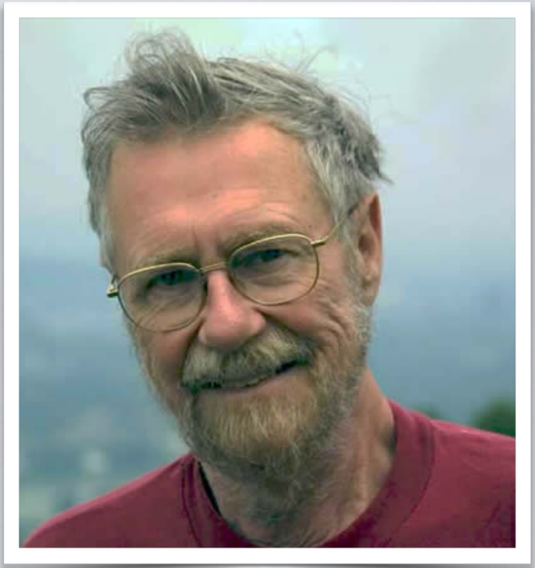
Dijkstra's Algorithm



- The algorithm is as follows:
  - Decorate source with distance 0 & all other nodes with ∞
  - Add all nodes to priority queue w/ distance as priority
  - While the priority queue isn't empty
    - Remove node from queue with minimal priority
    - Update distances of the removed node's neighbors if distances decreased
- When algorithm terminates, every node is decorated with minimal cost from source

Today:
- Finishing Dijkstra's
- Minimum spanning trees

Dijkstra Pseudo-Code

```
function dijkstra(G, s):
   // Input: graph G with vertices V, and source s
   // Output: Nothing
   // Purpose: Decorate nodes with shortest distance from s
   for v in V:
     v.dist = infinity  // Initialize distance decorations
     v.prev = null      // Initialize previous pointers to null
   s.dist = 0         // Set distance to start to 0

   PQ = PriorityQueue(V)    // Use v.dist as priorities
   while PQ not empty:
      u = PQ.removeMin()
      for all edges (u, v):  //each edge coming out of u
         if u.dist + cost(u, v) < v.dist: // cost() is weight
            v.dist = u.dist + cost(u,v)   // Replace as necessary
            v.prev = u      // Maintain pointers for path
            PQ.decreaseKey(v, v.dist)
```

Dijkstra Runtime w/ Heap

- If PQ implemented with Heap
  - `insert( )` is `O(log|V|)`
    - you may need to upheap
  - `removeMin( )` is `O(log|V|)`
    - you may need to downheap
  - `decreaseKey()` is `O(log|V|)`
    - assume we have dictionary that maps vertex to heap entry in `O(log|V|)` time (so no need to scan heap to find entry)
    - you may need to upheap after decreasing the key

Dijkstra Runtime w/ Heap

```
function dijkstra(G, s):
   for v in V:                                   <-- O(|V|)
      v.dist = infinity
      v.prev = null
   s.dist = 0

   PQ = PriorityQueue(V)                         <-- O(|V|log|V|)
   while PQ not empty:                           <-- O(|V|)
       u = PQ.removeMin()                        <-- O(log|V|)
       for all edges (u, v):                     <-- O(|E|) total
           if v.dist > u.dist + cost(u, v):
              v.dist = u.dist + cost(u,v)
              v.prev = u
              PQ.decreaseKey(v, v.dist)          <-- O(log|V|)
```

Dijkstra Runtime w/ Heap

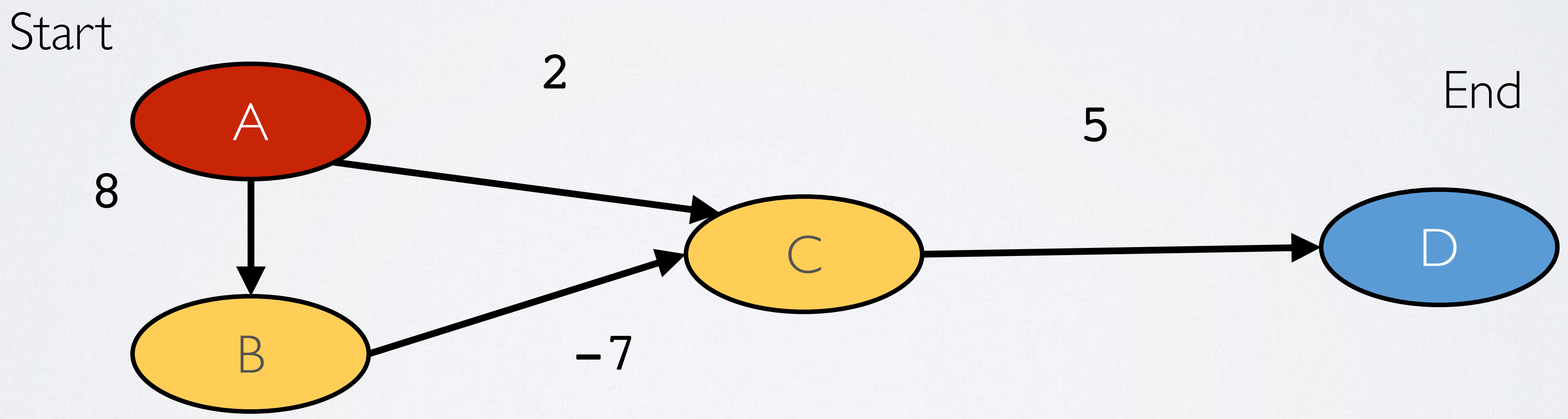
- If PQ implemented with Heap

$$\begin{aligned} &O(|V| + |V|\log|V| + |V|\log|V| + |E|\log|V|) \\ &\quad = O(|V| + |V|\log|V| + |E|\log|V|) \\ &\quad = O\Big((|V| + |E|)\cdot\log|V|\Big) \end{aligned}$$

- Note
  - though the **`O(|E|)`** loop is nested in the **`O(|V|)`** loop
  - we visit each edge at most twice rather than **`|V|`** times
  - That's why while loop is $O\Big((V\log|V|) + (|E|\log|V|)\Big)$

Dijkstra isn't perfect!

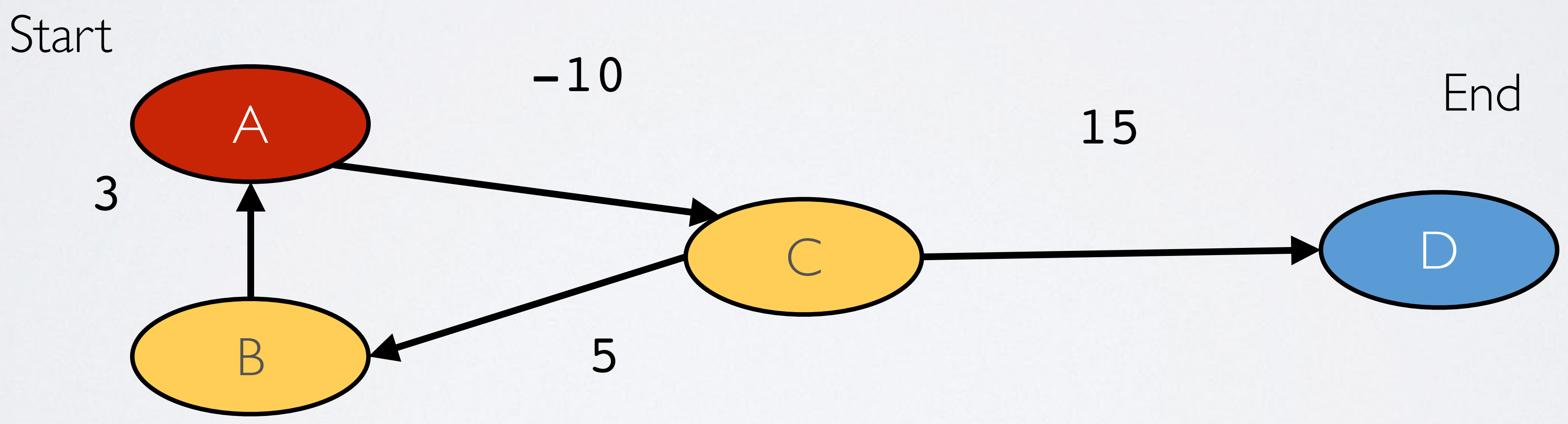
- We can find shortest path on weighted graph in
  - `O((|V|+|E|)×log|V|)`
  - or can we…
- Dijkstra fails with negative edge weights



- Returns **`[A,C,D]`** when it should return **`[A,B,C,D]`**

Negative Edge Weights

- Negative edge weights are problem for Dijkstra
- But negative cycles are even worse!
  - because there is no true shortest path!



Bellman-Ford Algorithm

- Algorithm that handles graphs w/ neg. edge weights
- Similar to Dijkstra's but more robust
  - Returns same output as Dijkstra's for any graph w/ only positive edge weights (but runs slower)
  - Returns correct shortest paths for graphs w/ neg. edge weights
  - Detects and reports negative cycles
  - How: not greedy!

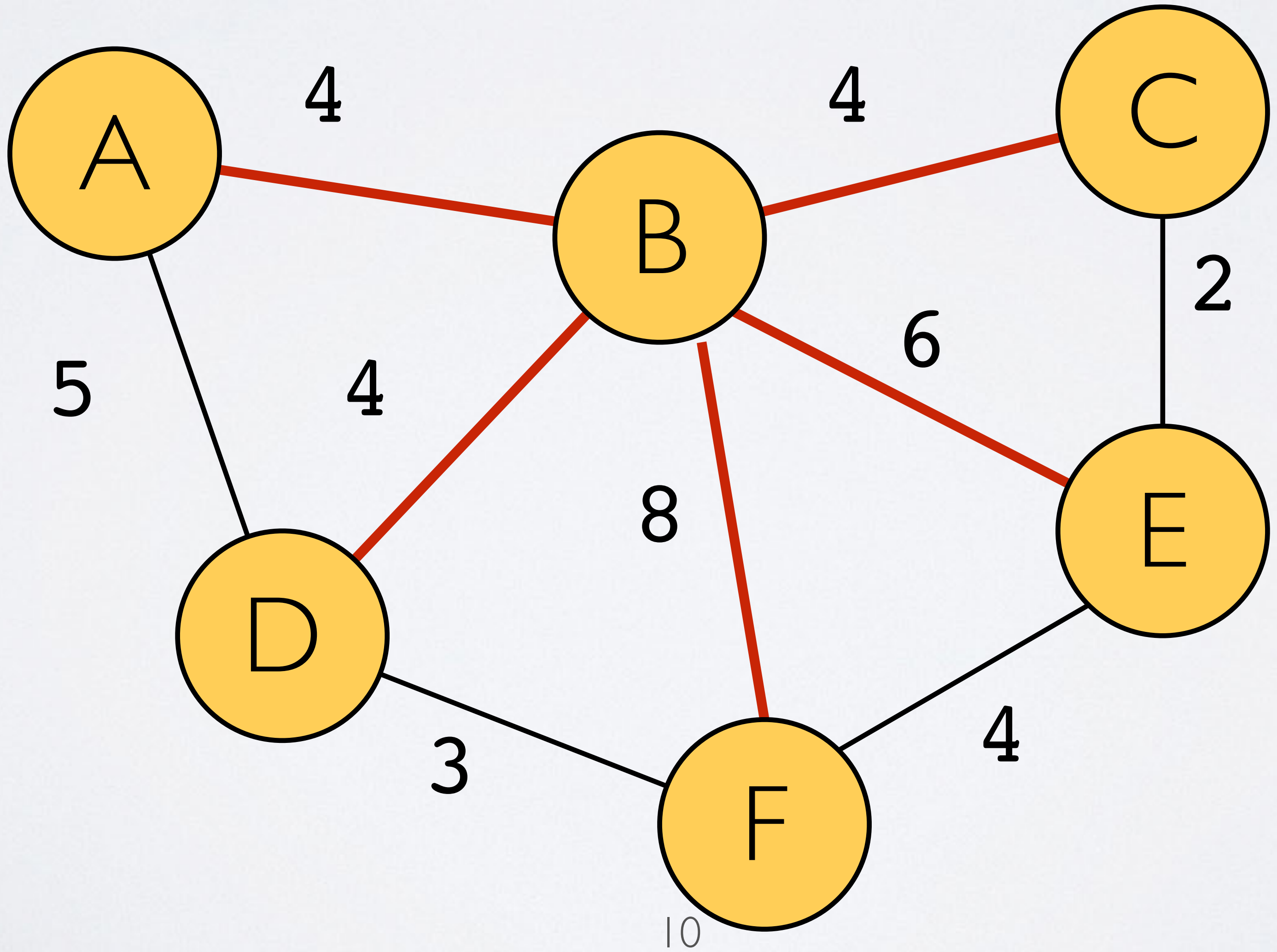
Minimum Spanning Trees: Prim-Jarnik

CS16: Introduction to Data Structures & Algorithms

Summer 2021

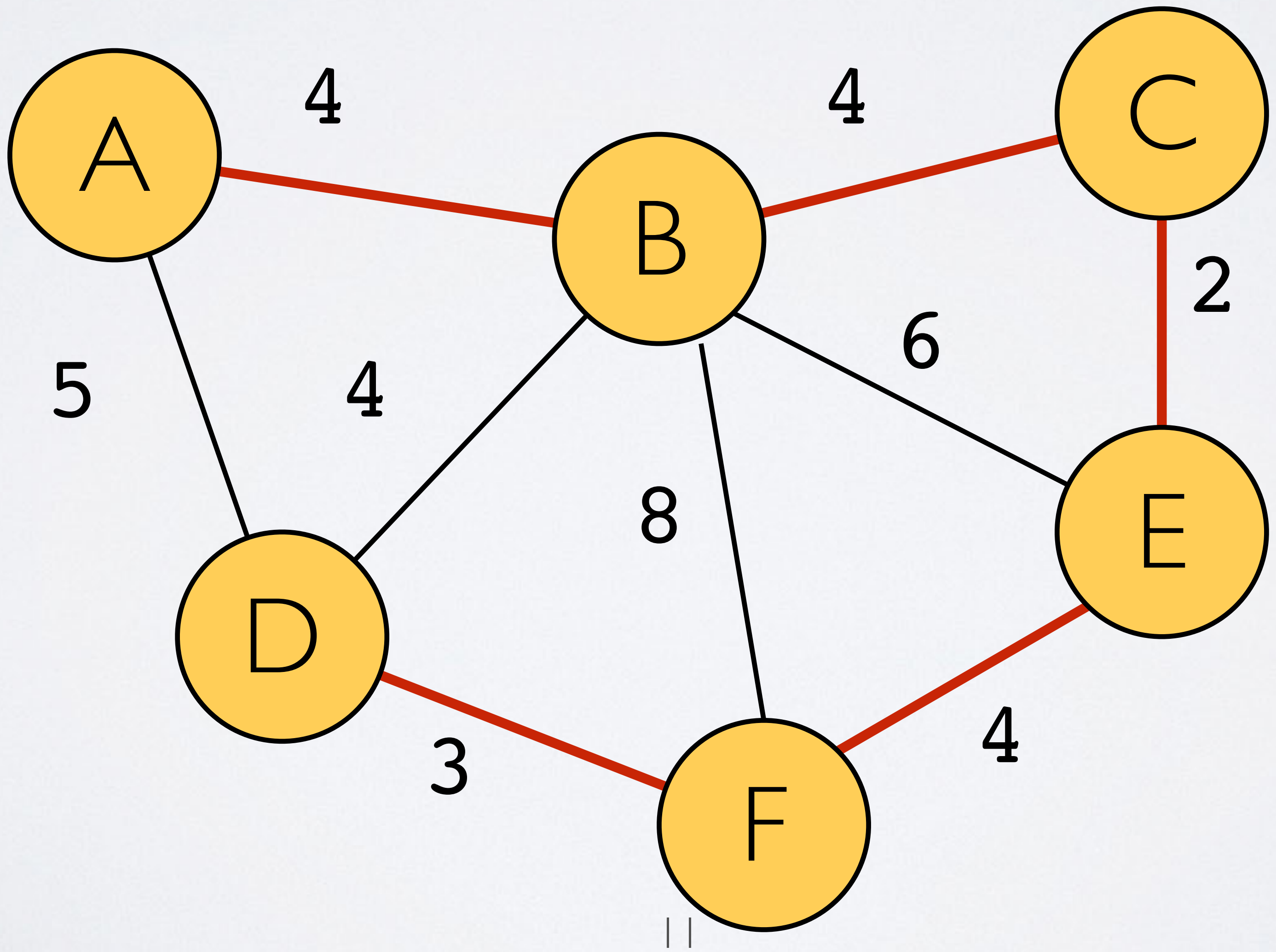
Spanning Trees

- A **spanning tree** of a graph is
  - edge subset forming a tree that spans every vertex



