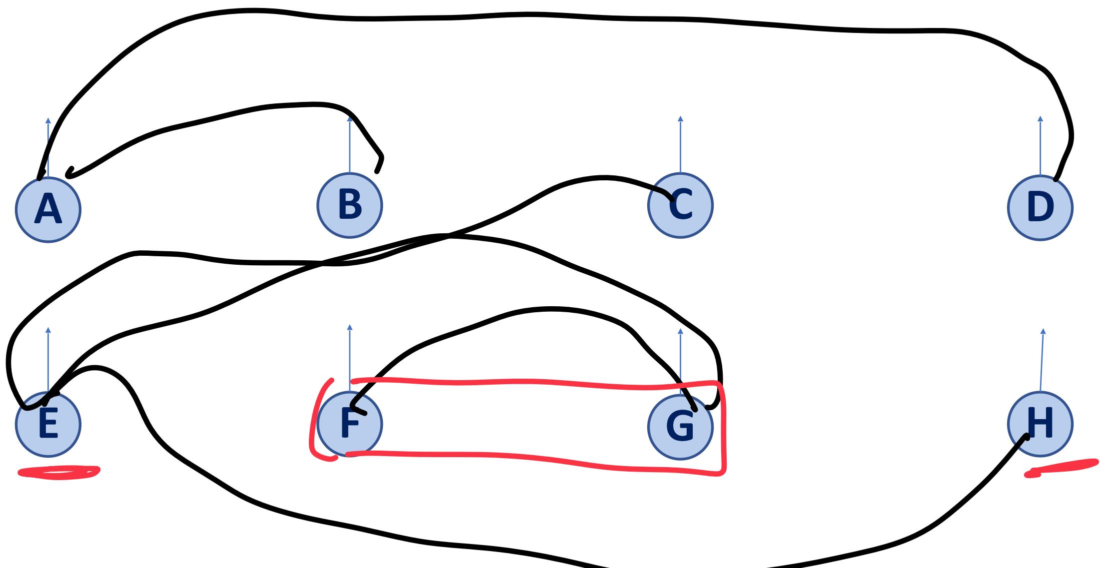


Kruskal's Algorithm

- (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



F = E's partition #

G = E's partition #

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

```
1   def kruskal(G):
2       sortedEdgeList = sortEdges(G)
3       outEdges = []
4   
5       part = {}
6       belong = {}
7       i = 0
8       for v in G.nodes():
9           belong[v]=i
10          part[i]=set(v)
11          i+=1
12  
13      i=0
14      numV = len(G.nodes())
15      while len(outEdges) < numV - 1:
16  
17          u, v, w = sortedEdgeList[i]
18          i+=1
19          x = belong[u]
20          y = belong[v]
21          if x!=y:
22              outEdges.append( (u, v, w))
23              part[x]=part[x].union(part[y])
24              for t in part[y]:
25                  belong[t]=x
26              part.pop(y)
27  
28      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

```
1  def kruskal(G):
2      sortedEdgeList = sortEdges(G)
3      outEdges = []
4
5      part = {}
6      belong = {}
7      i = 0
8      for v in G.nodes():
9          belong[v]=i
10         part[i]=set(v)
11         i+=1
12
13     i=0
14     numV = len(G.nodes())
15     while len(outEdges) < numV - 1:
16
17         u, v, w = sortedEdgeList[i]
18         i+=1
19         x = belong[u]
20         y = belong[v]
21         if x!=y:
22             outEdges.append( (u, v, w))
23             part[x]=part[x].union(part[y])
24             for t in part[y]:
25                 belong[t]=x
26             part.pop(y)
27
28     return outEdges
```

Kruskal Runtime

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

understand these

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**** ← Tricky!

