Algorithms and Data Structures for Data Science
Minimum Spanning Tree

CS 277
Brad Solomon

April 26, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Please fill out end-of-semester evaluations

Feedback is important for the development of the class

This is especially important for this class as it is very new and growing

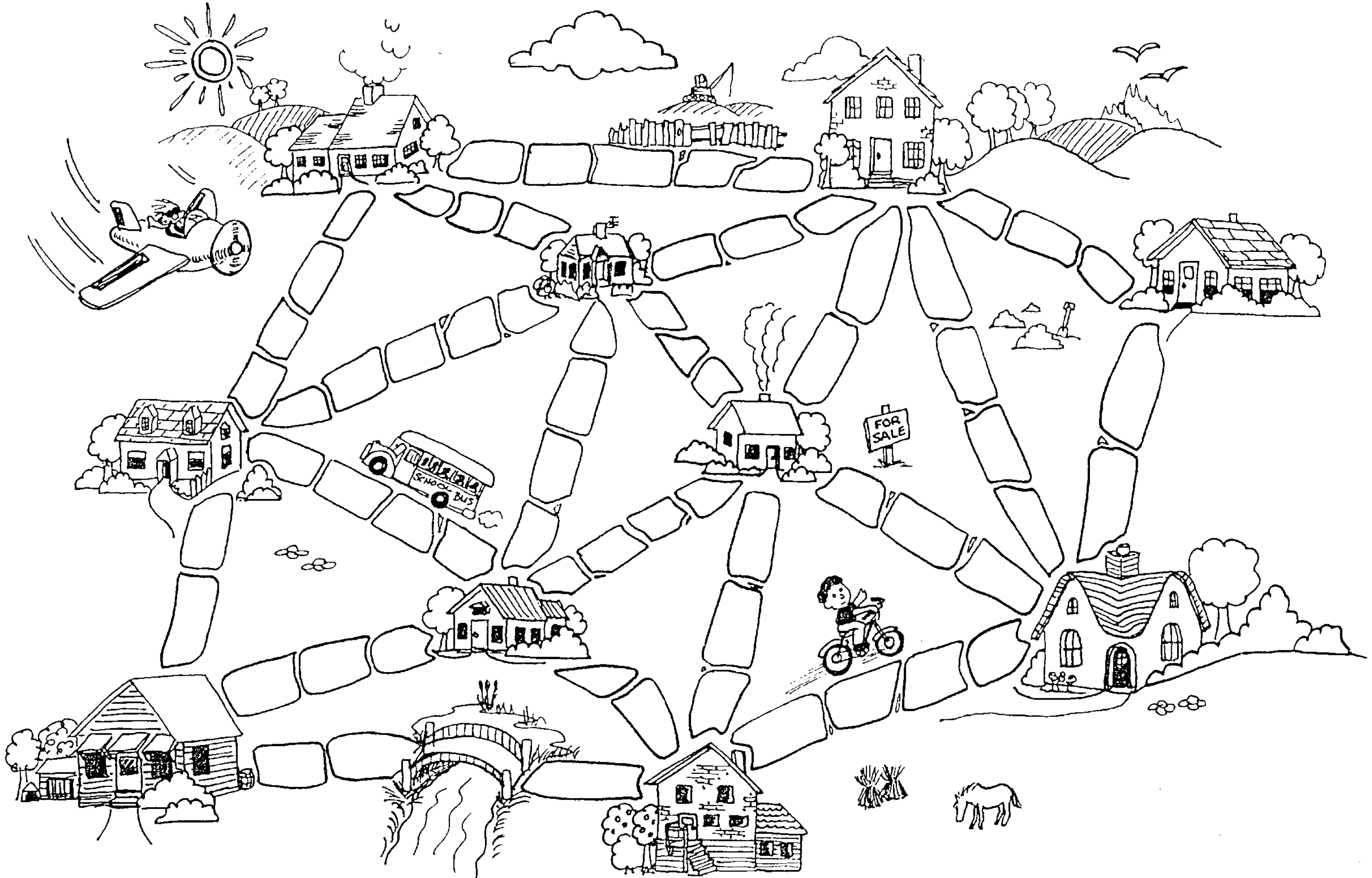
An optional final reflection form: Mini-Project Reflection Feedback Form

