Minimum Spanning Trees: Kruskal

CS16: Introduction to Data Structures & Algorithms

Summer 2021

Review: Prim-Jarnik

```
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
```

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

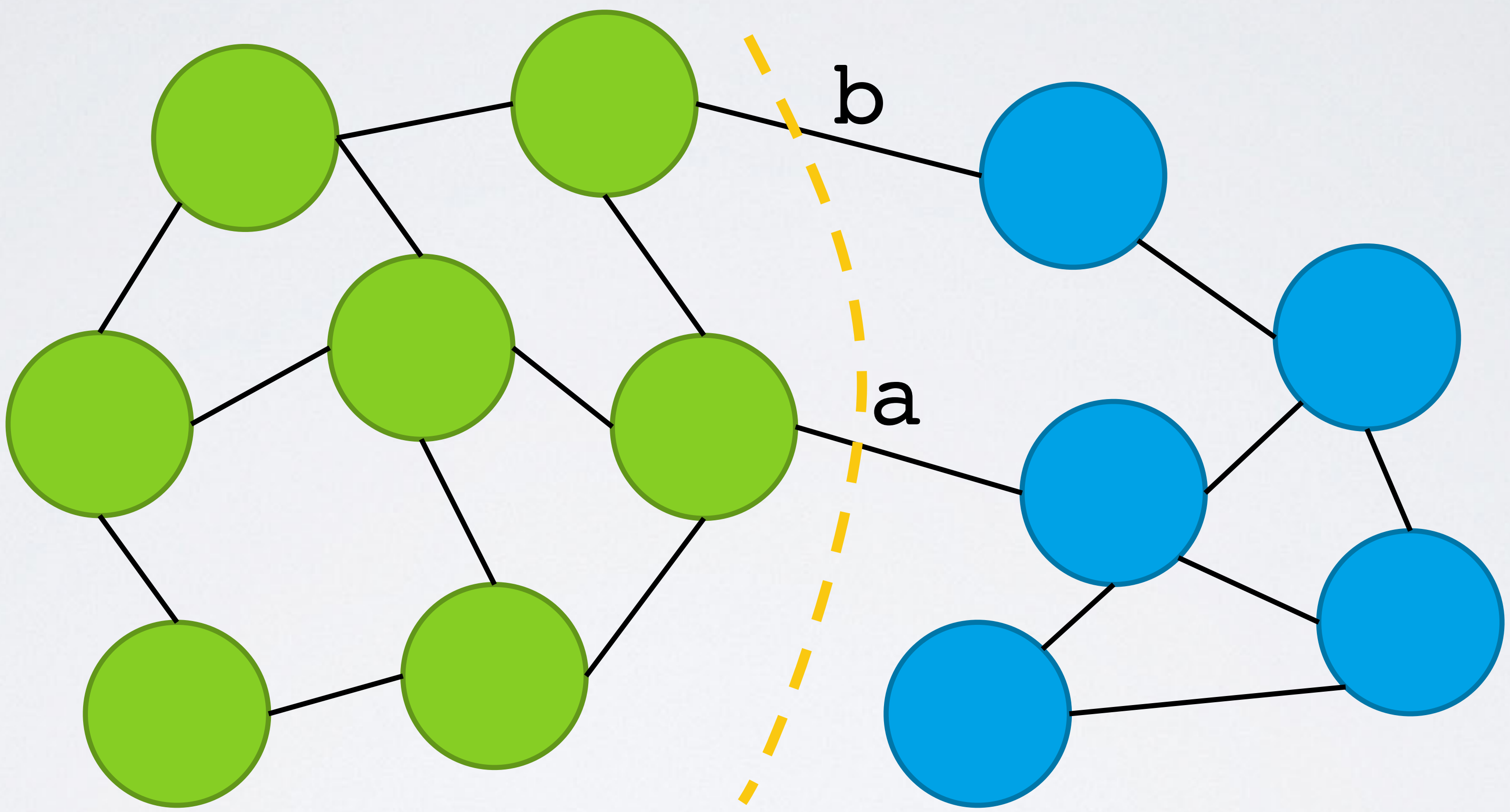
- Want an invariant **`P(n)`**, where **`n`** is number of edges added so far
- Need to have:
  - **`P(0)`** *[base case]*
  - **`P(n)`** implies **`P(n + 1)`** *[inductive case]*
  - **`P(size of MST)`** implies correctness

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

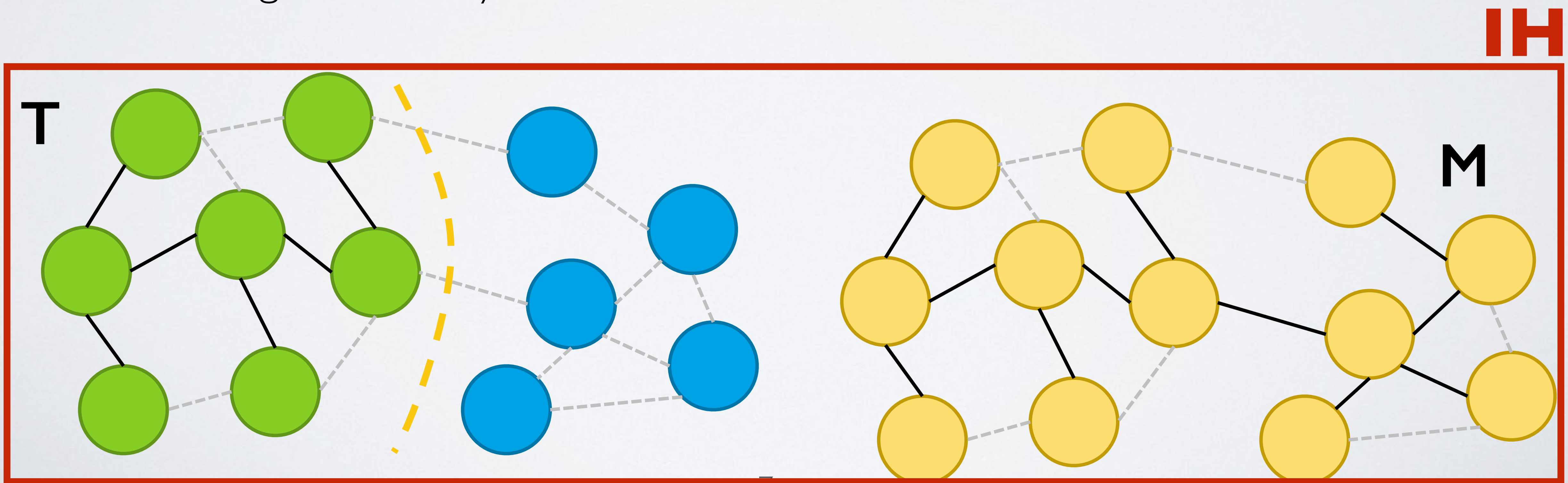
- A cut is any partition of the vertices into two groups