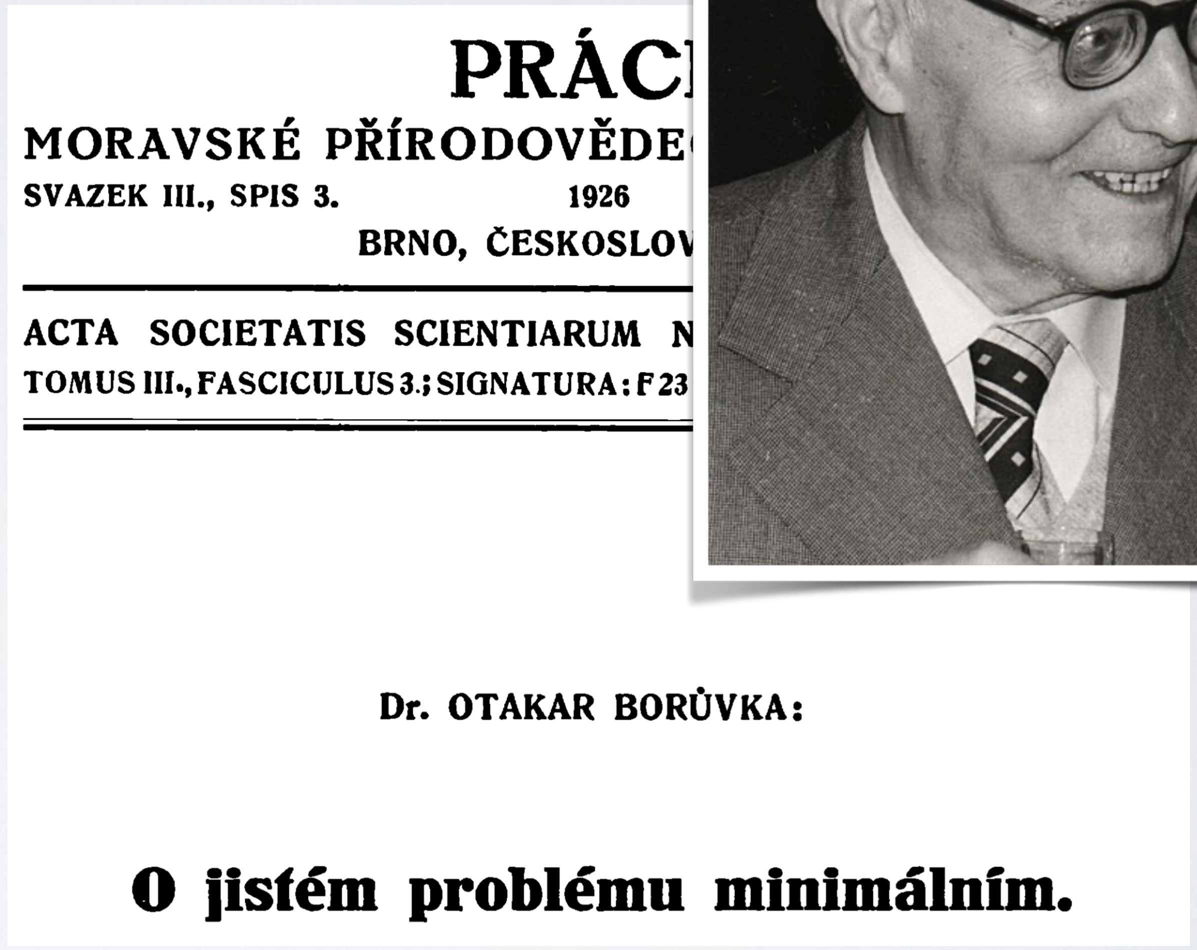
Minimum Spanning Trees

- A **minimum spanning tree** (MST) is
  - spanning tree with minimum total edge weight



Applications

- Networks
  - **electric**
  - computer
  - water
  - transportation
- Computer vision
  - Facial recognition
  - Handwriting recognition
  - Image segmentation
- Low-density parity check codes (LDPC)



PRÁC

MORAVSKÉ PŘÍRODOVĚDE

SVAZEK III., SPIS 3. 1926

BRNO, ČESKOSLOV

ACTA SOCIETATIS SCIENTIARUM N

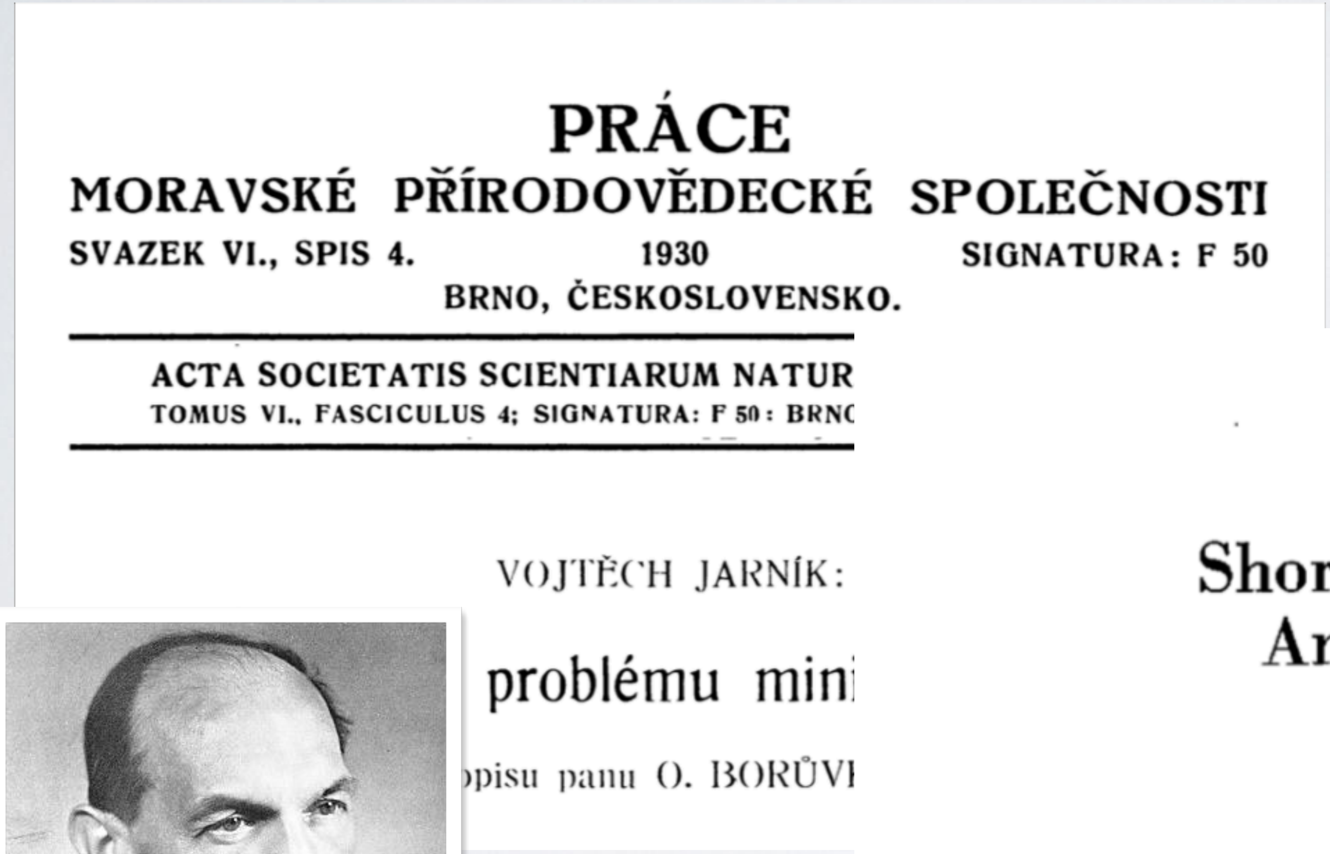
TOMUS III., FASCICULUS 3.; SIGNATURA: F 23

Dr. OTAKAR BORŮVKA:

O jistém problému minimálním.

Minimum Spanning Tree Algos

- Prim-Jarnik Algorithm



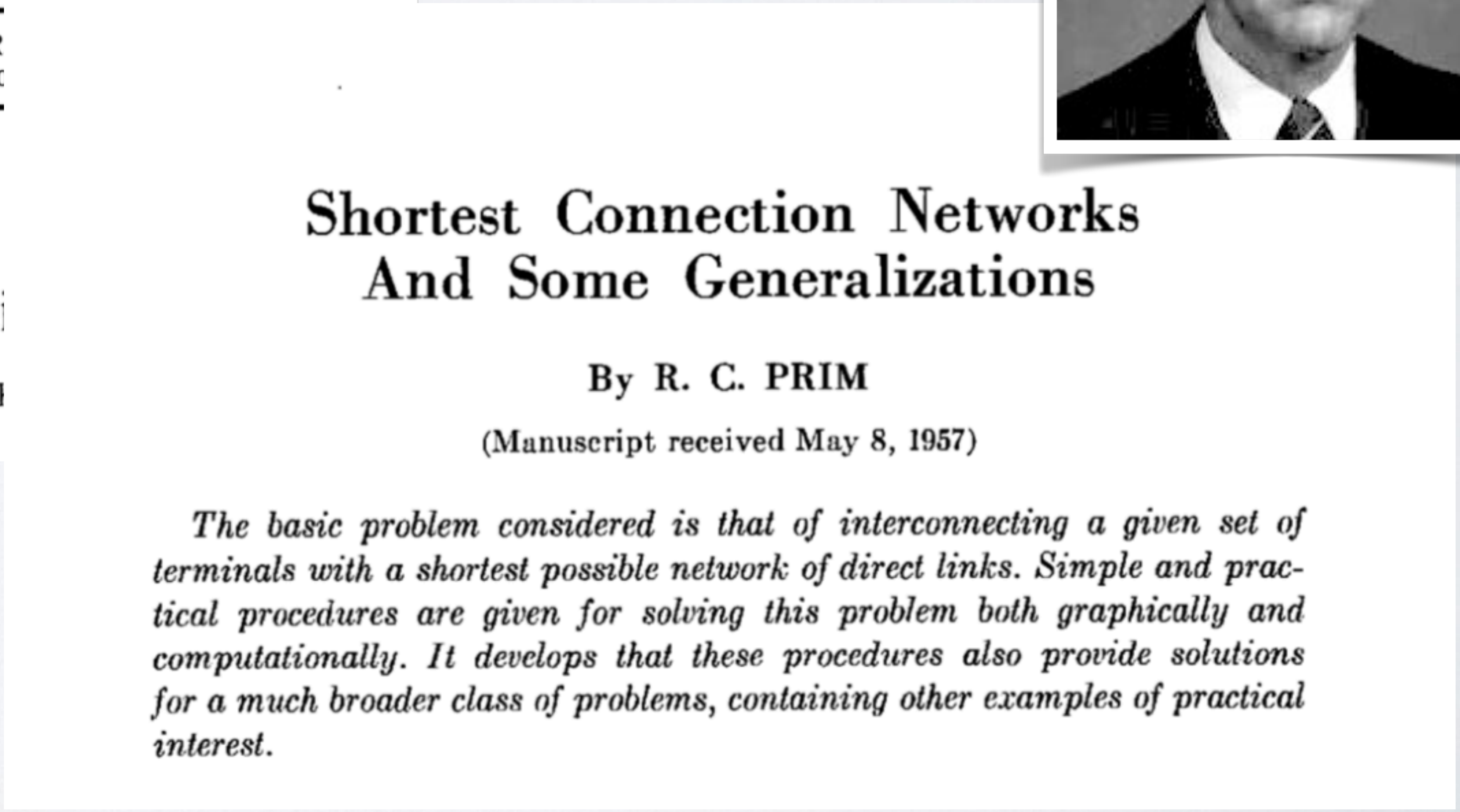
PRÁCE
MORAVSKÉ PŘÍRODOVĚDECKÉ SPOLEČNOSTI
SVAZEK VI., SPIS 4. 1930 SIGNATURA: F 50
BRNO, ČESKOSLOVENSKO.