Learning Objectives

Formalize Minimum Spanning Tree Problem

Conceptualize Kruskal and Prim MST algorithms

Compare runtimes and implementation details


SCHOOL BUS
FOR SALE

Minimum Spanning Tree

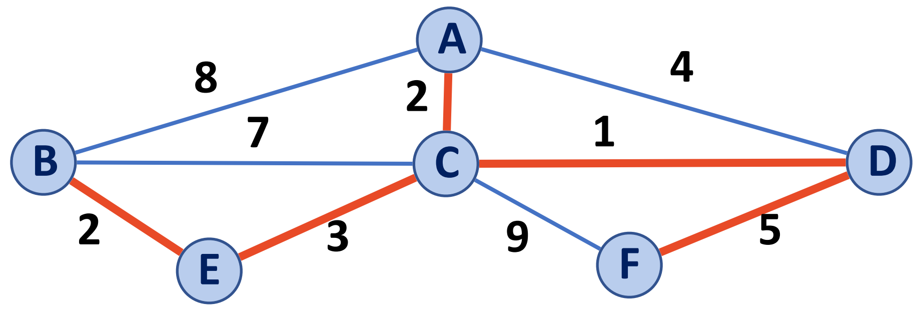
Input: Connected, undirected graph G with positive edge weights

Output: A graph G' with the following properties:

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

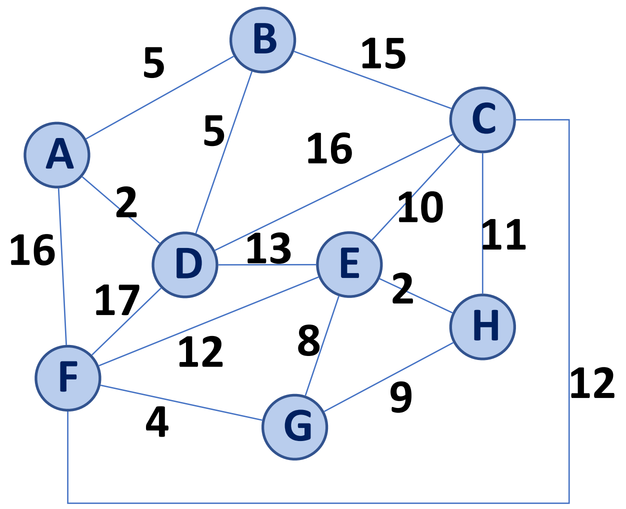
G' is acyclic

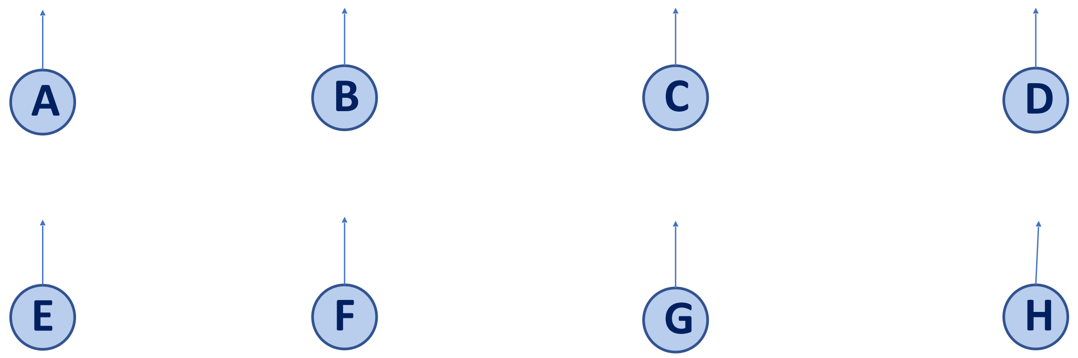
G' has a minimal total weight among all possible spanning trees



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(inGraph):
    sortedEdgeList = sortEdges(inGraph)
    outEdges = []

    part = {}
    belong = {}
    i = 0
    for v in inGraph.edges.keys():
        belong[v]=i
        part[i]=set(v)
        i+=1

    i=0
    numV = len(inGraph.edges.keys())
    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

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

    part = {}
    belong = {}
    i = 0
    for v in inGraph.edges.keys():
        belong[v]=i
        part[i]=set(v)
        i+=1

    i=0
    numV = len(inGraph.edges.keys())
    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