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

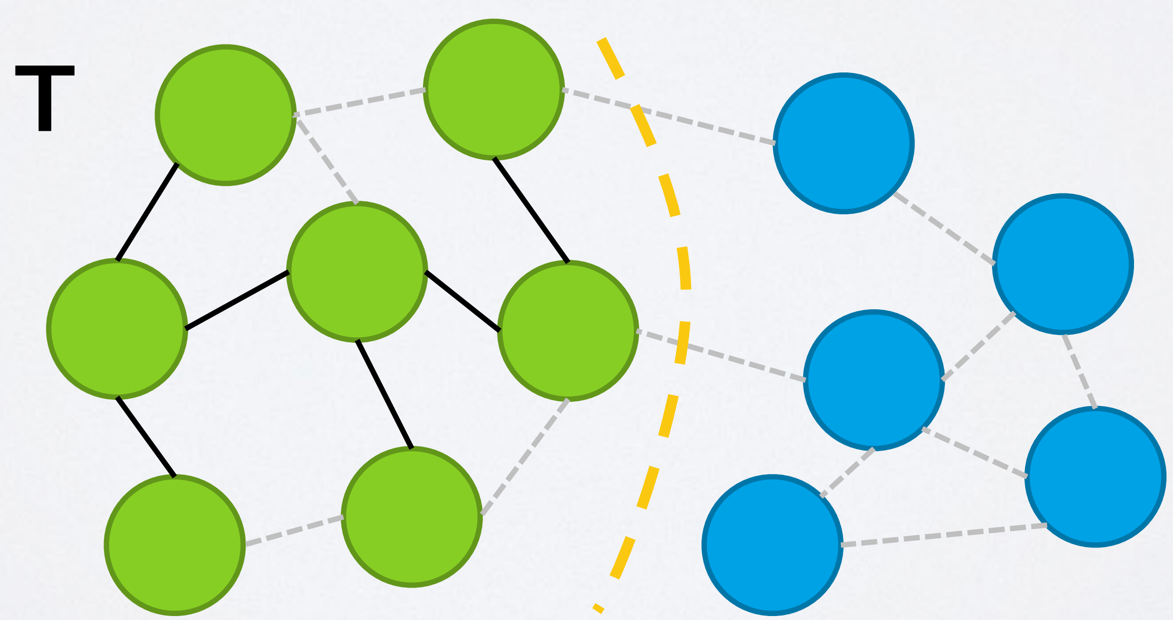
Proof of Correctness

- `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`



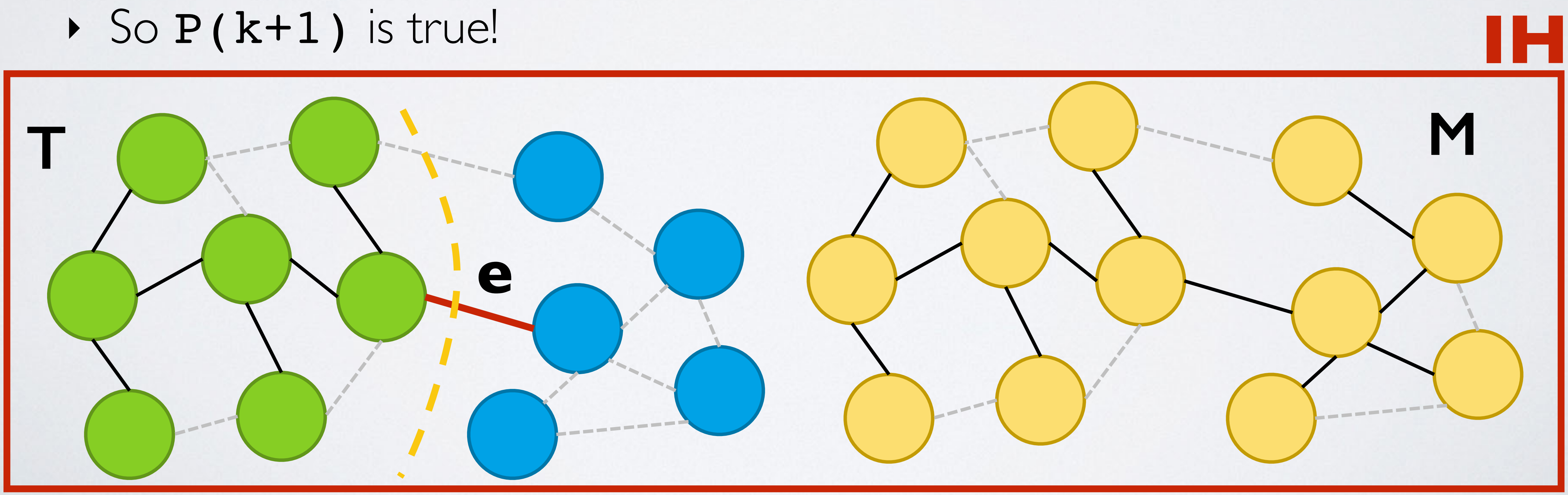
Proof of Correctness

- 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`)



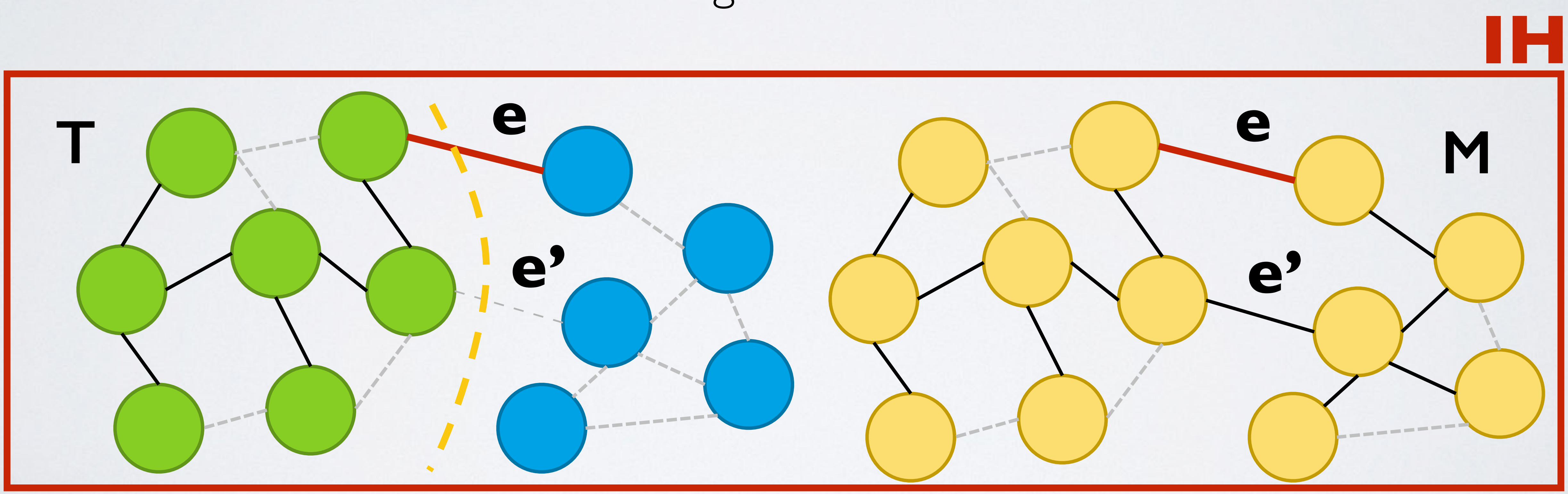
Proof of Correctness

- 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

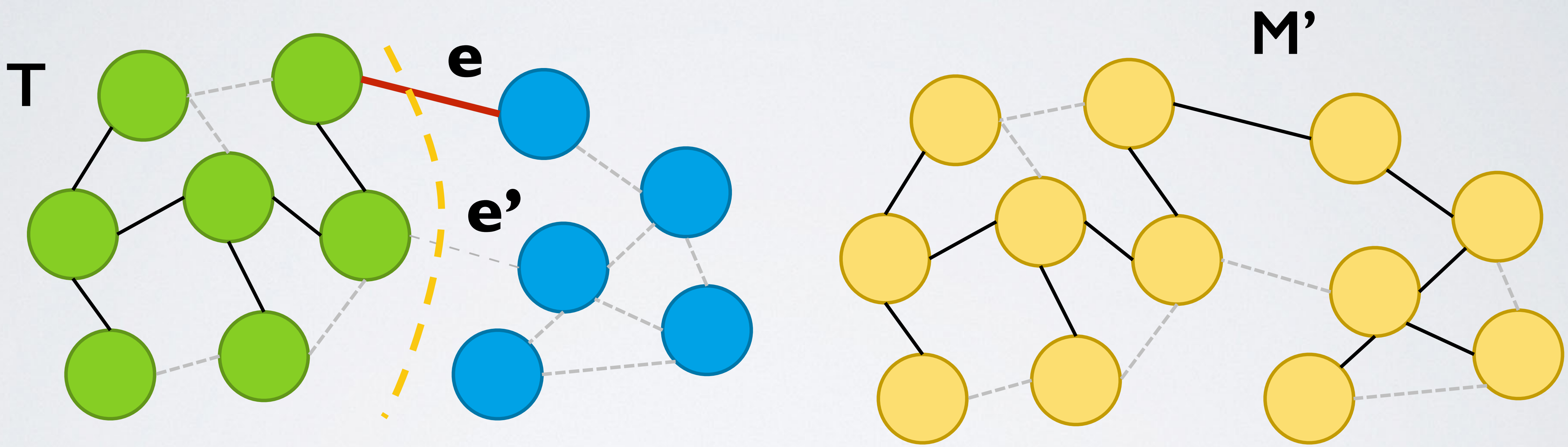
- Case `2`: **`e`** is not in **`M`**
  - if we add **`e=(u,v)`** to **`M`** then we get a cycle
  - why? since **`M`** is span. tree there must be path from **`u`** to **`v`** w/o **`e`**
  - so there must be another edge **`e'`** that connects **`T`** to unvisited nodes