ACTA SOCIETATIS SCIENTIARUM NATUR
TOMUS VI., FASCICULUS 4; SIGNATURA: F 50: BRNC

VOJTĚCH JARNÍK:

problému mini

pisu panu O. BORŮVI

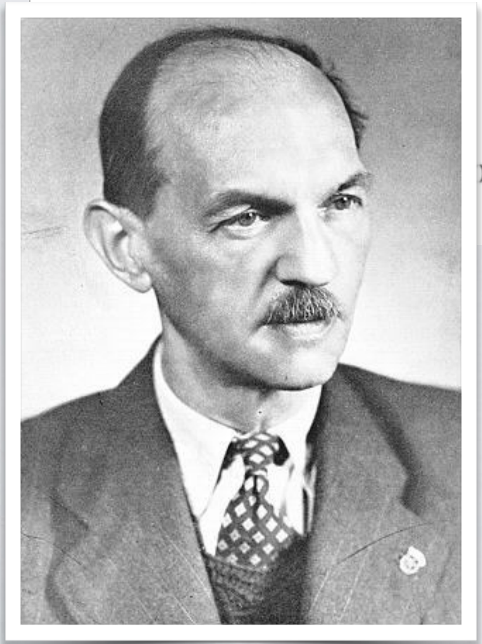


Shortest Connection Networks And Some Generalizations

By R. C. PRIM

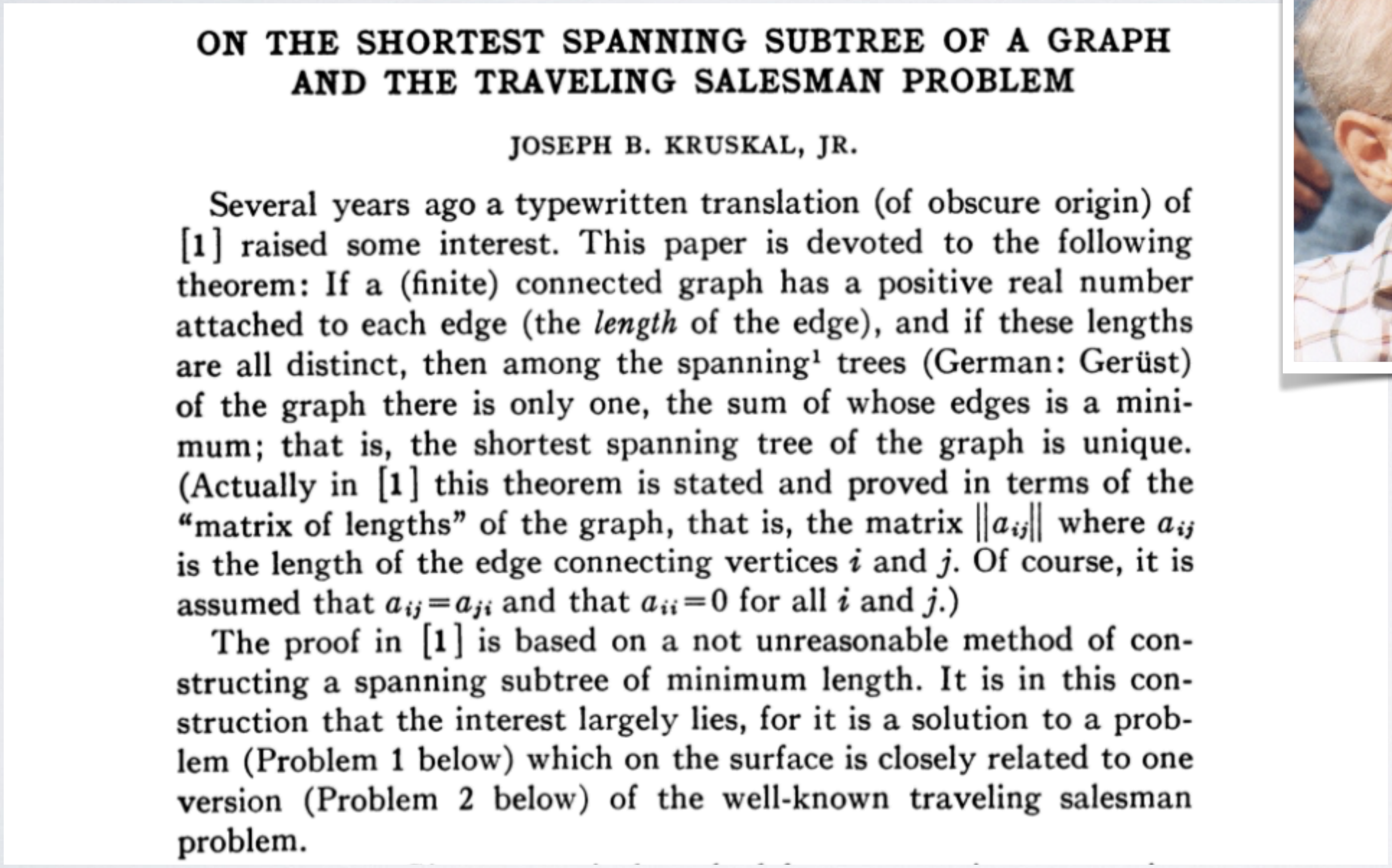
(Manuscript received May 8, 1957)

The basic problem considered is that of interconnecting a given set of terminals with a shortest possible network of direct links. Simple and practical procedures are given for solving this problem both graphically and computationally. It develops that these procedures also provide solutions for a much broader class of problems, containing other examples of practical interest.



Minimum Spanning Tree Algos

- Kruskal's algorithm (1956)



ON THE SHORTEST SPANNING SUBTREE OF A GRAPH AND THE TRAVELING SALESMAN PROBLEM

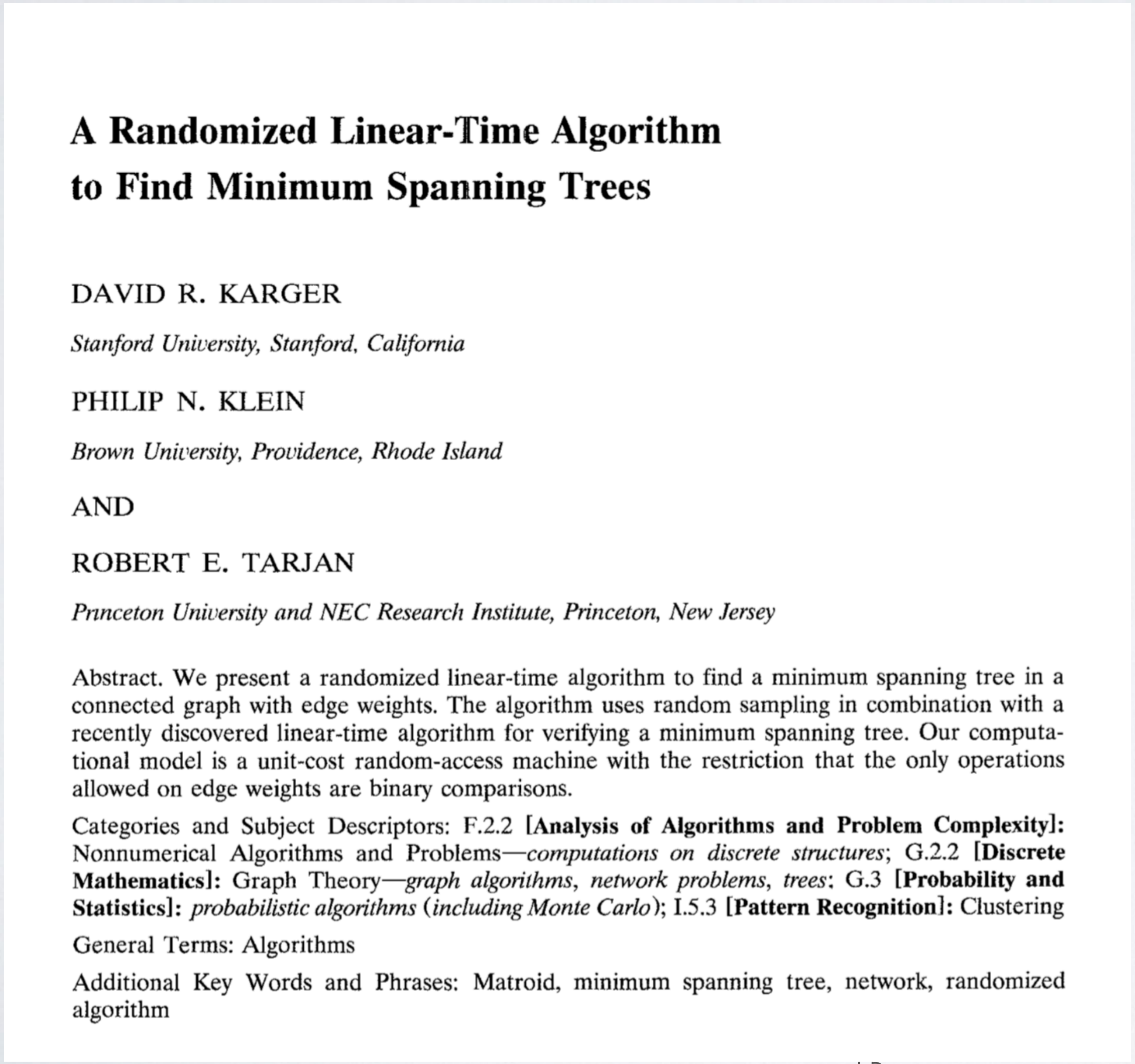
JOSEPH B. KRUSKAL, JR.

Several years ago a typewritten translation (of obscure origin) of [1] raised some interest. This paper is devoted to the following theorem: If a (finite) connected graph has a positive real number attached to each edge (the *length* of the edge), and if these lengths are all distinct, then among the spanning[1] trees (German: Gerüst) of the graph there is only one, the sum of whose edges is a minimum; that is, the shortest spanning tree of the graph is unique. (Actually in [1] this theorem is stated and proved in terms of the "matrix of lengths" of the graph, that is, the matrix $\|a_{ij}\|$ where a_{ij} is the length of the edge connecting vertices i and j. Of course, it is assumed that $a_{ij}=a_{ji}$ and that $a_{ii}=0$ for all i and j.)

The proof in [1] is based on a not unreasonable method of constructing a spanning subtree of minimum length. It is in this construction that the interest largely lies, for it is a solution to a problem (Problem 1 below) which on the surface is closely related to one version (Problem 2 below) of the well-known traveling salesman problem.

Minimum Spanning Tree Algos

- Karger-Klein-Tarjan (1995)



A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees

DAVID R. KARGER

Stanford University, Stanford, California

PHILIP N. KLEIN

Brown University, Providence, Rhode Island

AND

ROBERT E. TARJAN

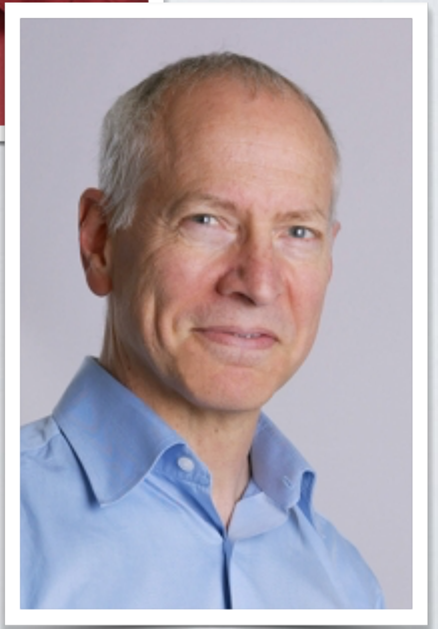
Princeton University and NEC Research Institute, Princeton, New Jersey

Abstract. We present a randomized linear-time algorithm to find a minimum spanning tree in a connected graph with edge weights. The algorithm uses random sampling in combination with a recently discovered linear-time algorithm for verifying a minimum spanning tree. Our computational model is a unit-cost random-access machine with the restriction that the only operations allowed on edge weights are binary comparisons.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*, *network problems*, *trees*; G.3 [**Probability and Statistics**]: *probabilistic algorithms* (*including Monte Carlo*); I.5.3 [**Pattern Recognition**]: Clustering

General Terms: Algorithms

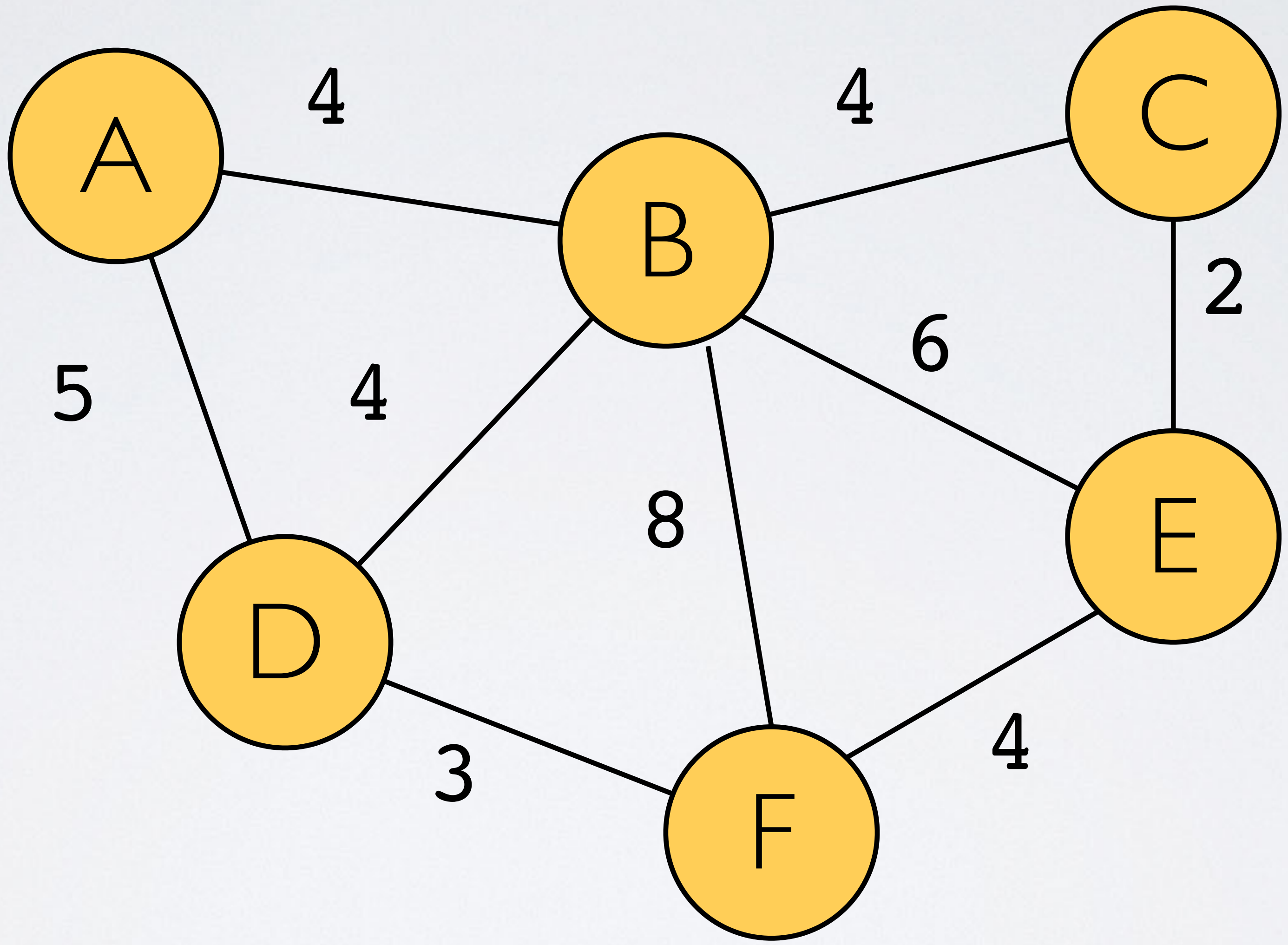
Additional Key Words and Phrases: Matroid, minimum spanning tree, network, randomized algorithm