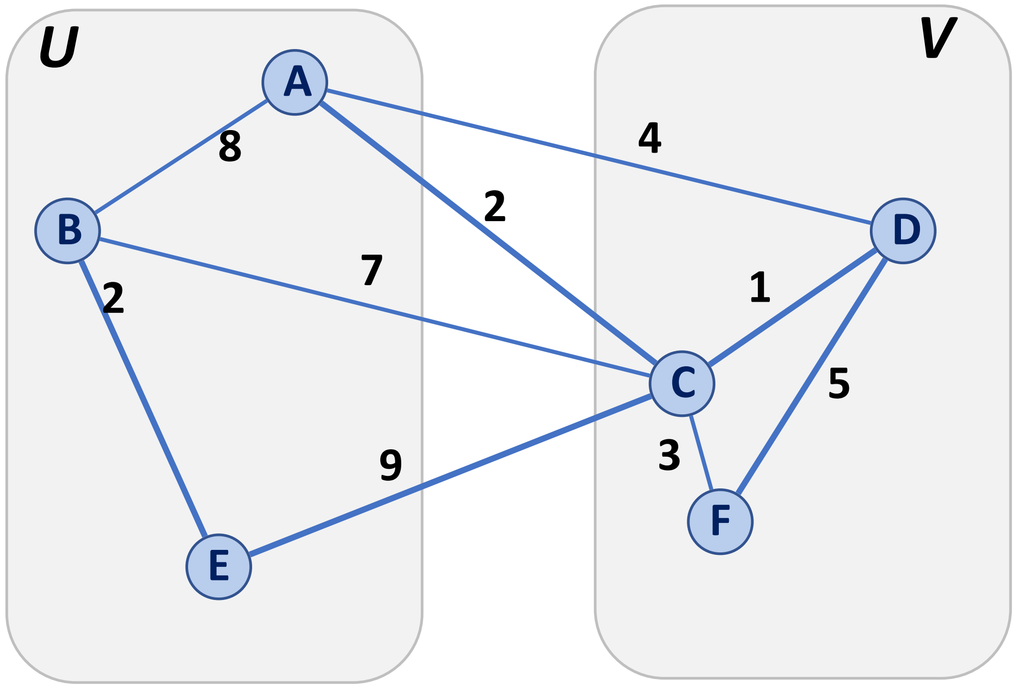
   $O(m + n)$ **total****

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

    part = {}
    belong = {}
    i = 0
    for v in inGraph.edges.keys():
        belong[v]=i
        part[i]=set(v)
        i+=1