  - 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

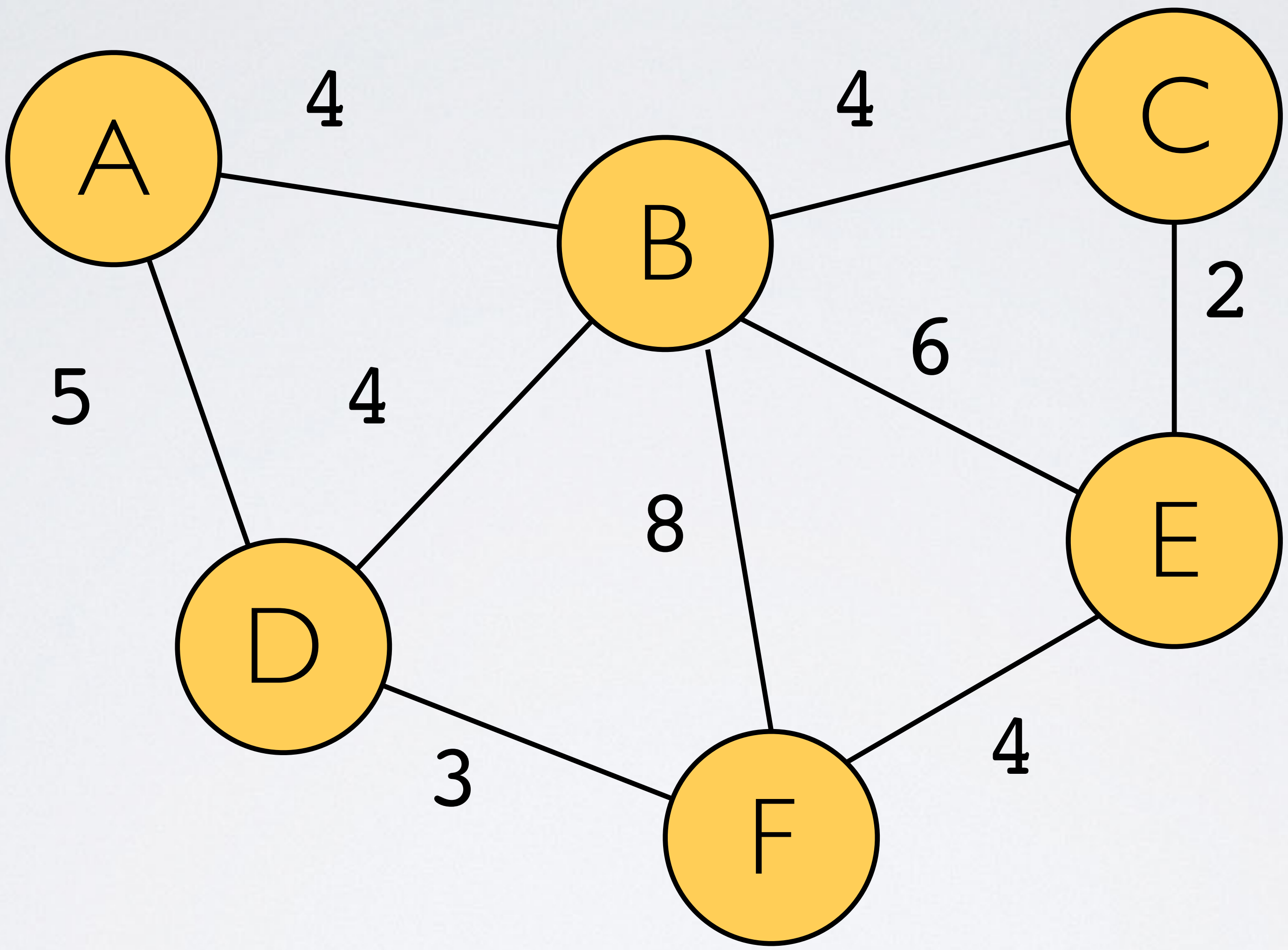
Proof of Correctness

- 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

Kruskal's Algorithm

- Sort edges by weight in ascending order
- For each edge in sorted list
  - If adding edge does not create cycle…
  - …add it to MST
- Stop when you have gone through all edges