Prim-Jarnik Algorithm

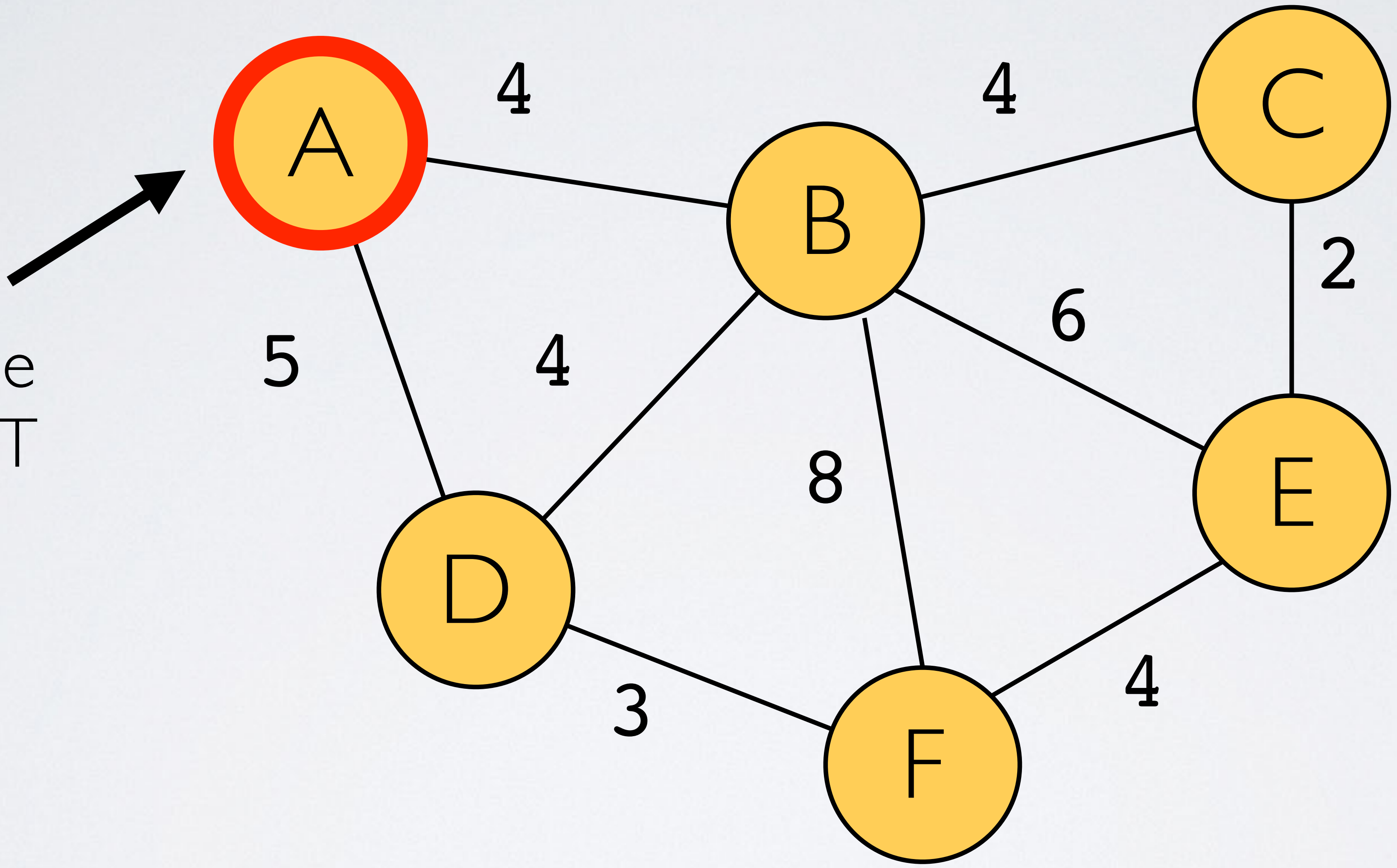
- Add a random node to MST
- At each step
  - Find the unconnected node that can be connected with the lowest-weight edge
  - Add that node and edge to the MST
- Stop when all nodes added to MST

Example

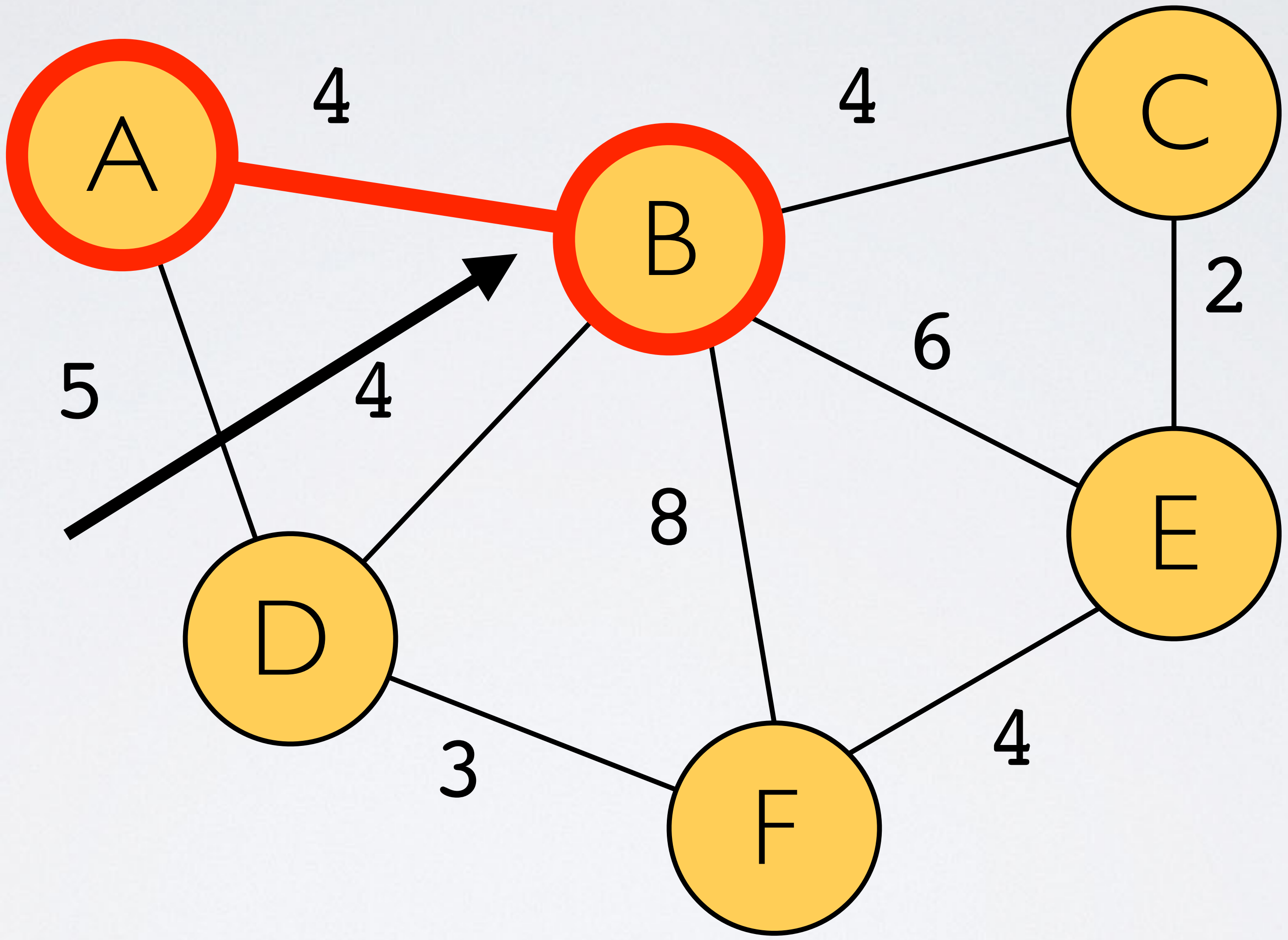


Example

Random node added to MST

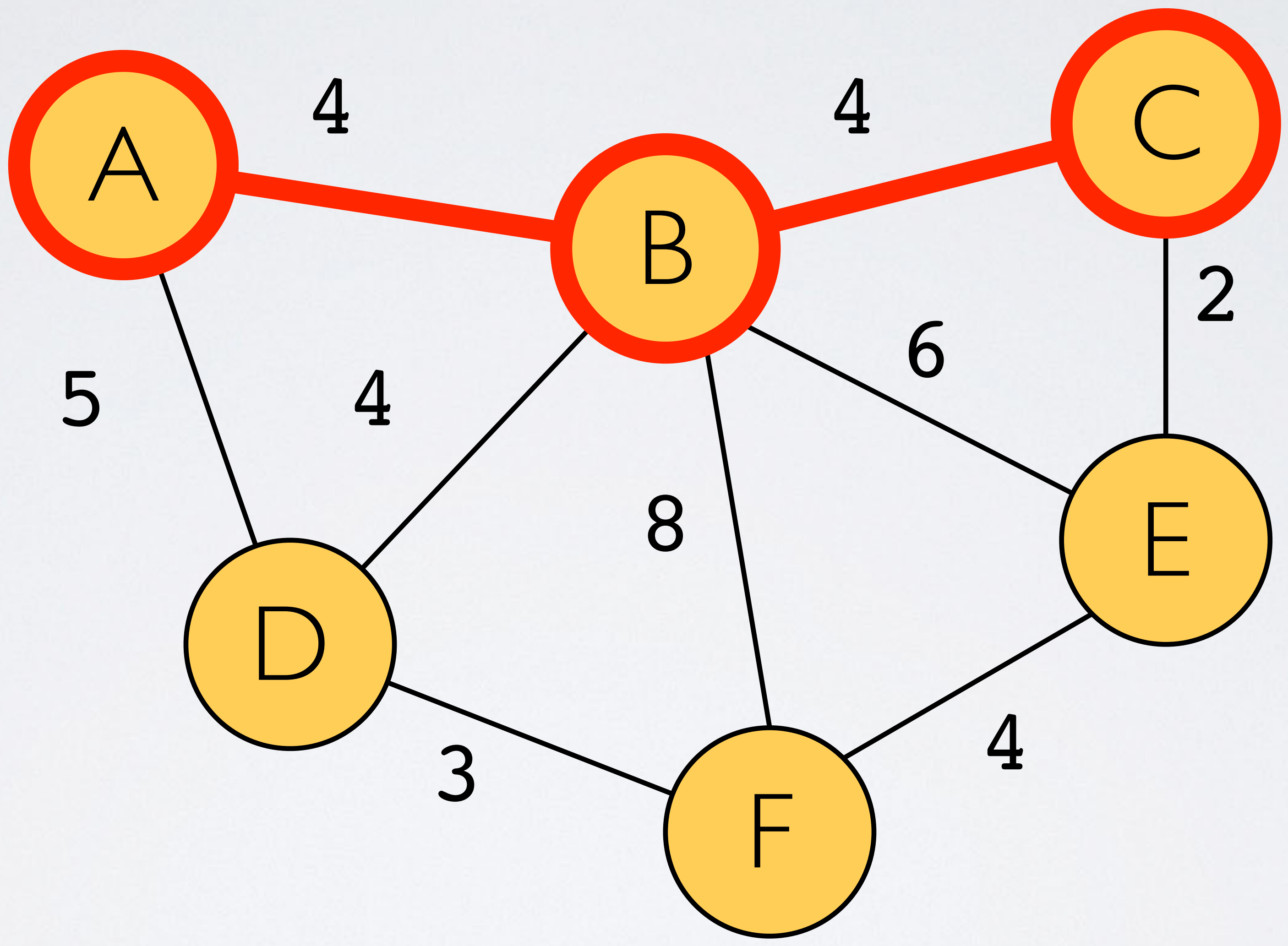


Example



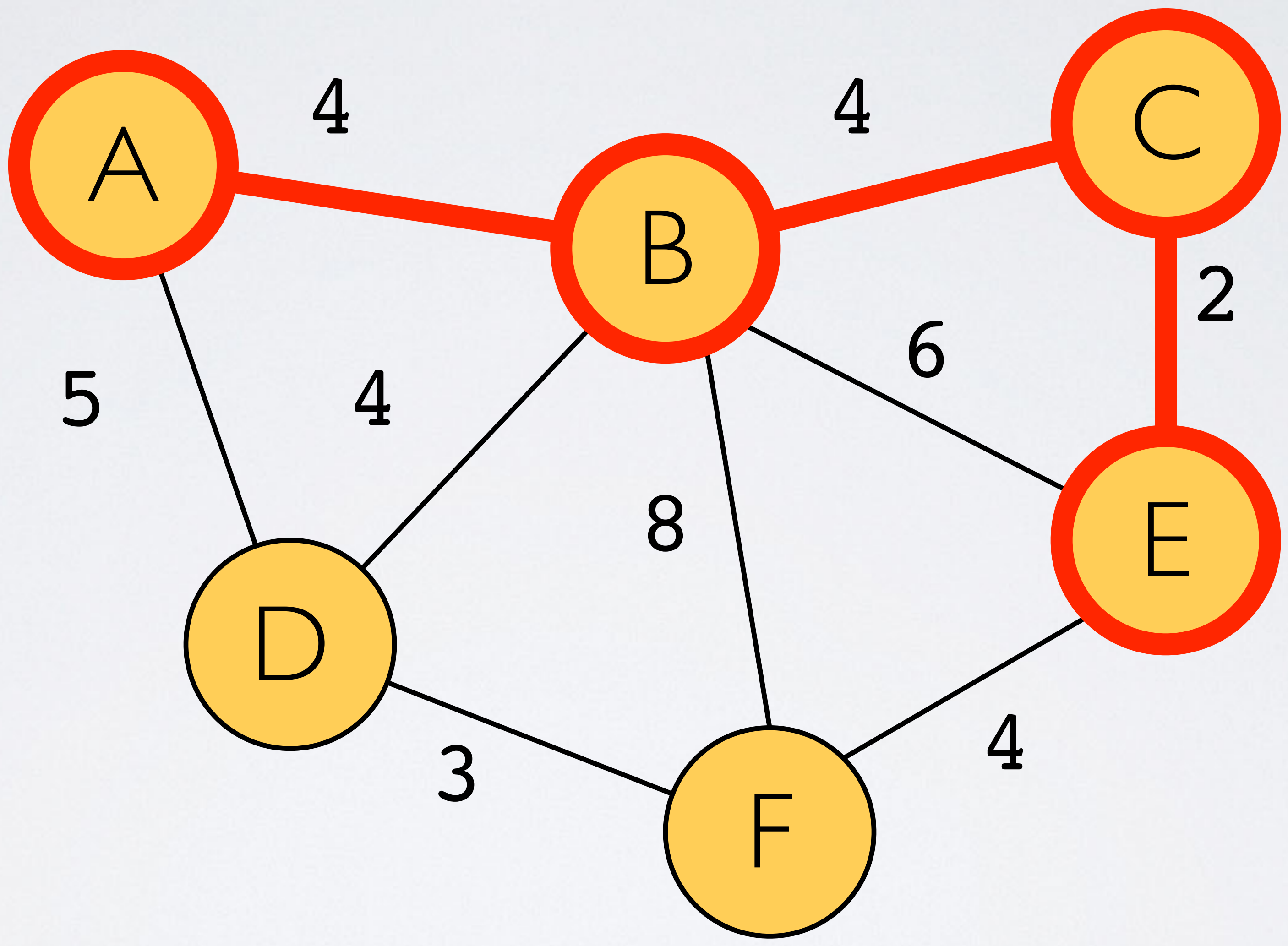
B can be connected by an edge of weight 4