```
1   def kruskal(G):
2       sortedEdgeList = sortEdges(G)
3       outEdges = []
4
5       part = {}
6       belong = {}
7       i = 0
8       for v in G.nodes():
9           belong[v]=i
10          part[i]=set(v)
11          i+=1
12
13      i=0
14      numV = len(G.nodes())
15      while len(outEdges) < numV - 1:
16
17          u, v, w = sortedEdgeList[i]
18          i+=1
19          x = belong[u]
20          y = belong[v]
21          if x!=y:
22              outEdges.append( (u, v, w))
23              part[x]=part[x].union(part[y])
24              for t in part[y]:
25                  belong[t]=x
26              part.pop(y)
27
28      return outEdges
```

Sort

× O(1)

MST in NetworkX

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

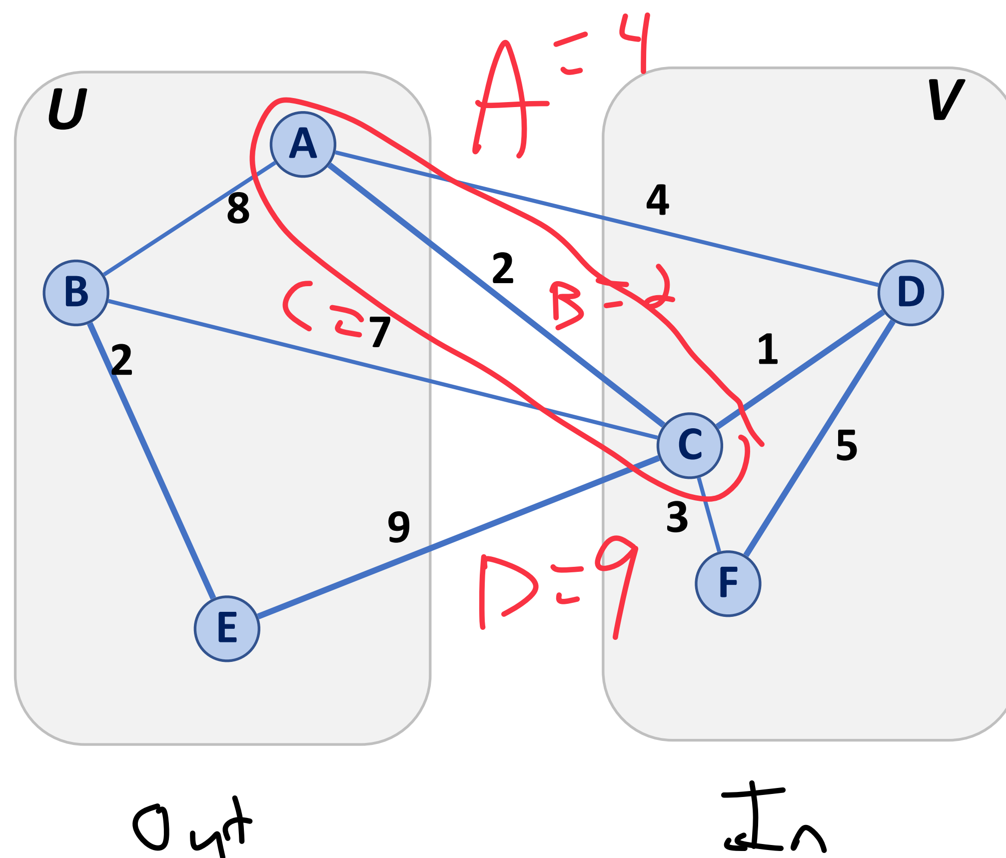
edge attribute

(Default is 'weight')

Partition Property

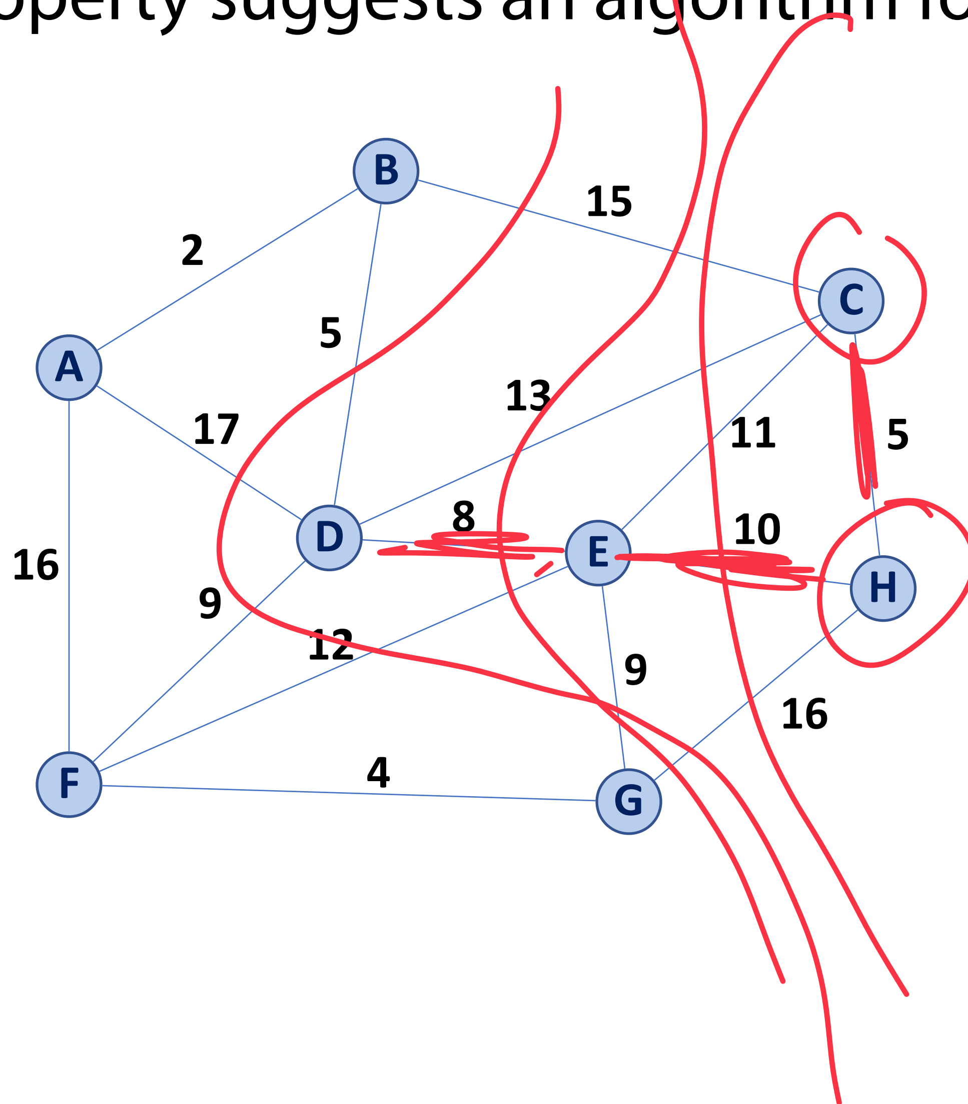
Consider an arbitrary partition of the vertices on **G** into two subsets **U** and **V**

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

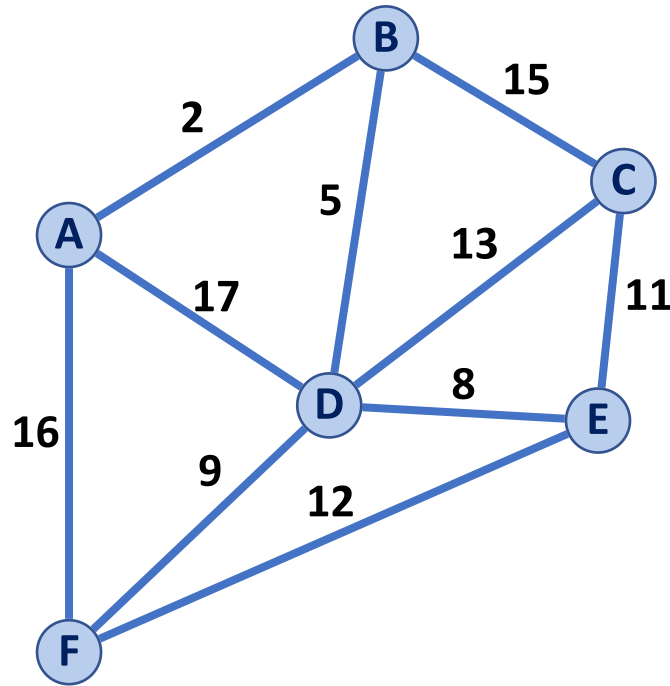


Partition Property

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