Example

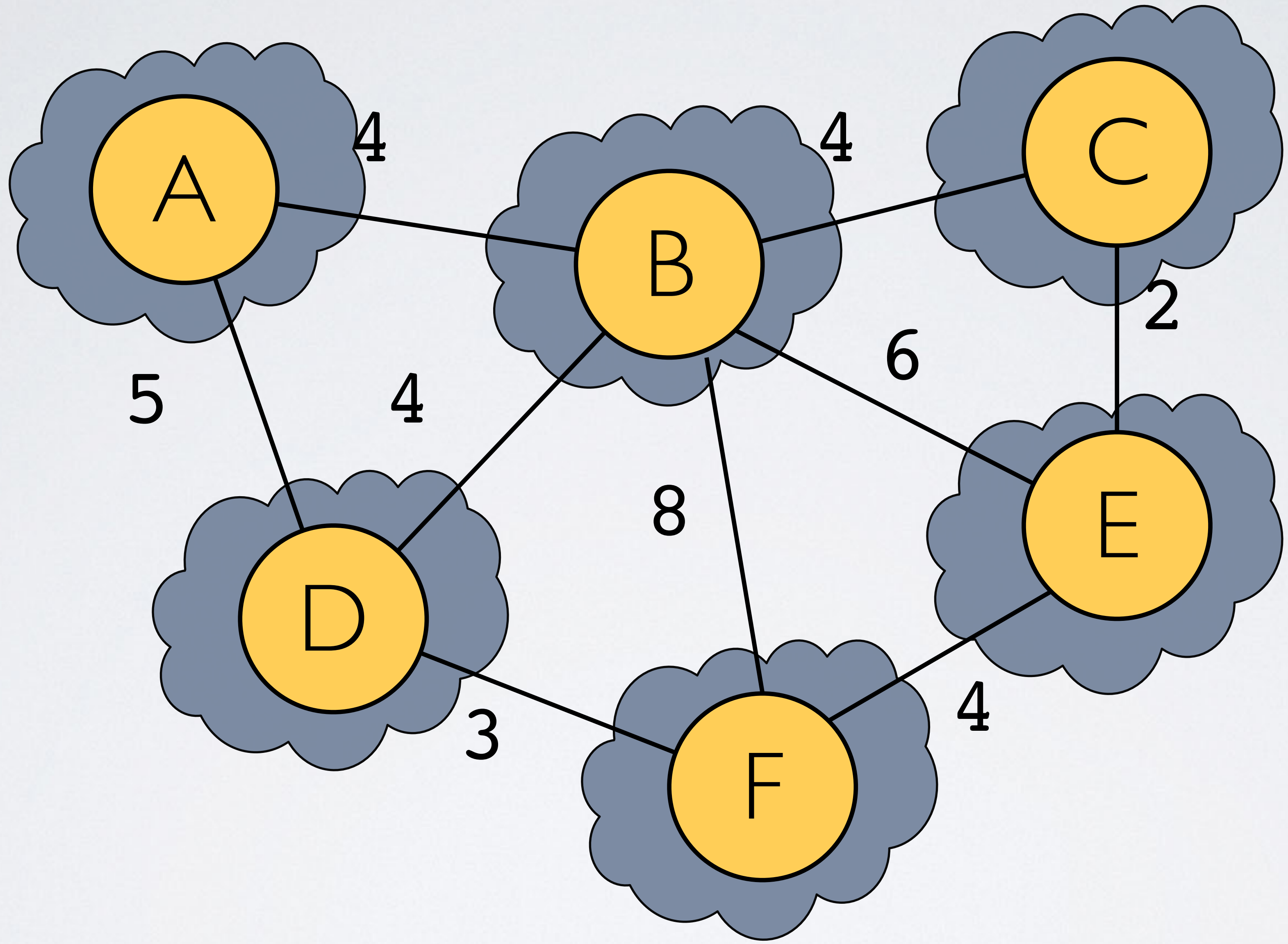



```
edges = [(C,E),(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Kruskal

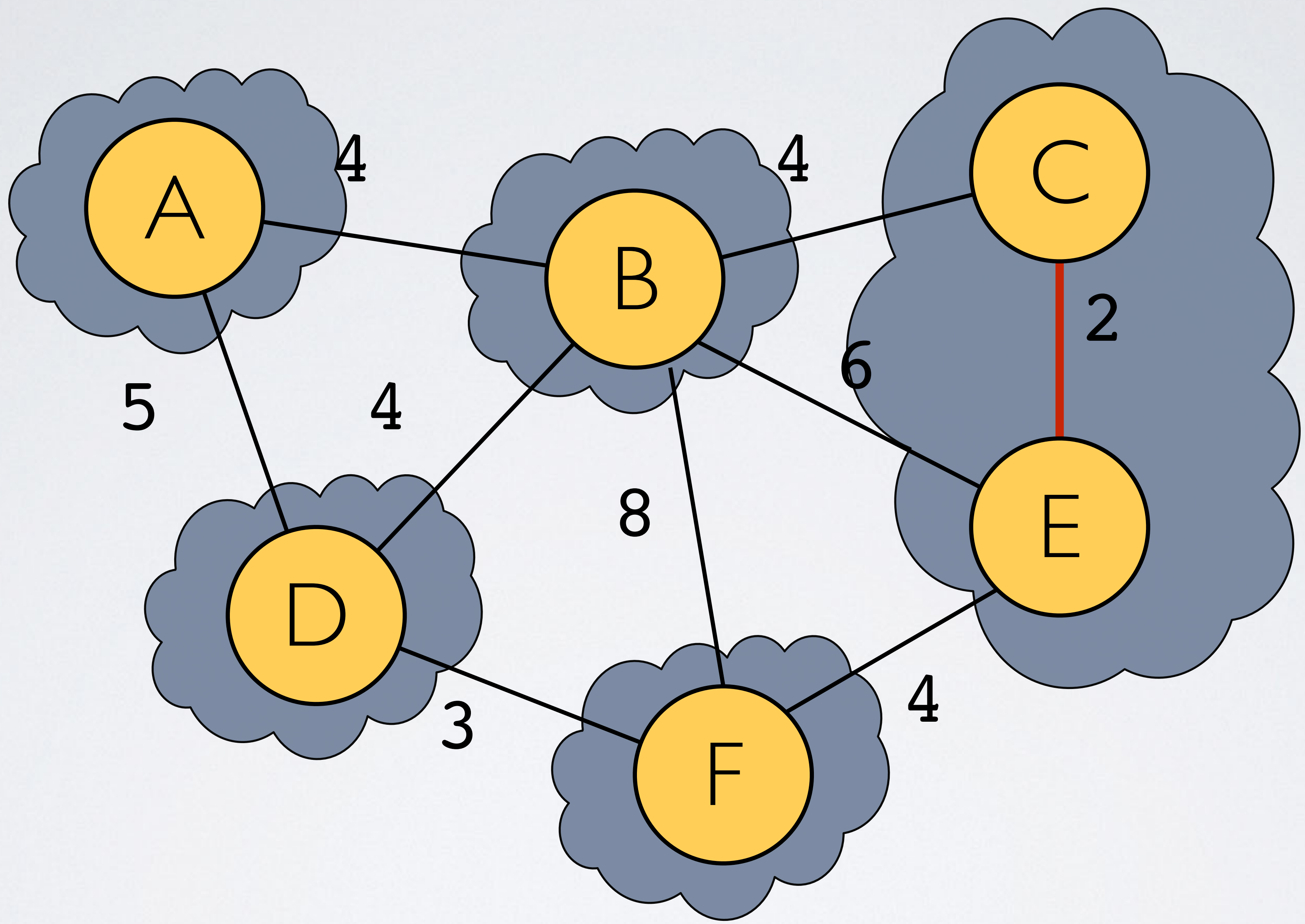
- How can we tell if adding edge will create cycle?
- Start by giving each vertex its own "cloud"
- If both ends of lowest-cost edge are in same cloud
  - we know that adding the edge will create a cycle!
- When edge is added to MST
  - merge clouds of the endpoints

Example



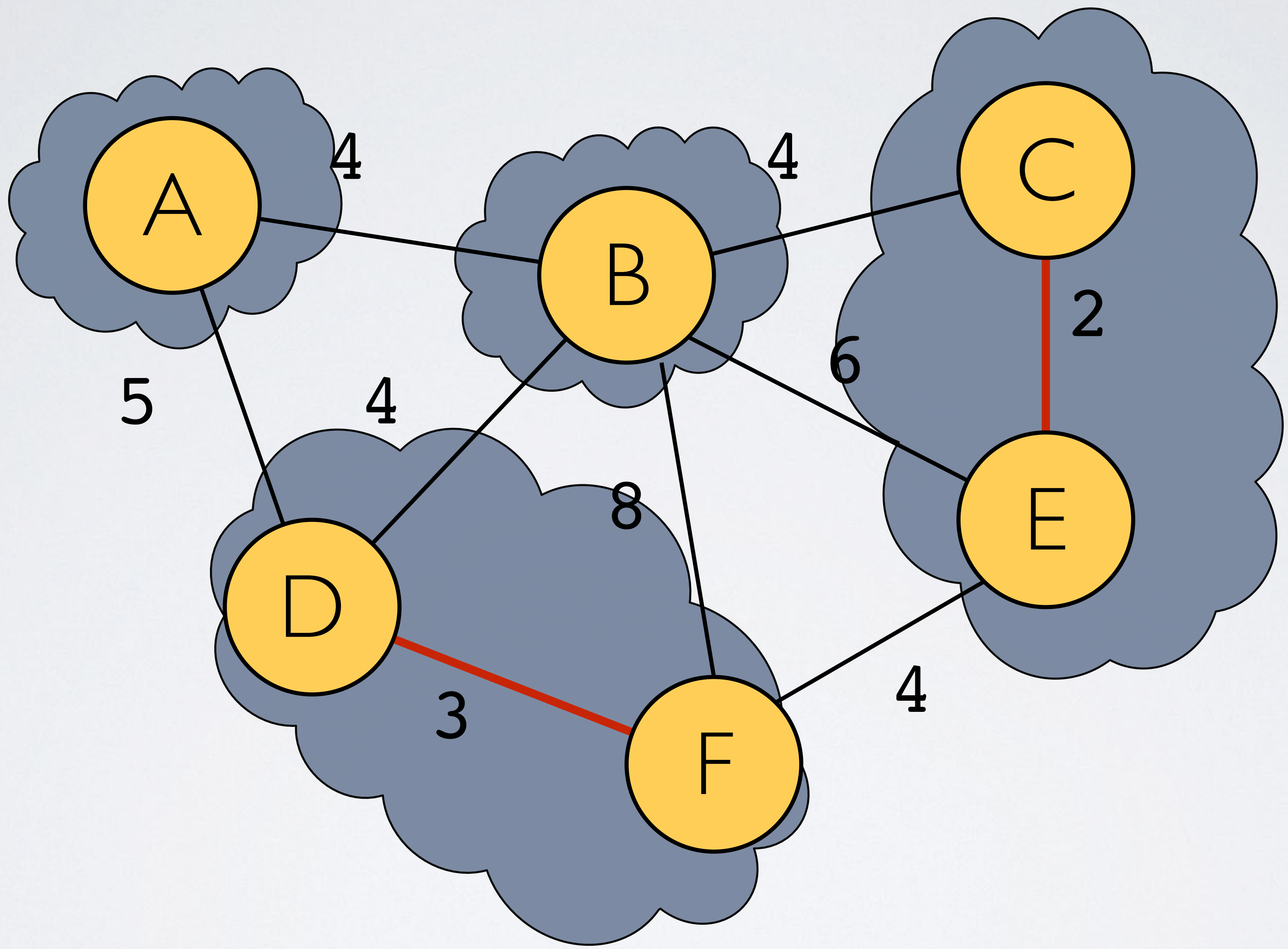