Example

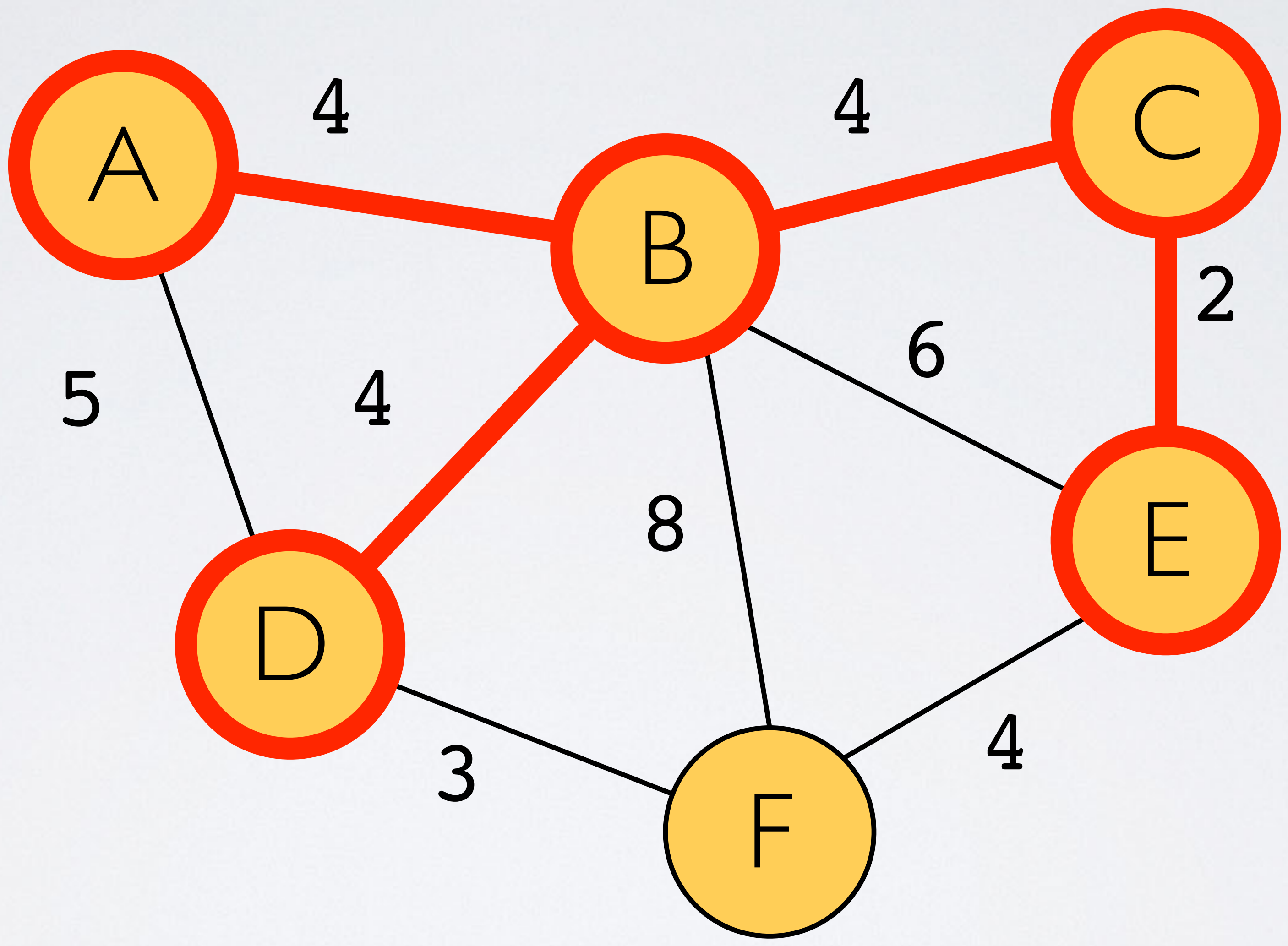


Either **C** or **D** could be added

Example

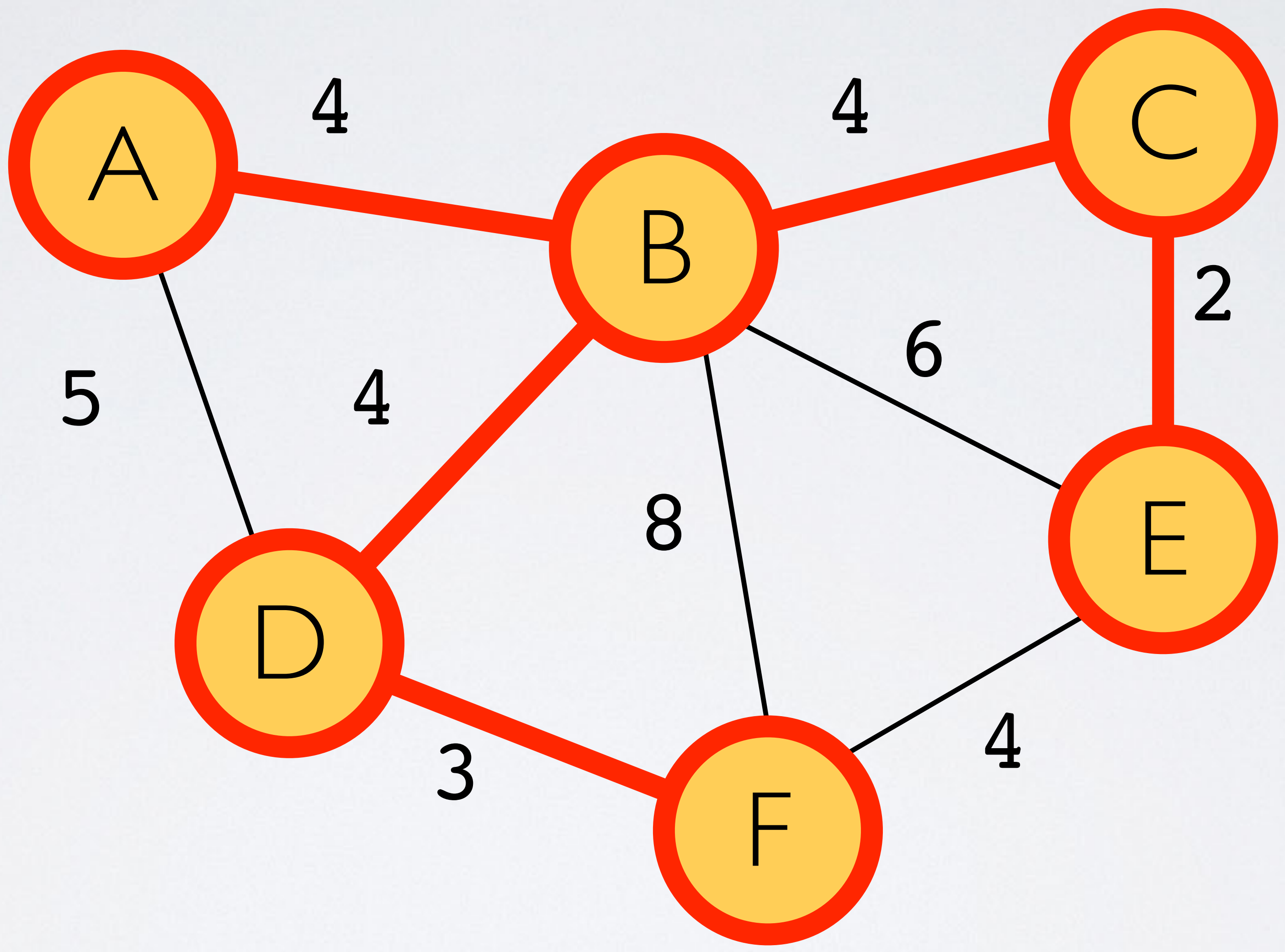


Example



Either **D** or **F** could be added

Example



Either **D** or **F** could be added

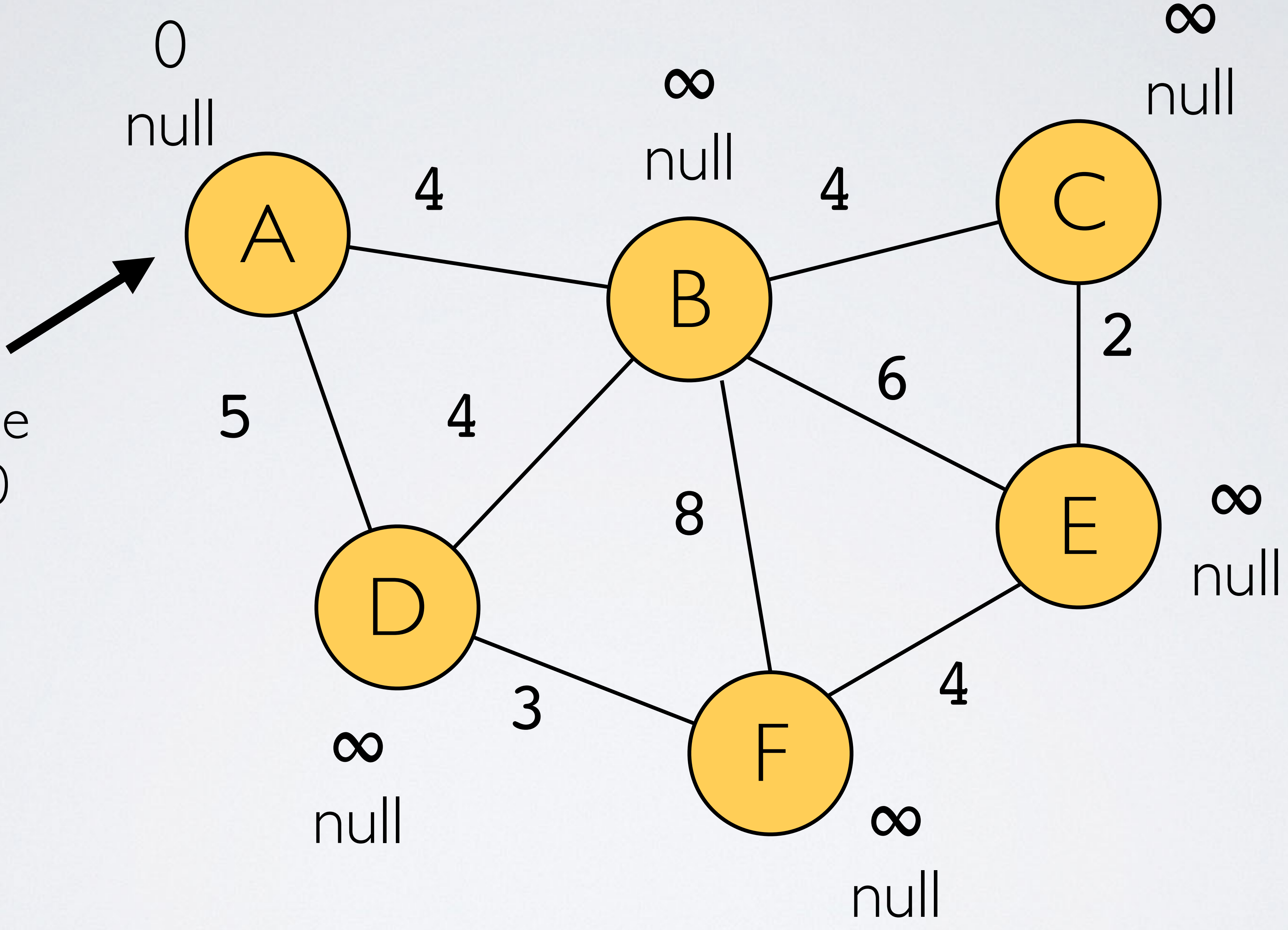
Prim-Jarnik Algorithm

- How to determine which node to add
    - Could consider all edges from MST each time
        - Sounds slow!
    - Instead: use a data structure that contains all unconnected nodes and lets us access the node with the smallest weight
        - Sounds familiar!
        - Think Dijkstra…

Prim-Jarnik Algorithm

- Keep all unconnected nodes in priority queue
- Priority of a node is the minimum weight of an edge connecting that node to the MST
- When adding a new node, update its neighbors' weights in PQ if necessary
- At start, set initial node's priority to 0 and all others to ∞
- Use previous-pointers to determine which edge to add