```
 1 def prim(G, start):
 2     outEdges=[]
 3     dist = {}
 4     prev = {}
 5     inGroup = set()
 6 
 7     for v in G.nodes():
 8         dist[v] = float("inf")
 9     dist[start]=0
10     prev[start]=None
11 
12     for _ in range(len(G.nodes()):
13         v = minOutEdge(dist, inGroup)
14         inGroup.add(v)
15         if prev[v]!=None:
16             outEdges.append( (prev[v], v, dist[v]) )
17 
18         for u in nx.neighbors(G, v):
19             weight = G[u][v][‘weight’]
20             if u not in inGroup and dist[u] > weight:
21                 dist[u]=weight
22                 prev[u]=v
23     return outEdges
24 
```

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 $O(n)$

```
1  def prim(G, start):
2      outEdges=[]
3      dist = {}
4      prev = {}
5      inGroup = set()
6
7      for v in G.nodes():
8          dist[v] = float("inf")
9      dist[start]=0
10     prev[start]=None
11
12     for _ in range(len(G.nodes()):
13         v = minOutEdge(dist, inGroup)
14         inGroup.add(v)
15         if prev[v]!=None:
16             outEdges.append( (prev[v], v, dist[v]) )
17
18         for u in nx.neighbors(G, v):
19             weight = G[u][v][`weight']
20             if u not in inGroup and dist[u] > weight:
21                 dist[u]=weight
22                 prev[u]=v
23     return outEdges
24
```

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***