```
edges = [(C,E),(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



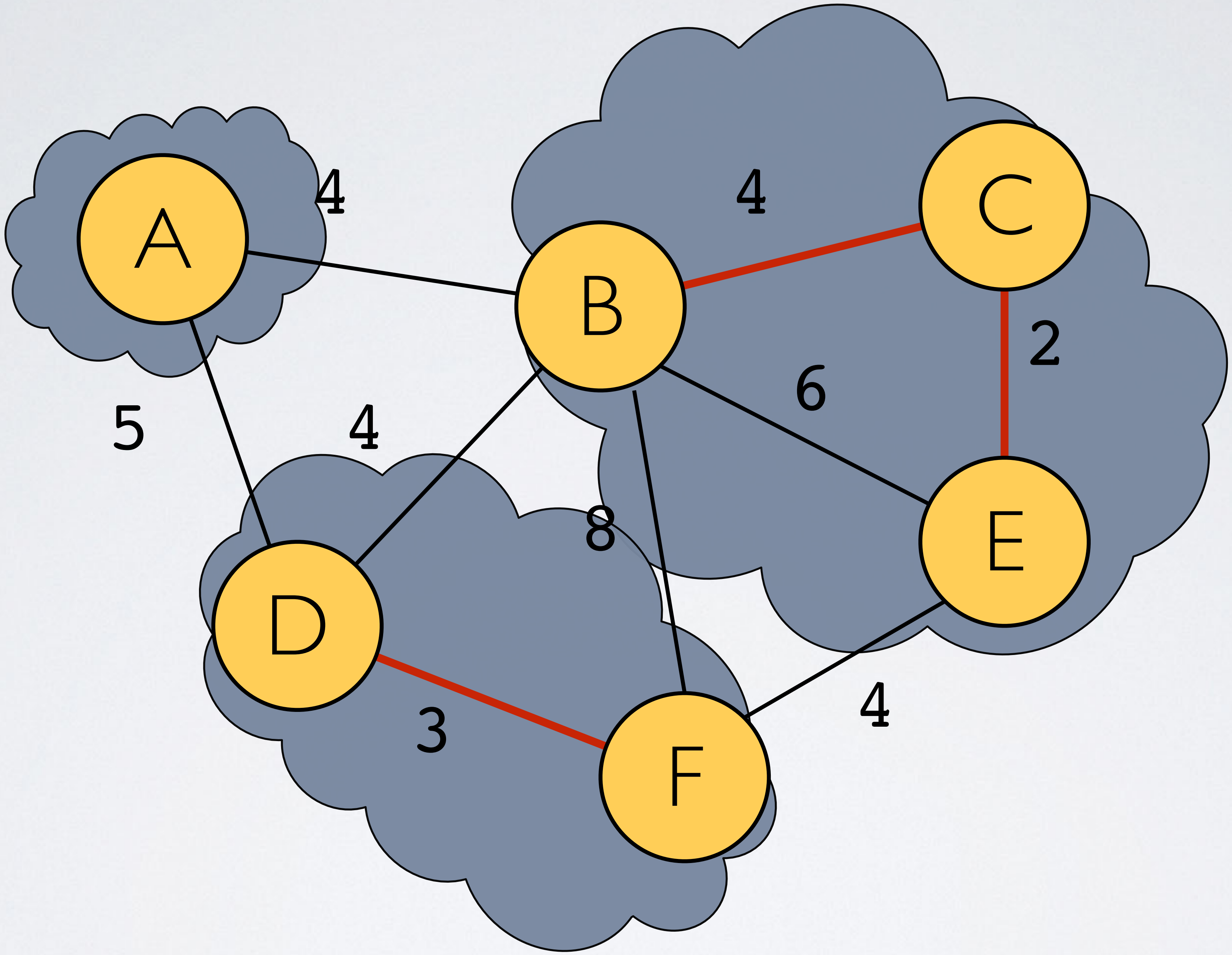

```
edges = [(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



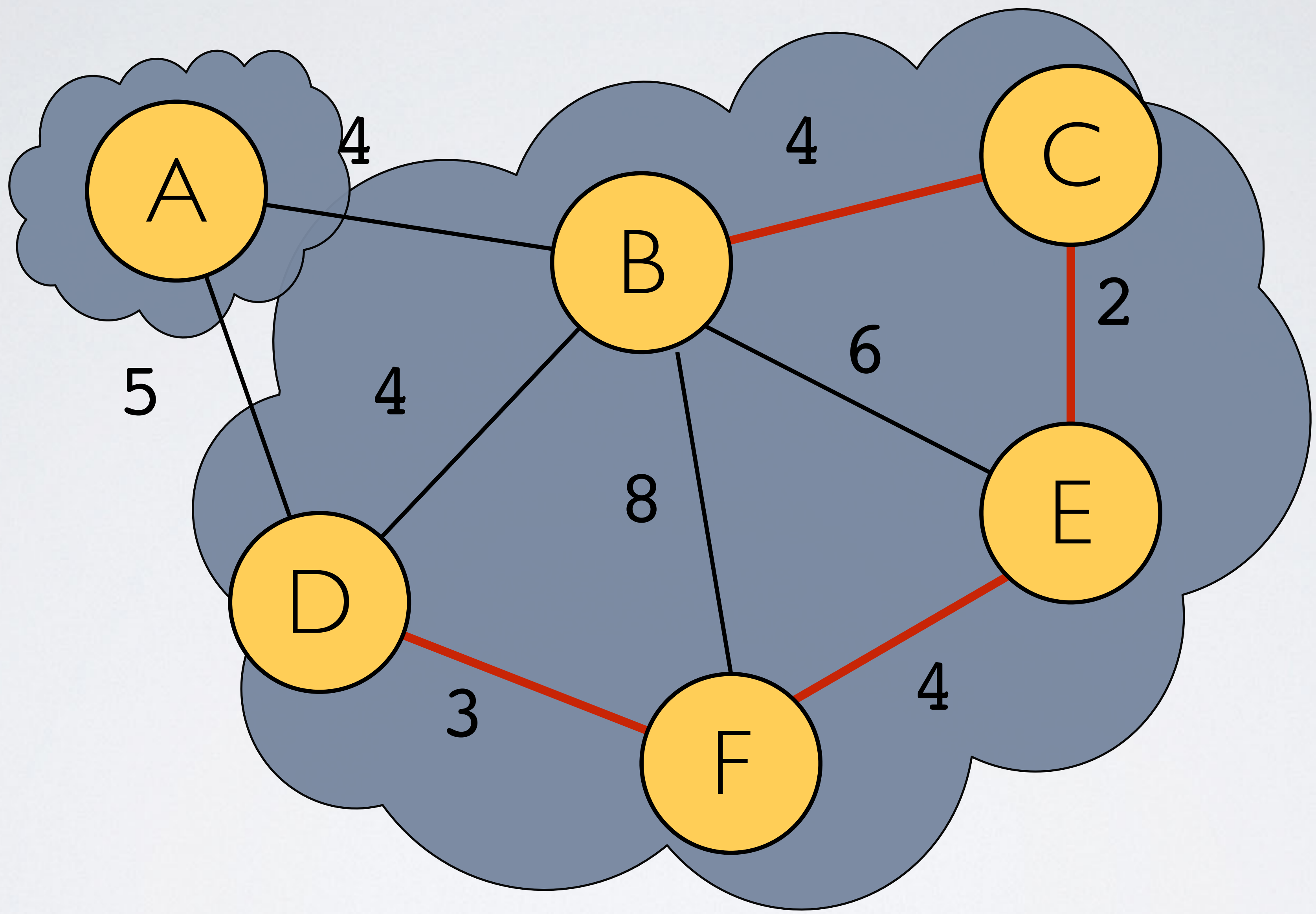