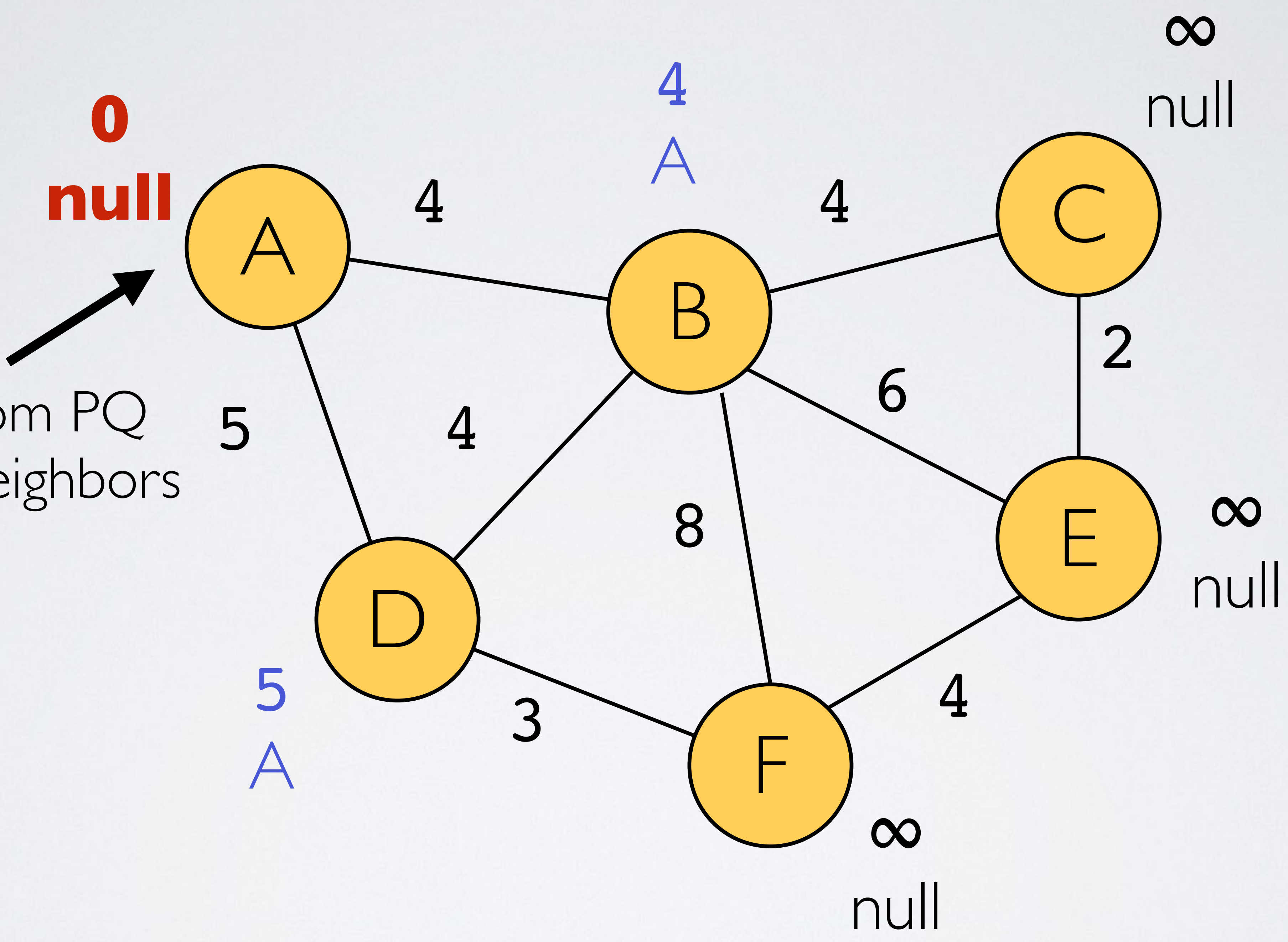
Example



Random node set to cost 0

```
PQ = [(0,A),(∞,B),(∞,C),(∞,D),(∞,E),(∞,F)]
```

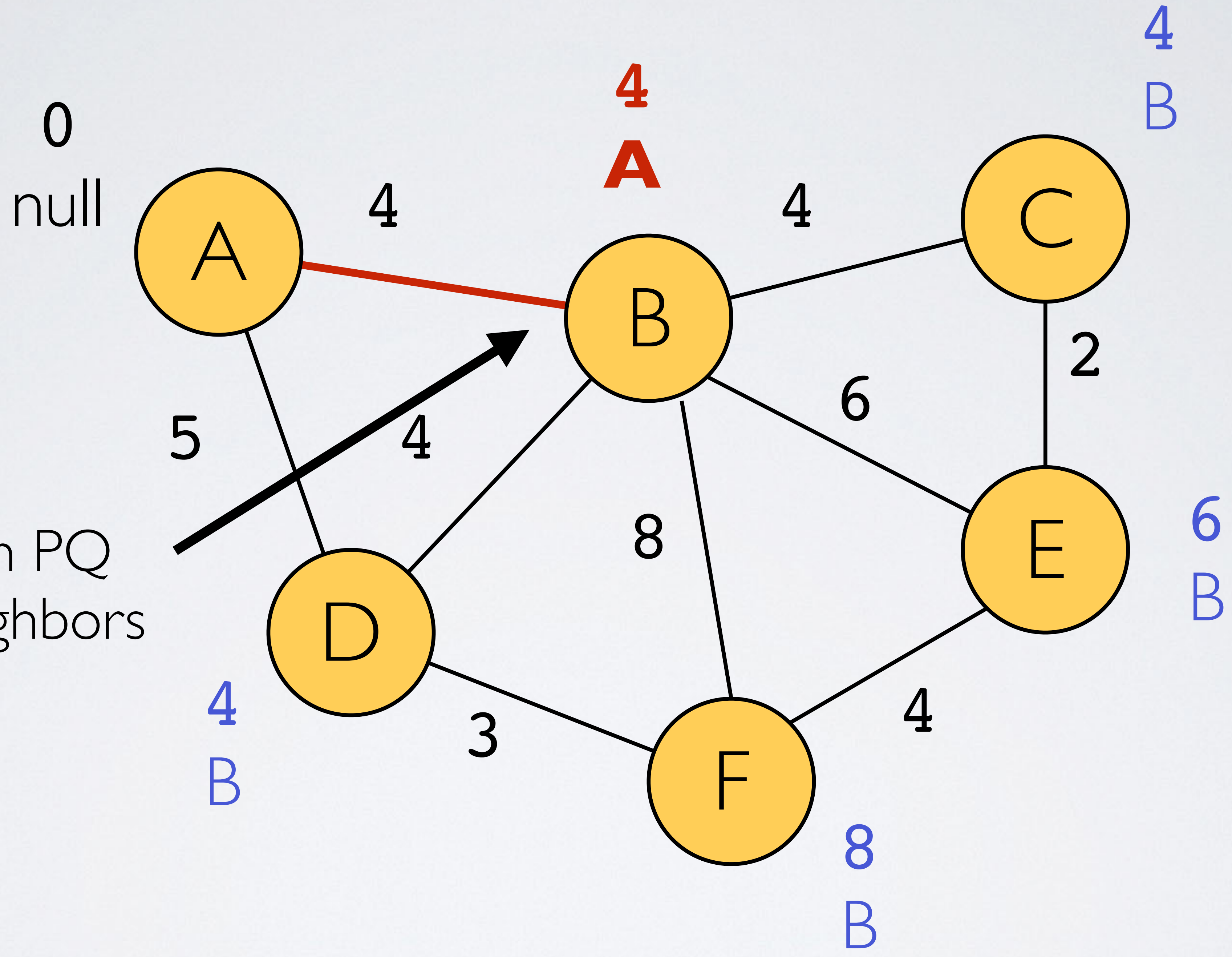
Example



Dequeue from PQ and update neighbors

```
PQ = [(4,B),(5,D),(∞,C),(∞,E),(∞,F)]
```