    i=0
    numV = len(inGraph.edges.keys())
    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
```

Partition Property

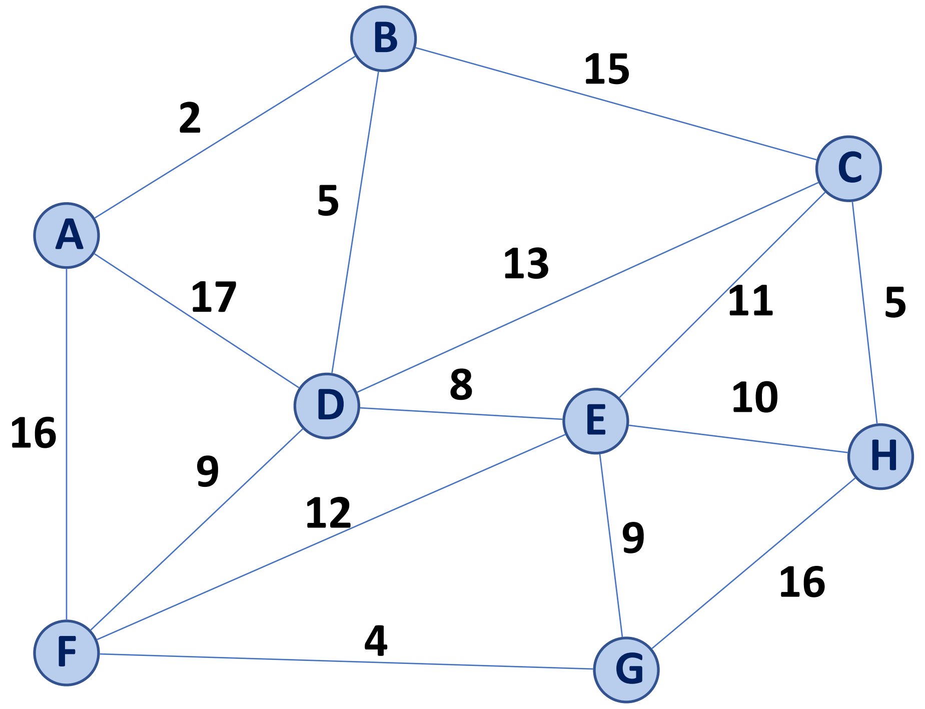
Consider an arbitrary partition of the vertices on **G** into two subsets **U** and **V**

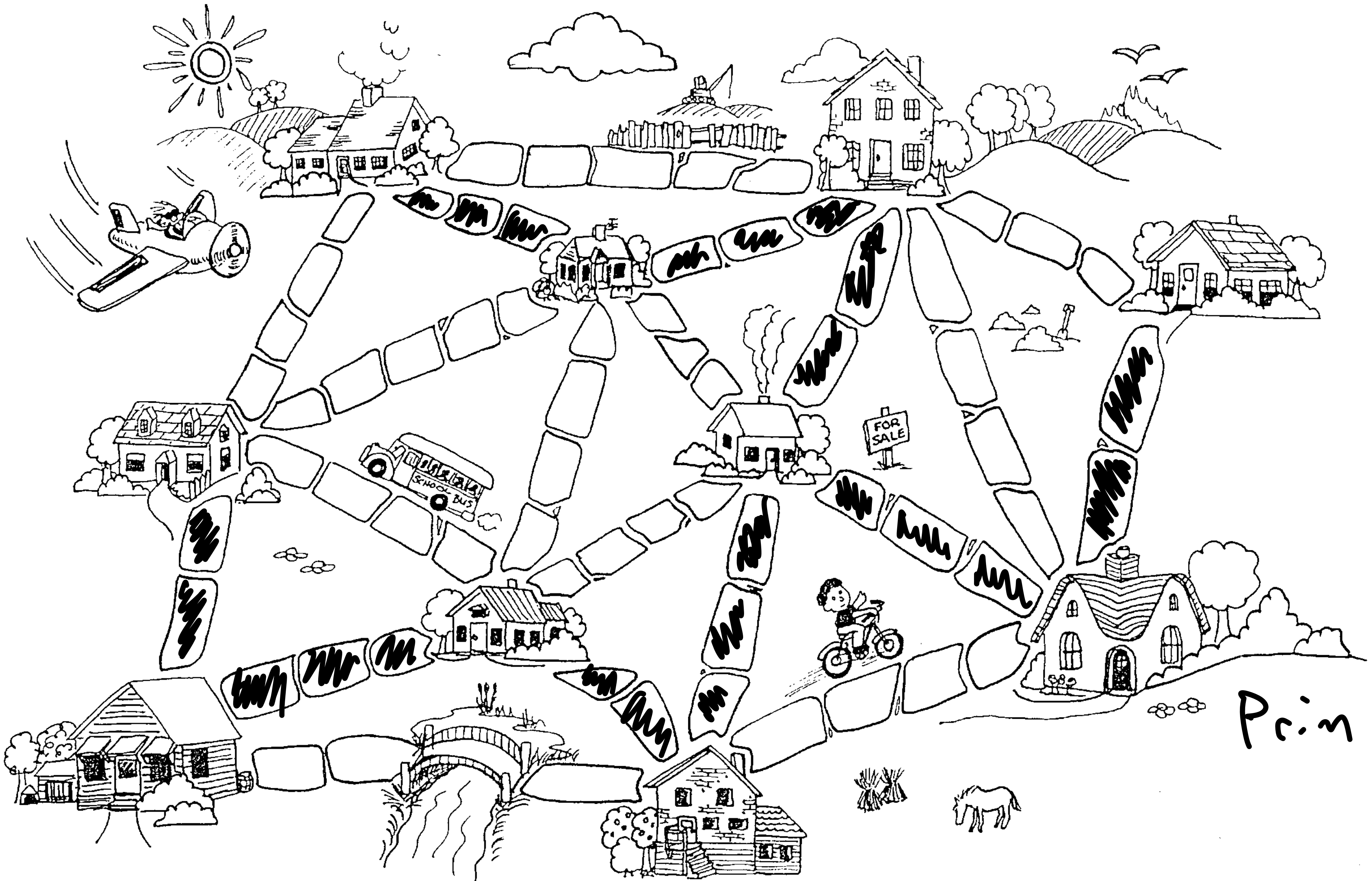
If **e** is an edge of minimum weight across the partition, then there exists a minimum spanning tree containing **e**



Partition Property

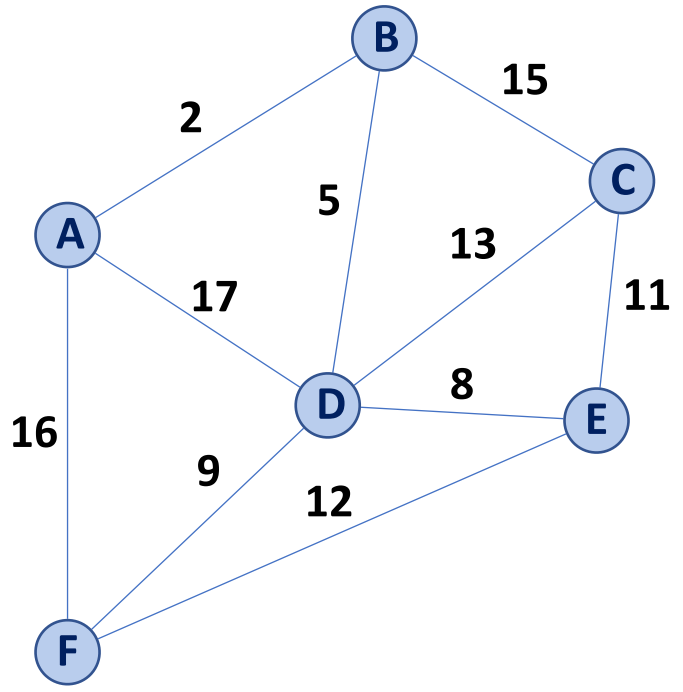
The partition property suggests an algorithm for MST:





Prim

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’

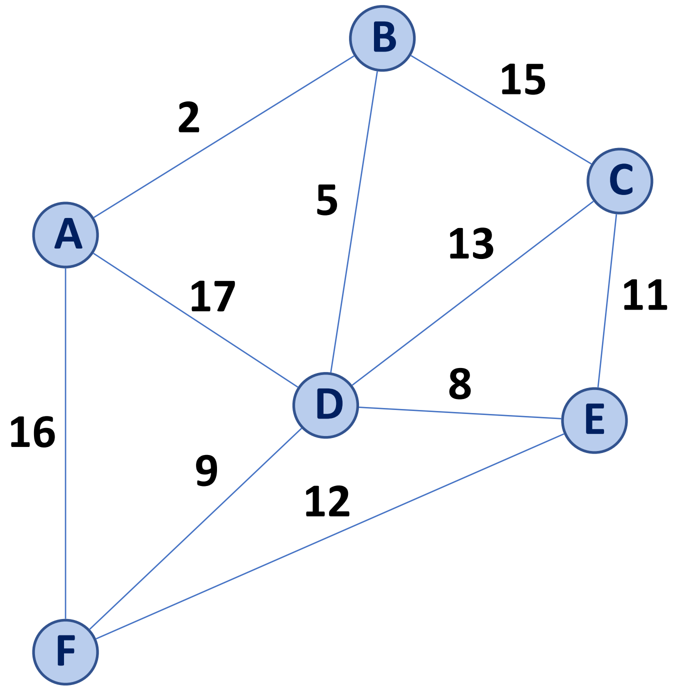
   Add the vertex this edge links to to ‘in’

   Update distances between ‘in’ and ‘out’

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

4) Repeat (3) once for every vertex

Prim’s Algorithm



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

    for v in inGraph.edges.keys():
        dist[v] = float("inf")
    dist[start]=0
    prev[start]=None

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

        for e in inGraph.edges[v]:
            u = e[0]
            weight = e[1]
            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

2. Get min valid edge

3. Add edge to MST

4. Update all distances

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

    for v in inGraph.edges.keys():
        dist[v] = float("inf")
    dist[start]=0
    prev[start]=None

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

        for e in inGraph.edges[v]:
            u = e[0]
            weight = e[1]
            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

   $O(n)$

x $O(n)$

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

    for v in inGraph.edges.keys():
        dist[v] = float("inf")
    dist[start]=0
    prev[start]=None

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

        for e in inGraph.edges[v]:
            u = e[0]
            weight = e[1]
            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***