```
edges = [(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



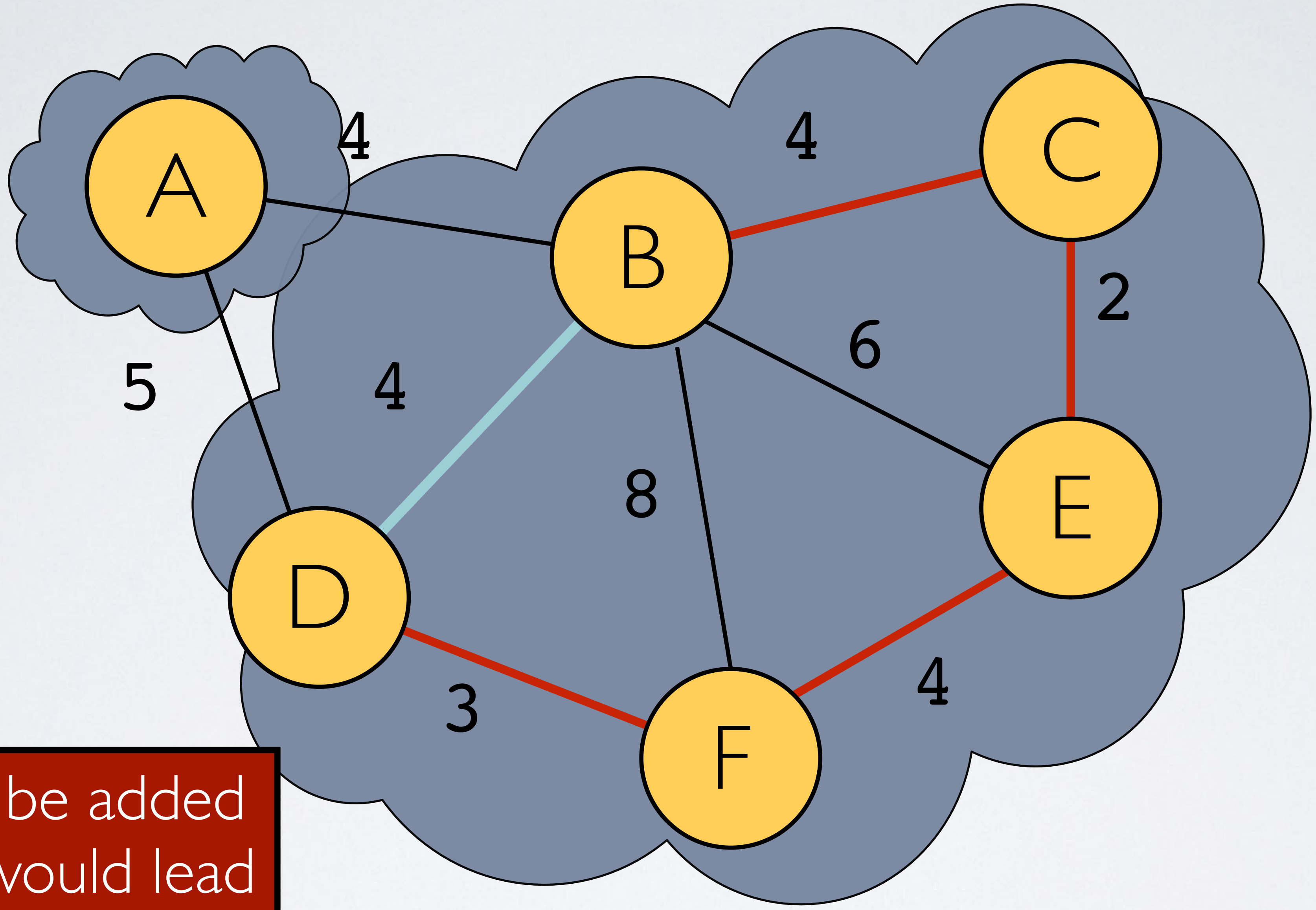
`edges = [(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]`