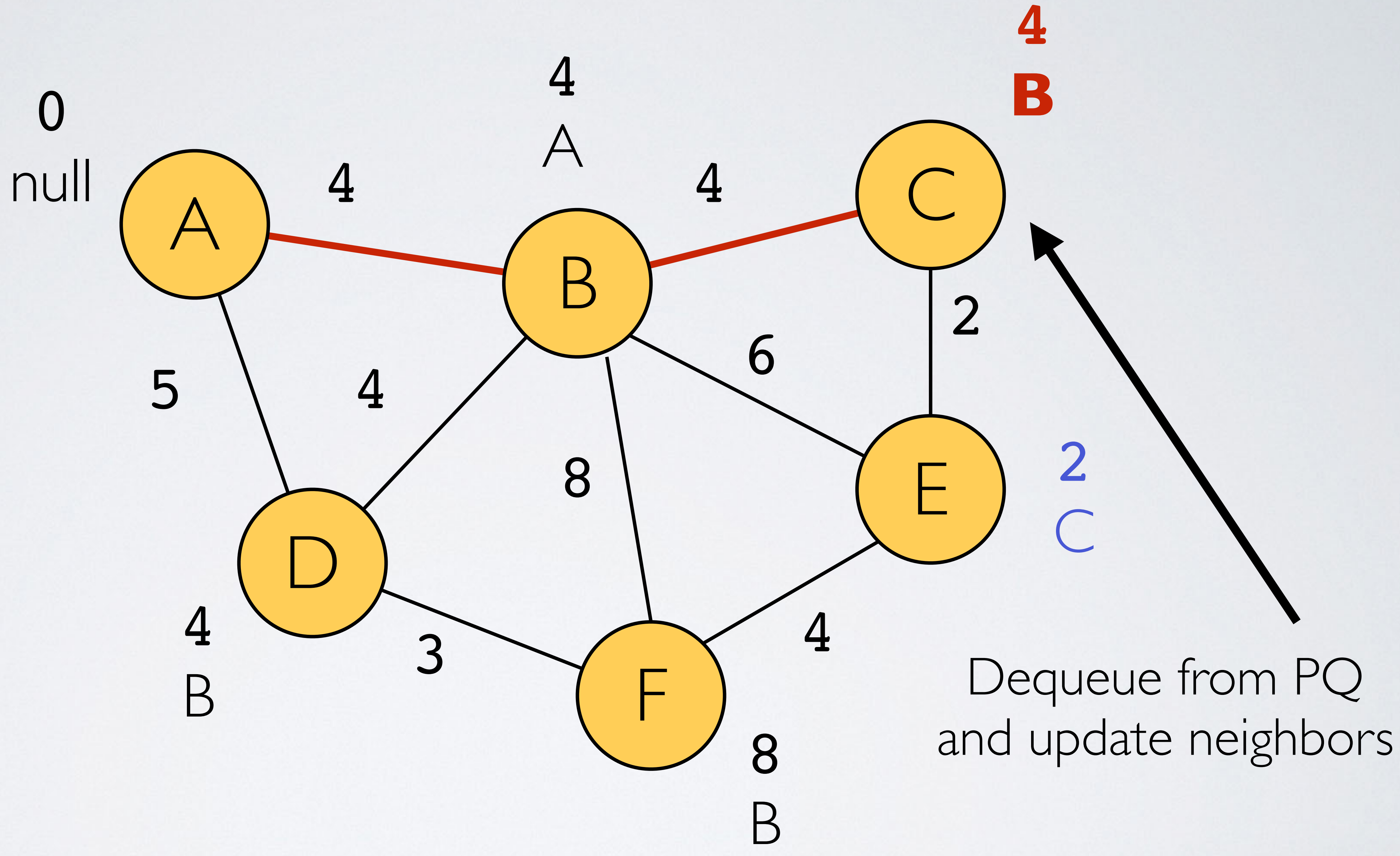
Example



Dequeue from PQ and update neighbors

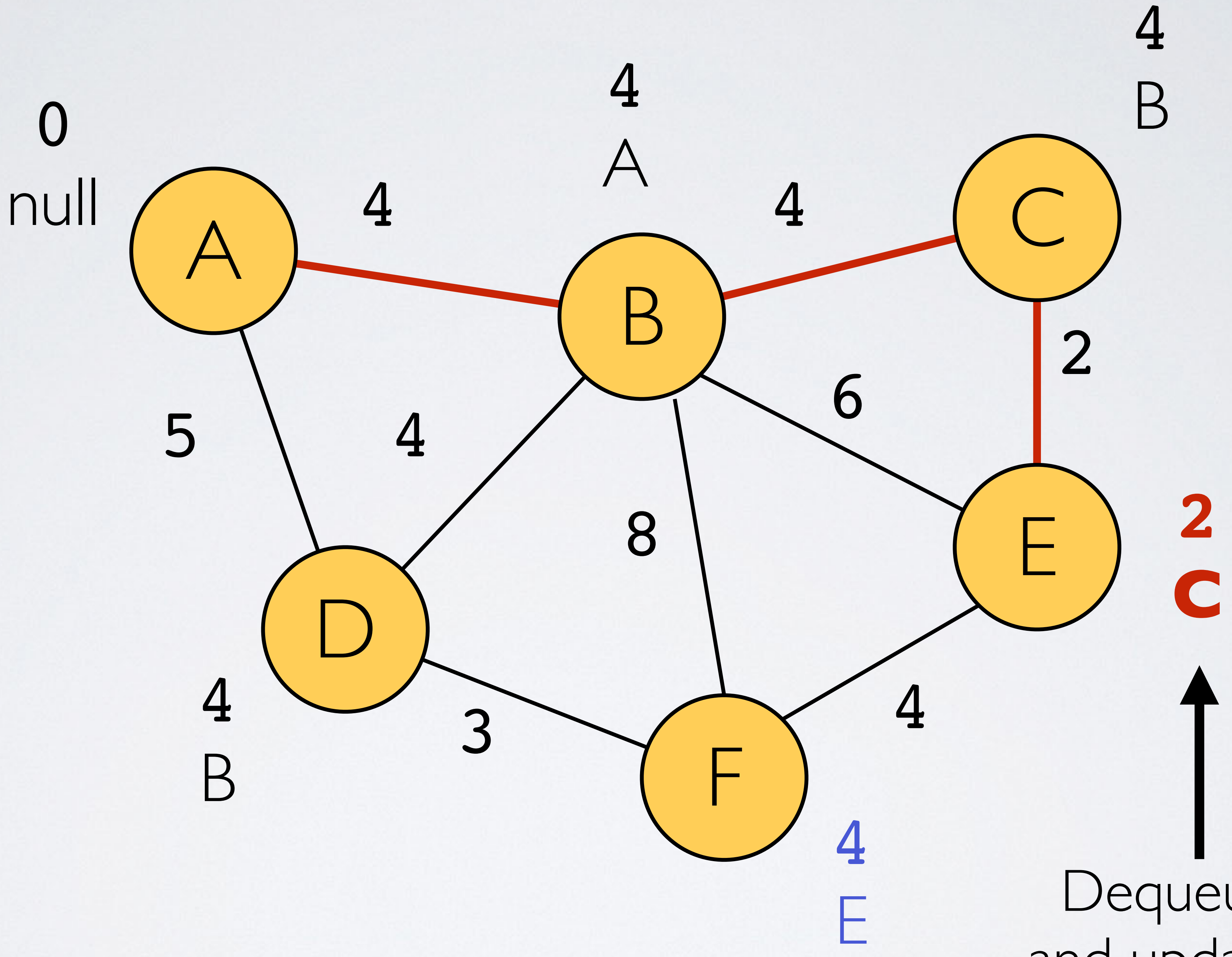
```
PQ = [(4,C),(4,D),(6,E),(8,F)]
```

Example



```
PQ = [(2,E),(4,D),(8,F)]
```

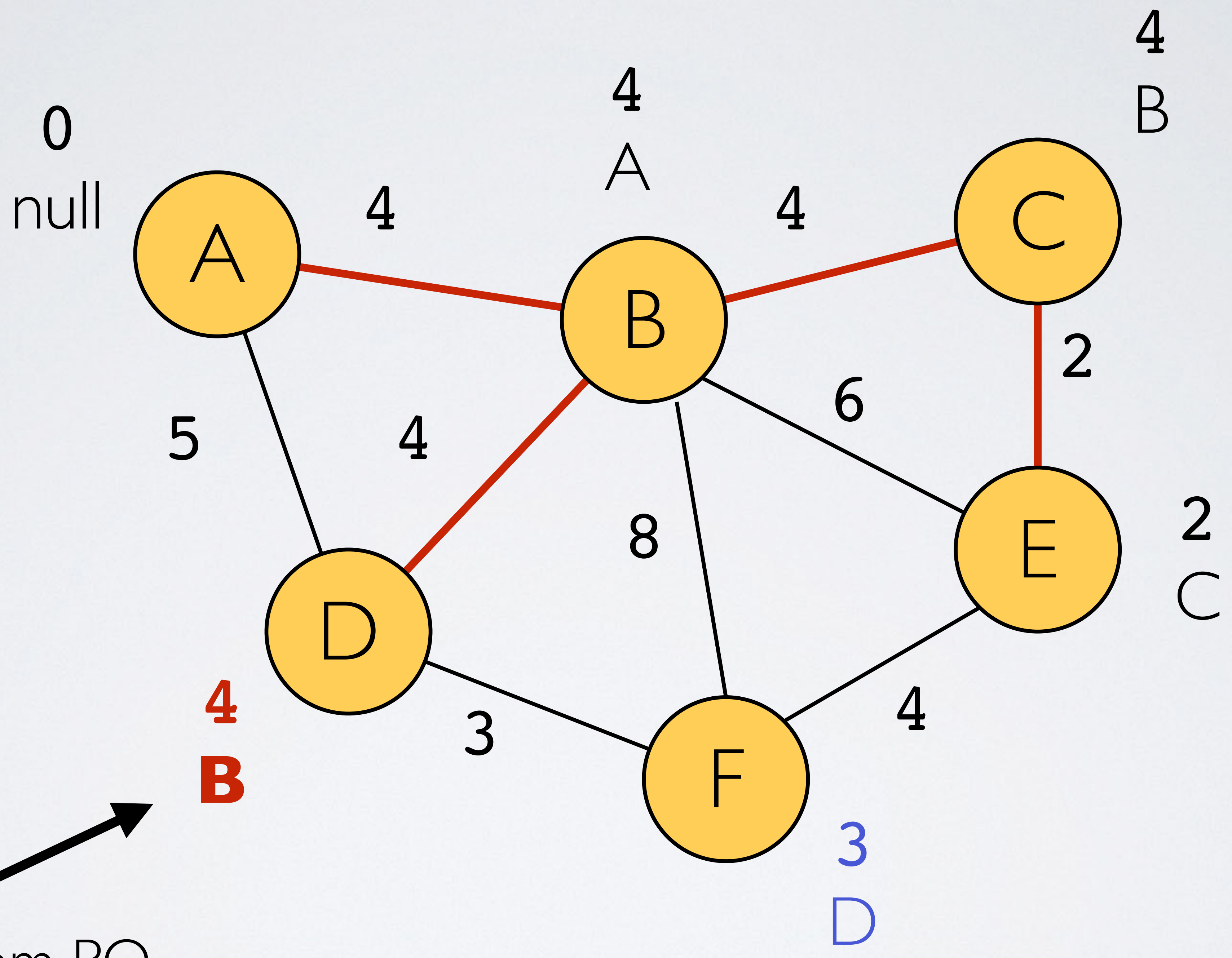
Example



Dequeue from PQ
and update neighbors

```
PQ = [(4,D),(4,F)]
```

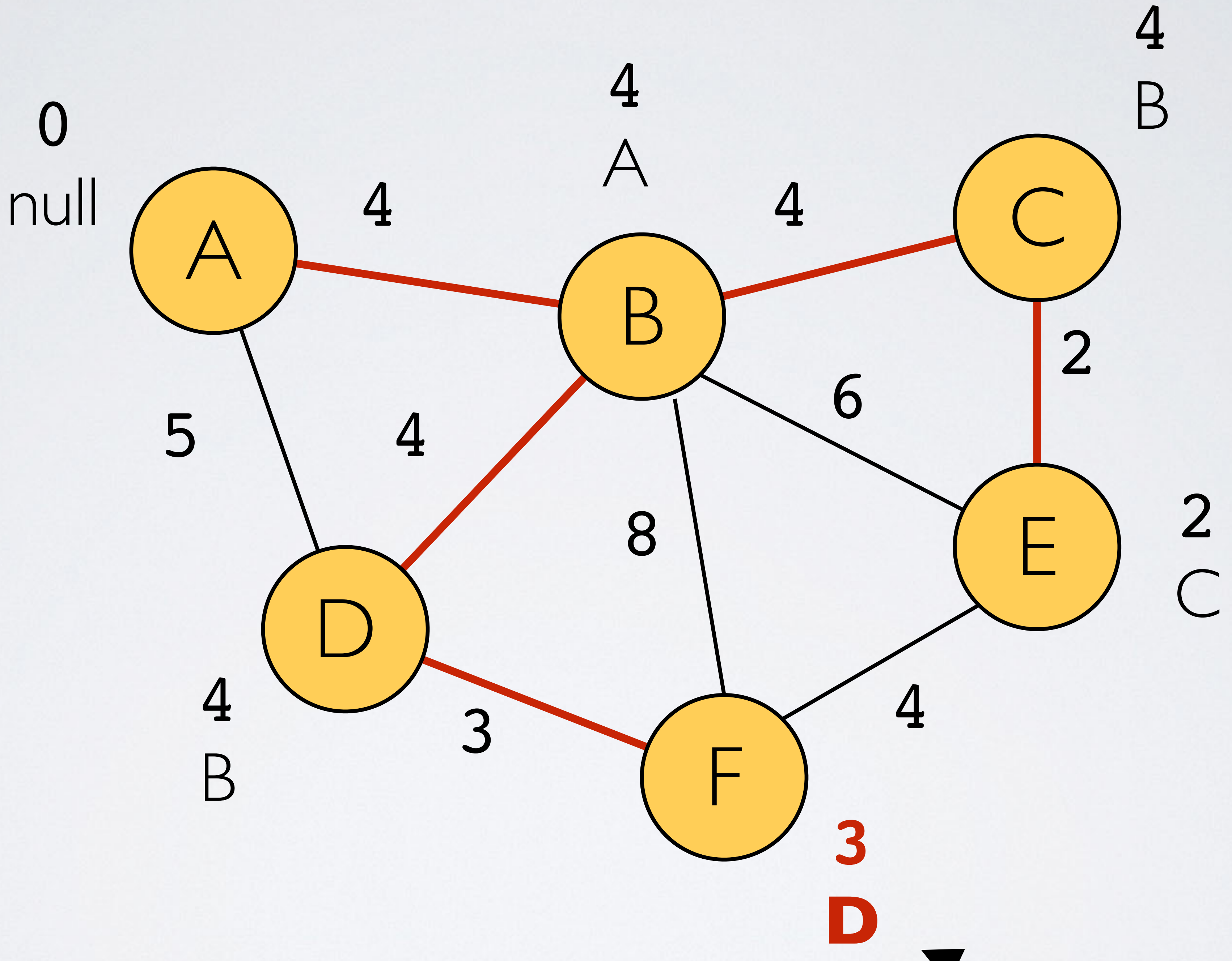
Example



Dequeue from PQ
and update neighbors

`PQ = [(3,F)]`

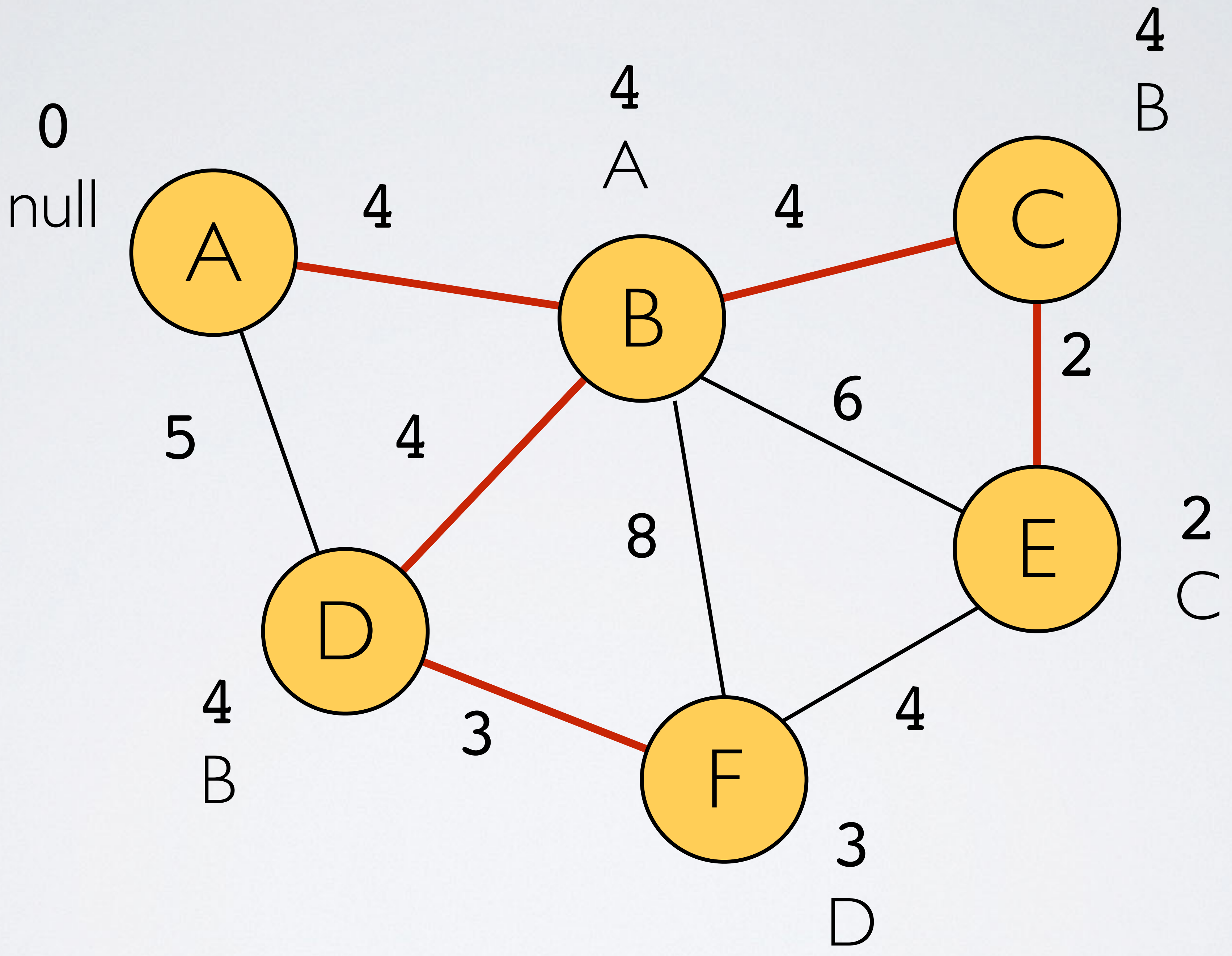
Example



`PQ = [ ]`

Dequeue from PQ and update neighbors

Example



Pseudo-code

```
function prim(G):
   // Input: weighted, undirected graph G with vertices V
   // Output: list of edges in MST
   for all v in V:
      v.cost = ∞
      v.prev = null
   s = a random v in V // pick a random source s
   s.cost = 0
   MST = []
   PQ = PriorityQueue(V) // priorities will be v.cost values
   while PQ is not empty:
      v = PQ.removeMin()
      if v.prev != null:
         MST.append((v, v.prev))
      for all incident edges (v,u) of v such that u is in PQ:
         if u.cost > (v,u).weight:
            u.cost = (v,u).weight
            u.prev = v
            PQ.decreaseKey(u, u.cost)
  return MST
```

Runtime Analysis

- Decorating nodes with distance and previous pointers is `O(|V|)`
- Putting nodes in PQ is `O(|V|log|V|)` (really `O(|V|)` since ∞ priorities)
- While loop runs `|V|` times
  - removing vertex from PQ is `O(log|V|)`
  - So `O(|V|log|V|)`
- For loop (in while loop) runs `|E|` times ***in total***
  - Replacing vertex's key in the PQ is `log|V|`
  - So `O(|E|log|V|)`
- Overall runtime
  - `O(|V| + |V|log|V| + |V|log|V| + |E|log|V|)`
    - `= O((|E| + |V|)log|V|)`

Proof of Correctness

- Common way of proving correctness of greedy algos
  - show that algorithm is always correct at every step
- Best way to do this is by induction
  - tricky part is coming up with the right invariant

Inductive invariant for Prim

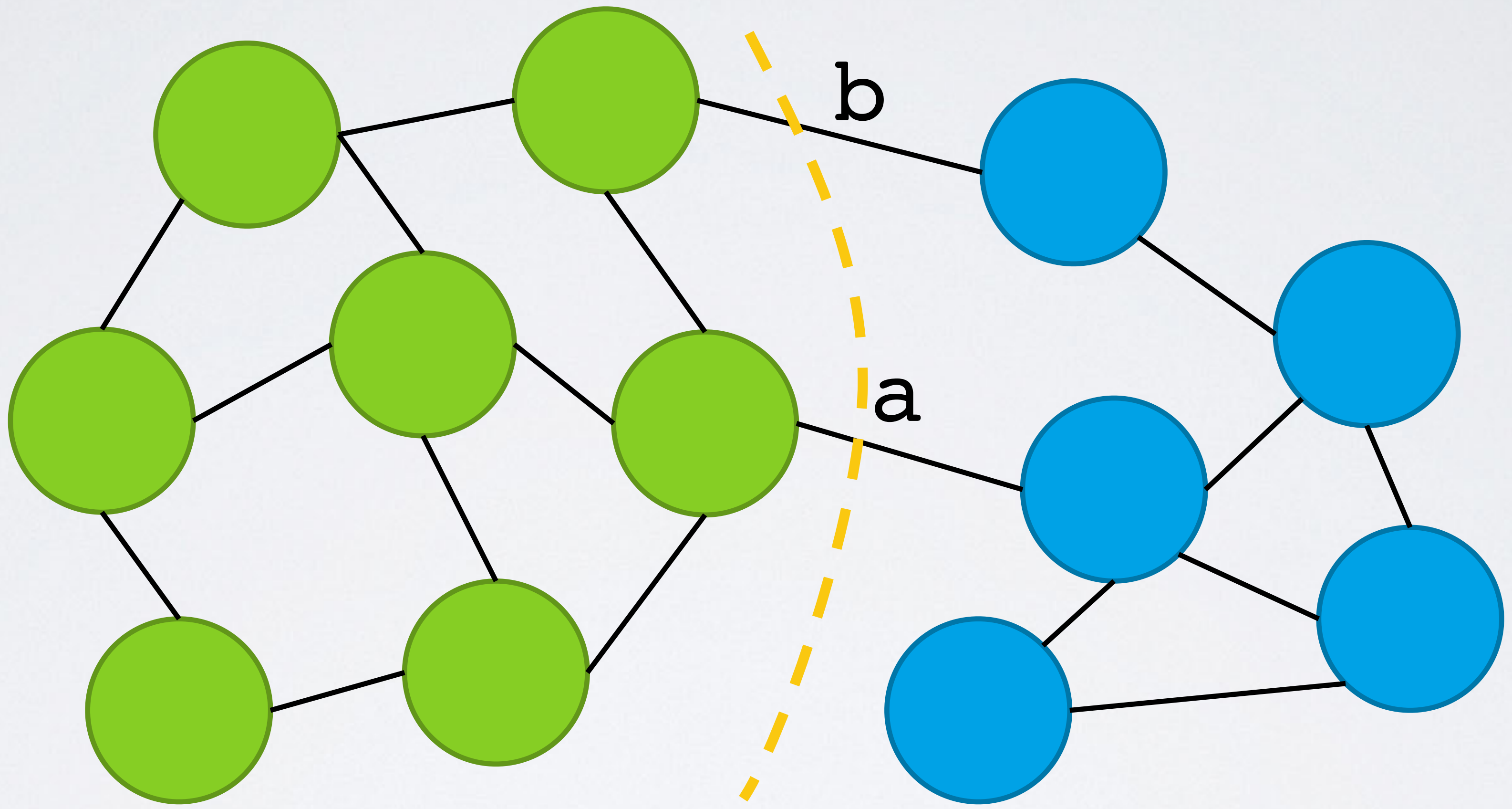
- Want an invariant **`P(n)`**, where **`n`** is number of edges added so far
- Need to have:
  - **`P(0)`** *[base case]*
  - **`P(n)`** implies **`P(n + 1)`** *[inductive case]*
  - **`P(size of MST)`** implies correctness