Example




```
edges = [(B,D),(A,B),(A,D),(B,E),(B,F)]
```

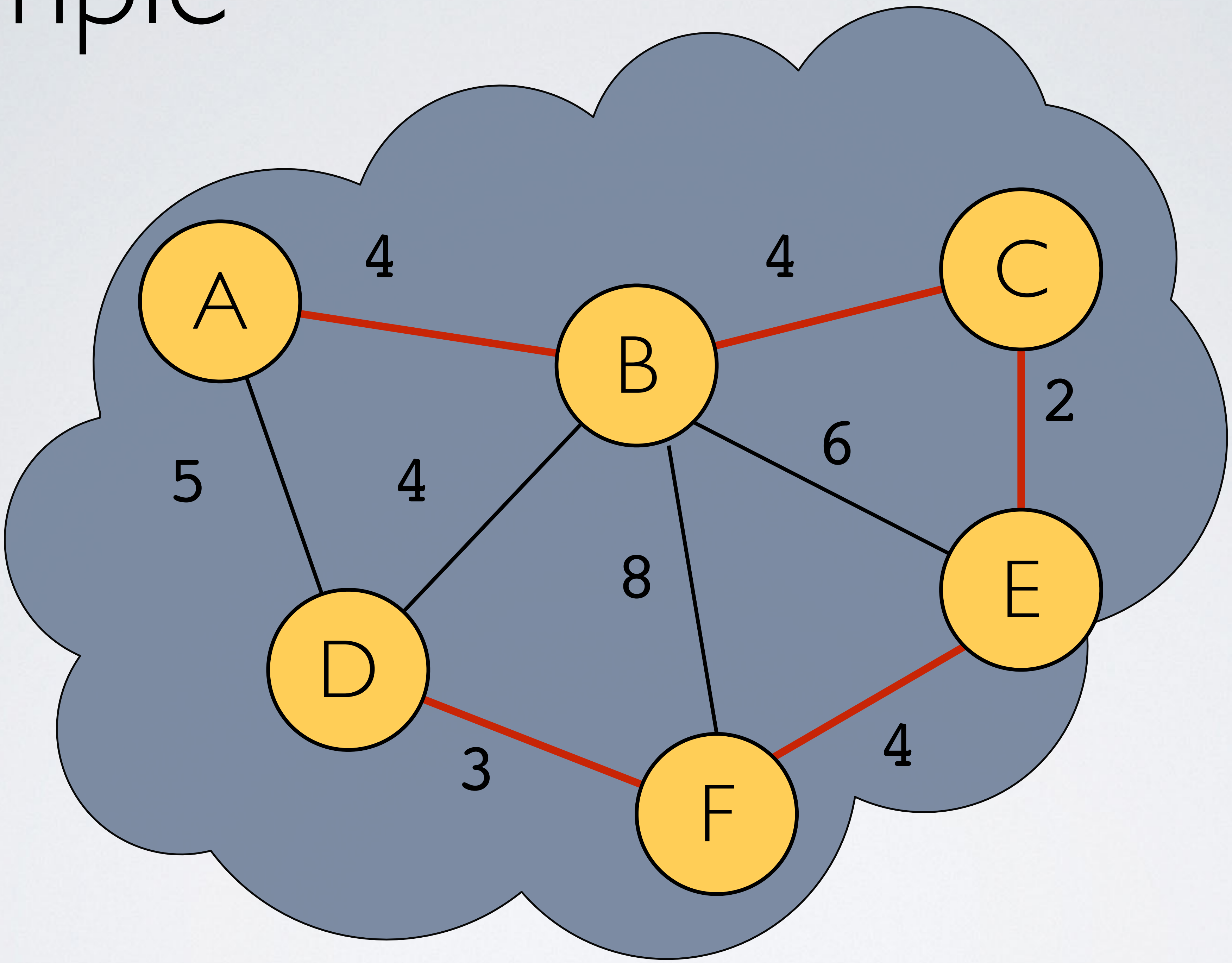
Example



BD cannot be added because it would lead to a cycle

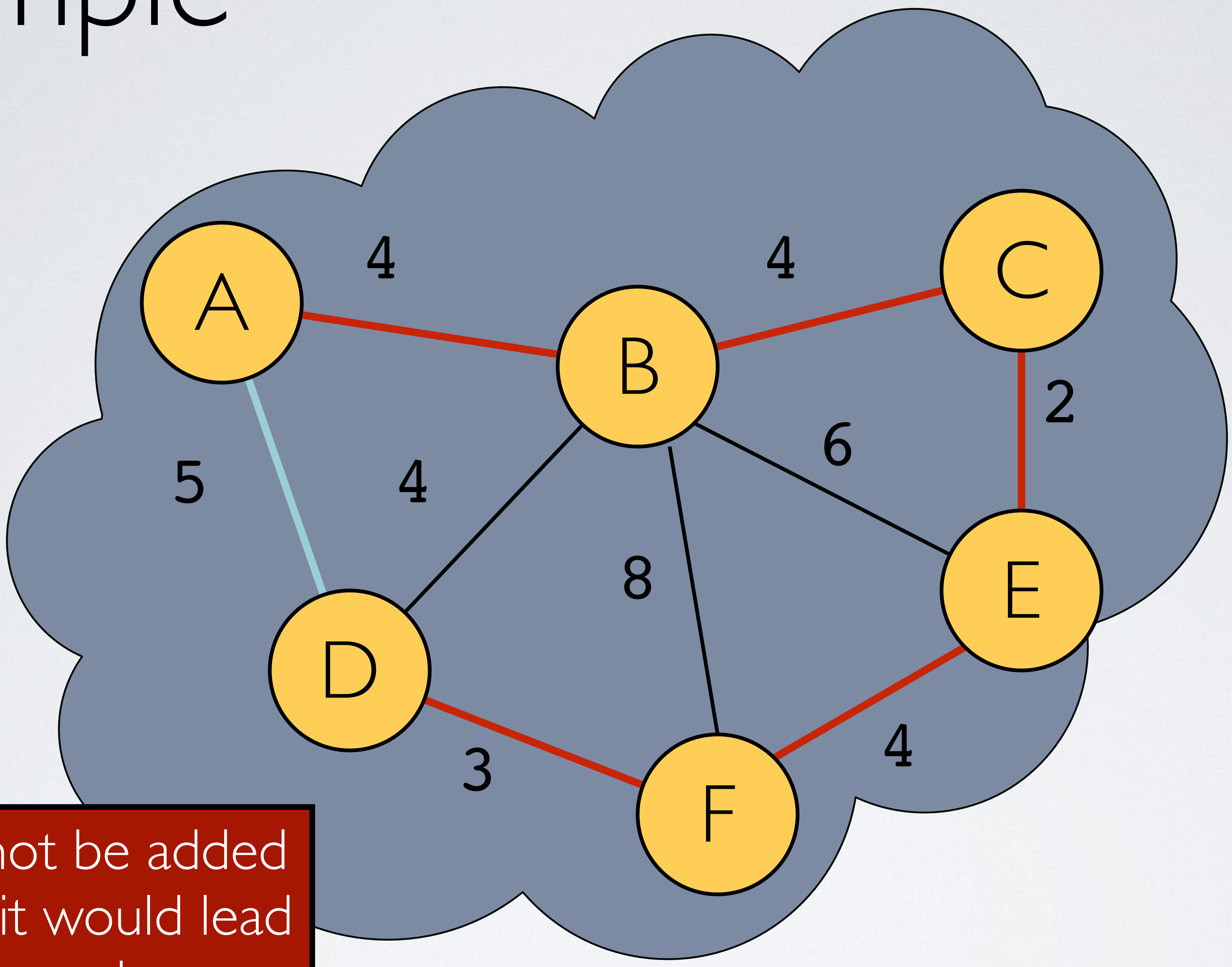
```
edges = [(A,B),(A,D),(B,E),(B,F)]
```

Example




```
edges = [(A,D),(B,E),(B,F)]
```

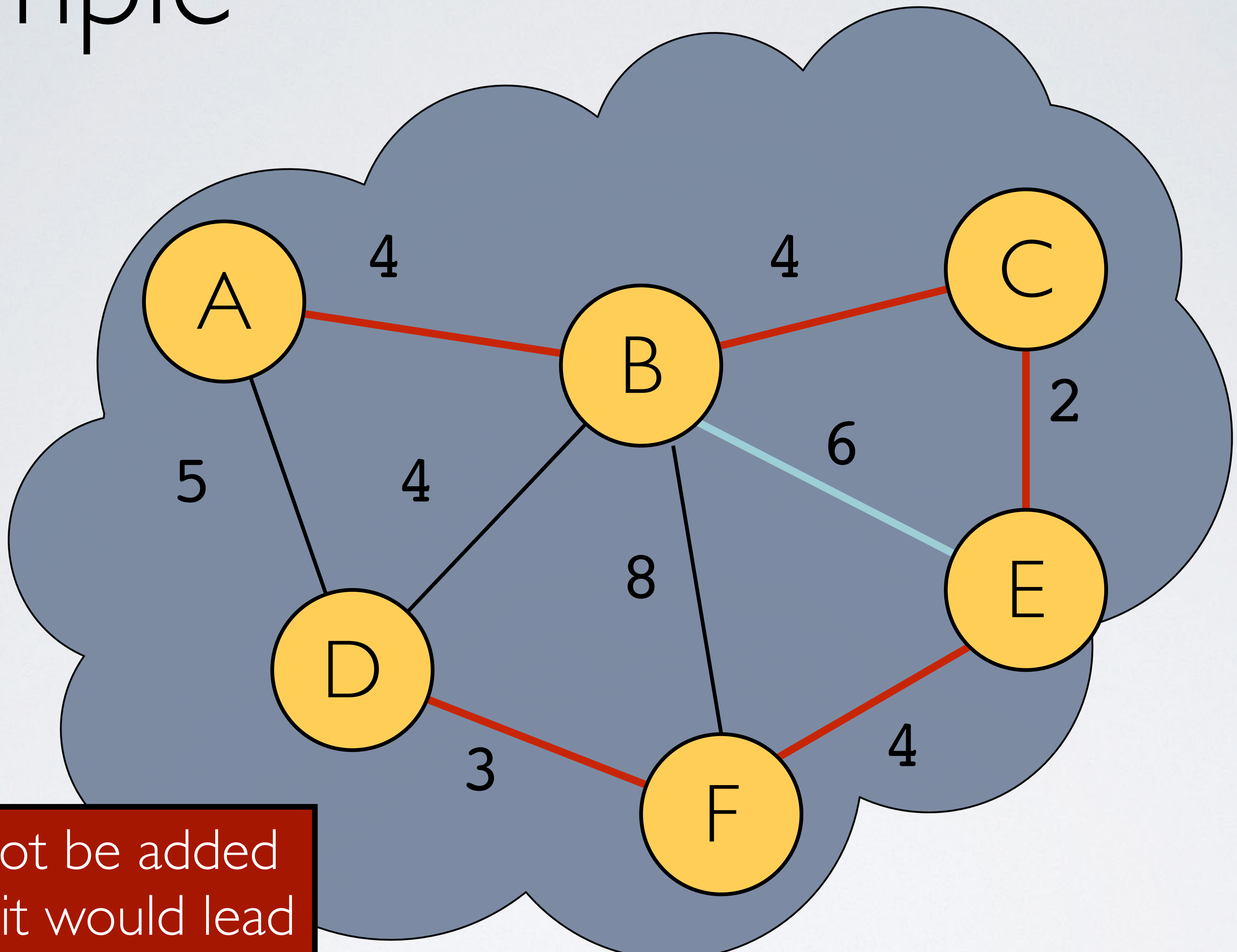
Example



AD cannot be added because it would lead to a cycle

```
edges = [(B,E),(B,F)]
```