Inductive invariant for Prim

- Want an invariant **`P(n)`**, where **`n`** is number of edges added so far
- Need to have:
  - **`P(0)`** *[base case]*
  - **`P(n)`** implies **`P(n + 1)`** *[inductive case]*
  - **`P(size of MST)`** implies correctness
- **`P(n)=`** first **`n`** edges added by Prim are a subtree of *some* MST

Graph Cuts

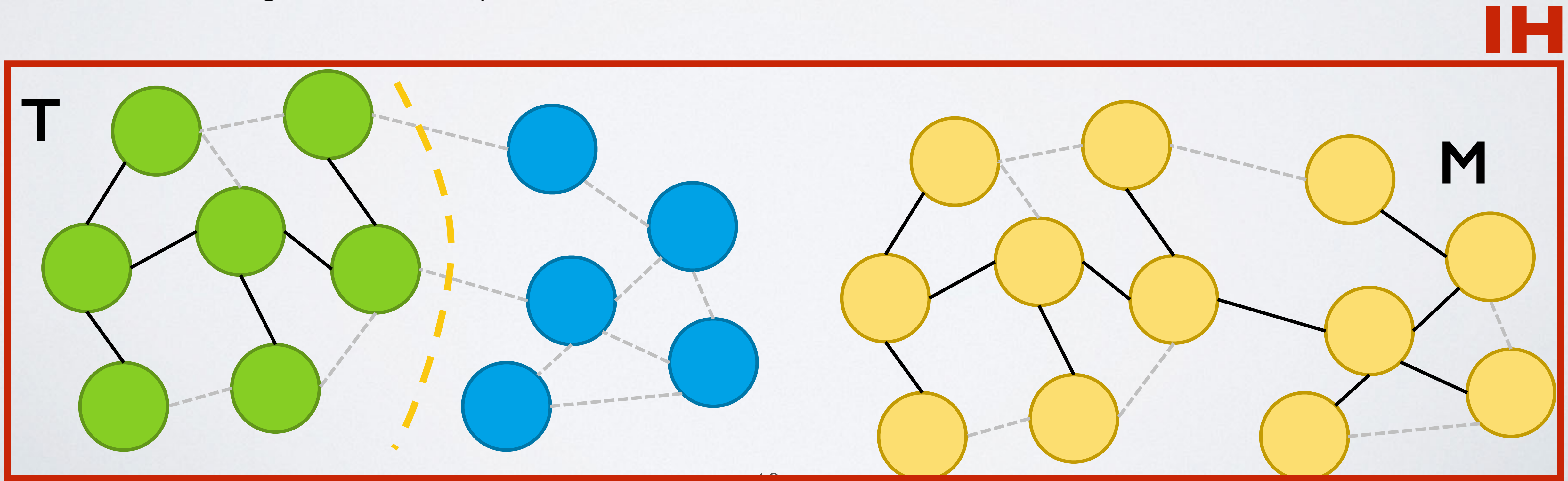
- A cut is any partition of the vertices into two groups



- Here **G** is partitioned in 2
  - with edges **b** and **a** joining the partitions

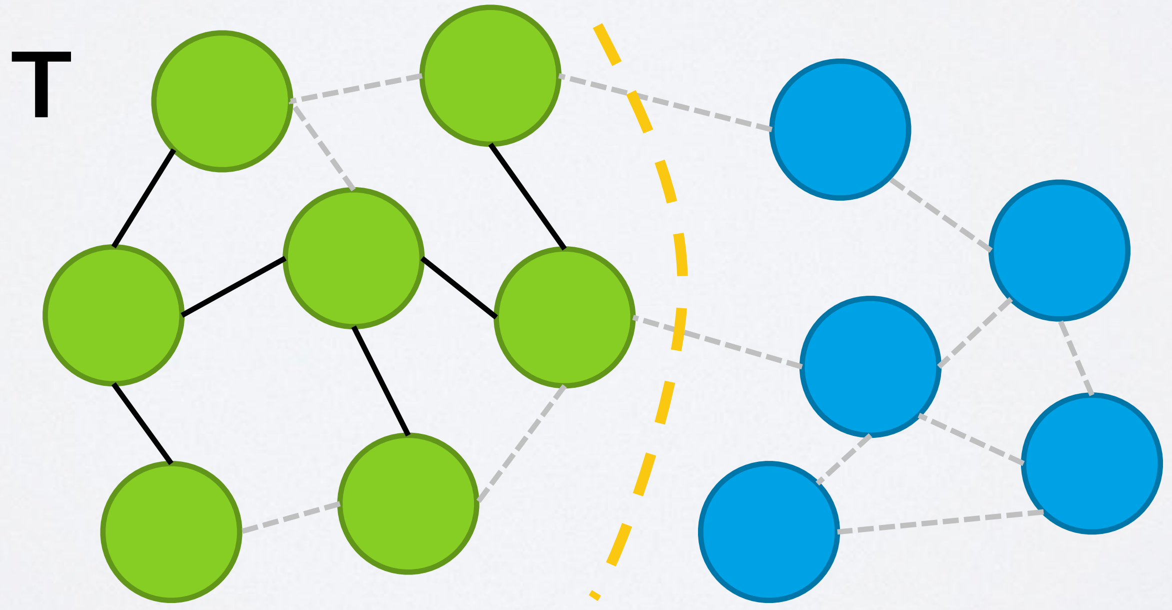
Proof of Correctness

- `P(n)`
  - first `n` edges added by Prim are a subtree of some MST
- Base case when `n=0`
  - no edges have been added yet so `P(0)` is trivially true
- Inductive Hypothesis
  - first `k` edges added by Prim form a tree `T` which is subtree of some MST `M`



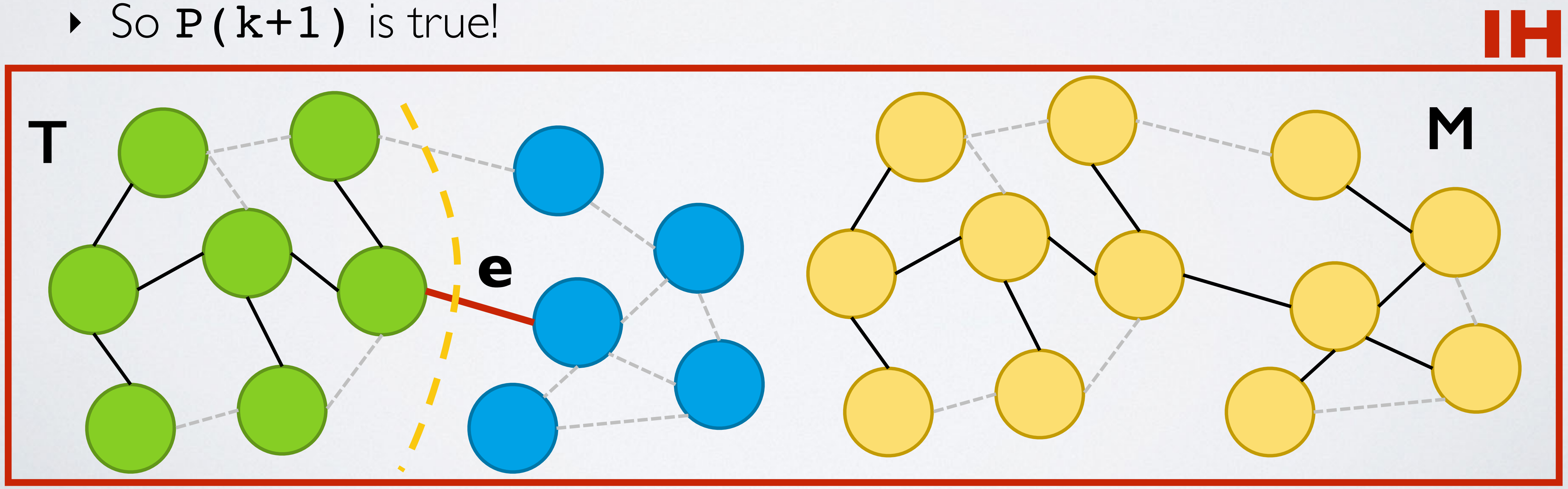
Proof of Correctness

- Inductive Step
  - Let **`e`** be the **`(k+1)`**th edge that is added
  - **`e`** will connect **`T`** (green nodes) to an unvisited node (one of blue nodes)
  - We need to show that adding **`e`** to **`T`**
    - forms a subtree of some MST **`M'`**
    - (which may or may not be the same MST as **`M`**)



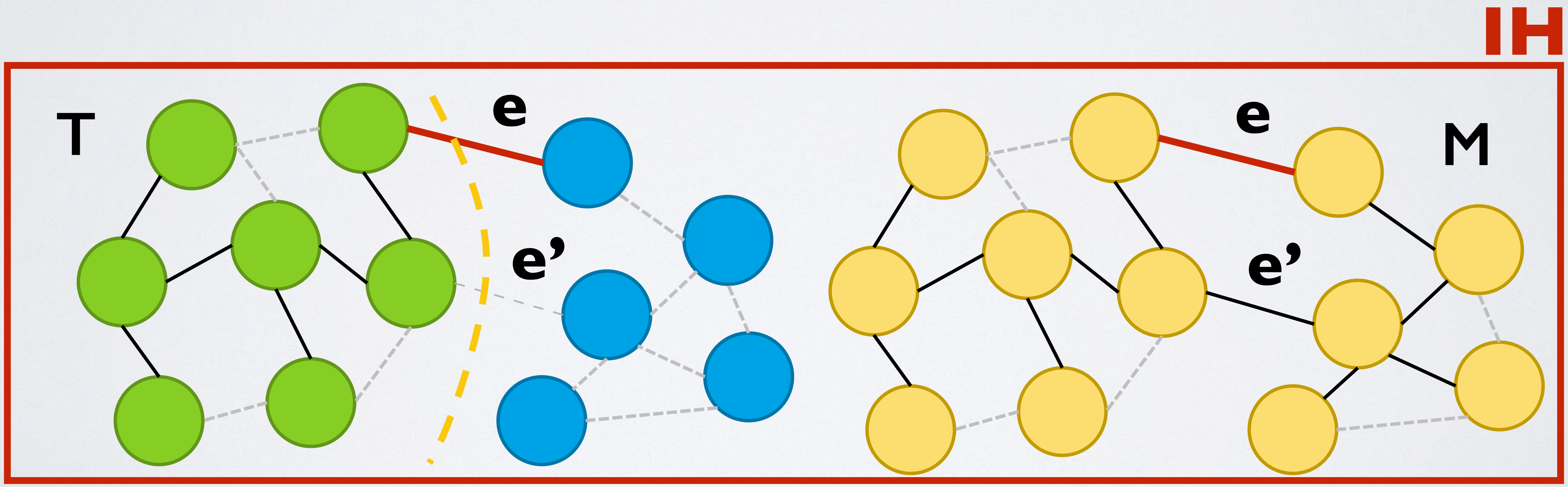
Proof of Correctness

- Two cases
  - `e` is in original MST `M`
  - `e` is not in `M`
- Case `1`: `e` is in `M`
  - there exists an MST that contains first `k+1` edges
  - So `P(k+1)` is true!



Proof of Correctness

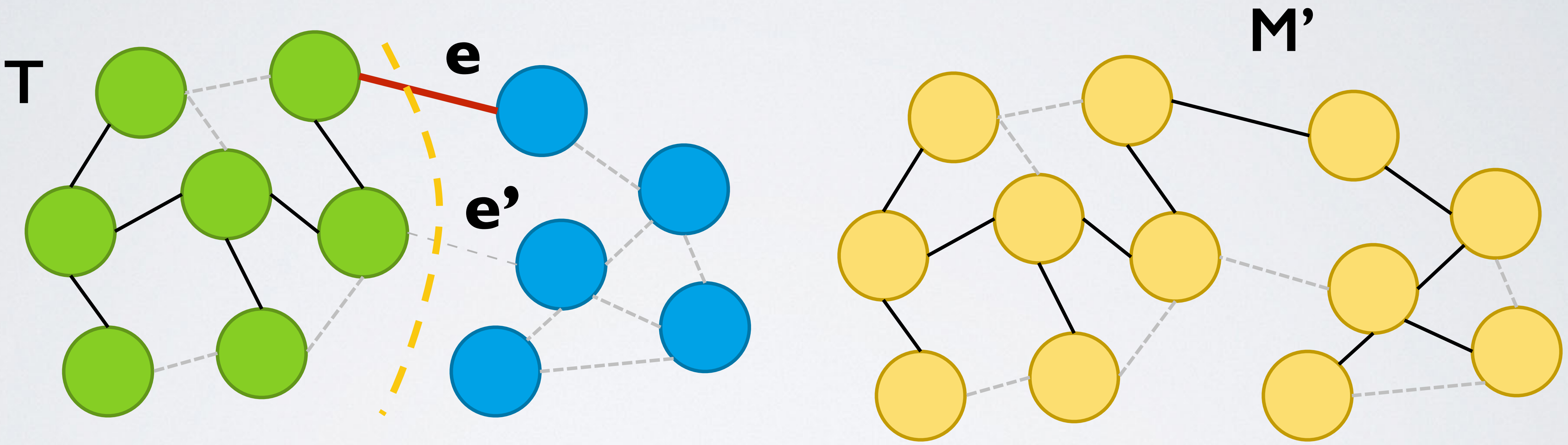
- Case `2`: `e` is not in `M`
  - if we add `e=(u,v)` to `M` then we get a cycle
  - why? since `M` is span. tree there must be path from `u` to `v` w/o `e`
  - so there must be another edge `e’` that connects `T` to unvisited nodes



  - We know `e.weight ≤ e’.weight` because Prim chose `e` first

Proof of Correctness

- So if we add **`e`** to **`M`** and remove **`e'`**
  - we get a new MST **`M'`** that is no larger than **`M`** and contains **`T`** & **`e`**



- **`P(k+1)`** is true
  - because **`M'`** is an MST that contains the first **`k+1`** edges added by Prim's

Proof of Correctness

- Since we have shown
  - `P(0)` is true
  - `P(k+1)` is true assuming `P(k)` is true (for both cases)
  - The first `n` edges added by Prim form a subtree of some MST

Readings

- Dasgupta Section 5.1
  - Explanations of MSTs
  - algorithms discussed in this lecture and next lecture