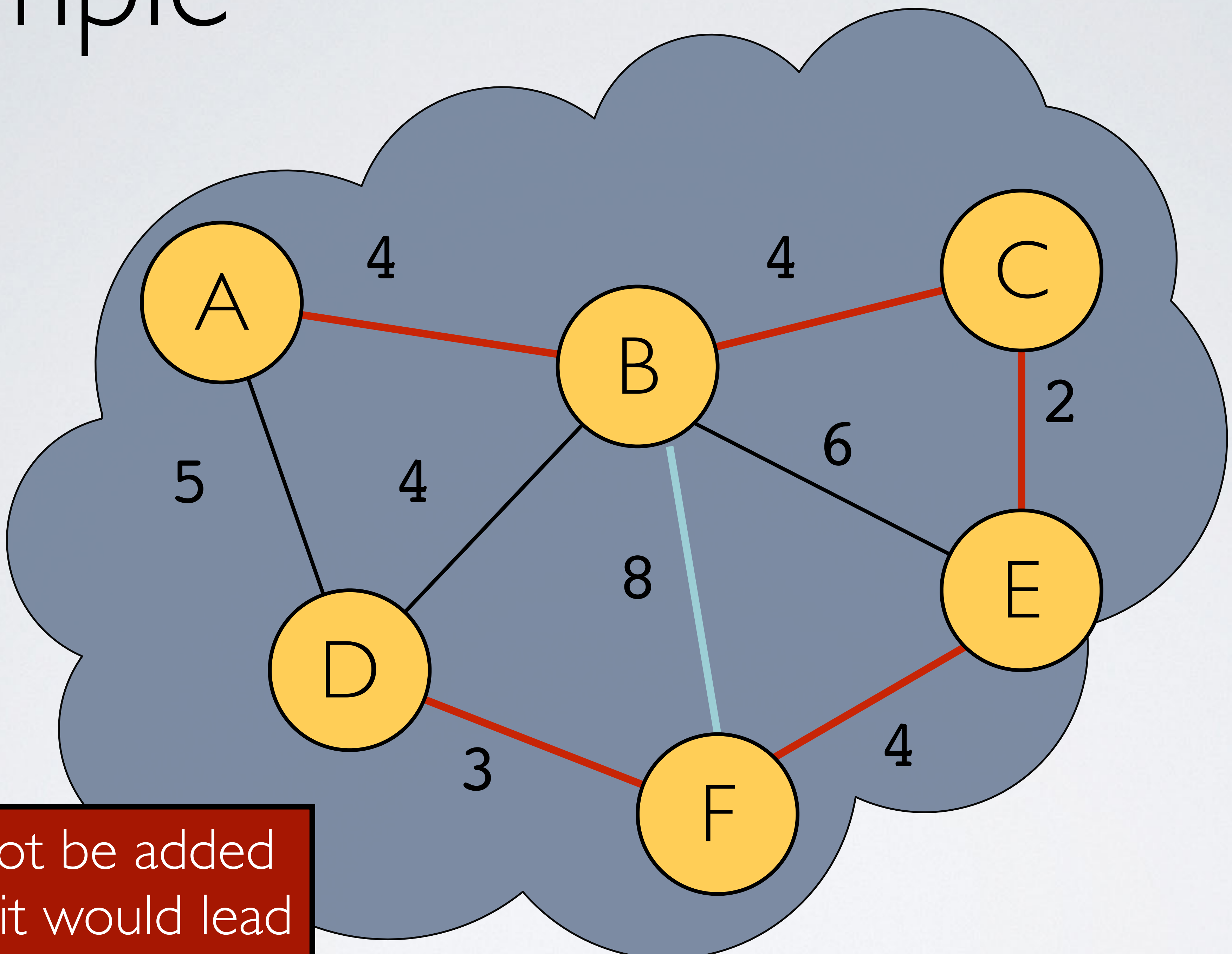
Example



BE cannot be added because it would lead to a cycle

`edges = [(B,F)]`

Example



BF cannot be added because it would lead to a cycle

`edges = [ ]`

Kruskal Pseudo-Code

```
function kruskal(G):
   // Input: undirected, weighted graph G
   // Output: list of edges in MST
   for vertices v in G:
      makeCloud(v) // put every vertex into it own set
   MST = []
   Sort all edges
   for all edges (u,v) in G sorted by weight:
      if u and v are not in same cloud:
         add (u,v) to MST
         merge clouds containing u and v
   return MST
```

Merging Clouds (Naive way)

- Assign each vertex a different number
  - that represents its initial cloud
- To merge clouds of **`u`** and **`v`**
  - Find all vertices in each cloud
  - Figure out which of the clouds is smaller
  - Redecorate all vertices in smaller cloud w/ bigger cloud's number

Merging Clouds (Naive way)

- Finding all vertices in `u` & `v`'s clouds is `O(|V|)`
  - because we have to iterate through each vertex…
  - …and check if its cloud number matches `u` or `v`'s cloud number
- Figuring out smaller cloud is `O(1)`
  - as long as we keep track of cloud size as we find vertices in them
- Changing cloud numbers of nodes in smaller cloud is `O(|V|)`
  - because smallest cloud could be as big as `|V|/2` vertices
- Total runtime to merge clouds
  - `O(|V| + 1 + |V|) = O(|V|)`

Kruskal Runtime w/ Naive Clouds

```
function kruskal(G):
   // Input: undirected, weighted graph G
   // Output: list of edges in MST
   for vertices v in G:                          O(|V|)
      makeCloud(v)
   MST = []
   Sort all edges                                O(|E|log|E|)
   for all edges (u,v) in G sorted by weight:    O(|E|)
      if u and v are not in same cloud:
         add (u,v) to MST
         merge clouds containing u and v         O(|V|)
   return MST
```

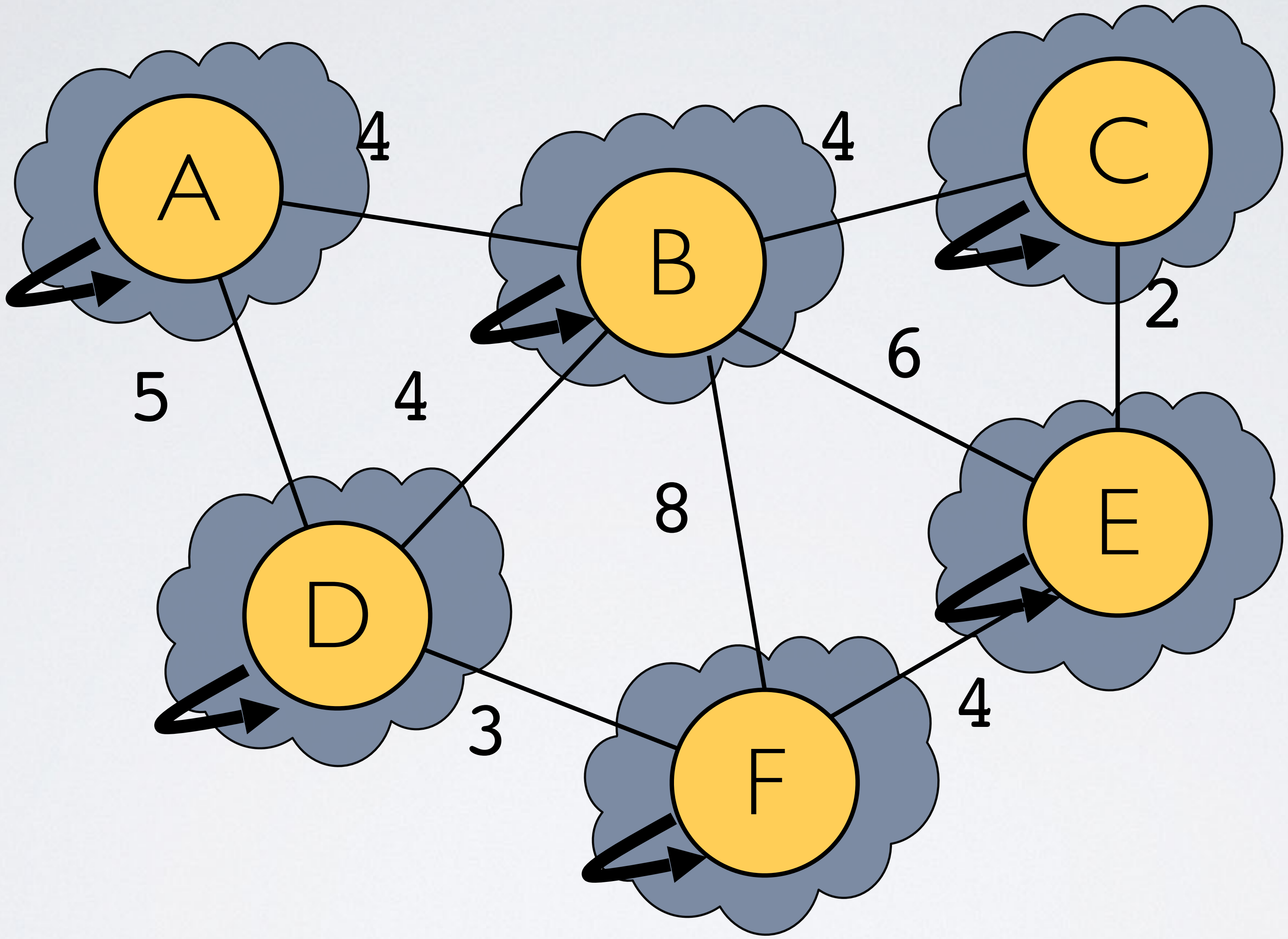
Kruskal Runtime

- `O(|V|)` for iterating through vertices
- `O(|E|log|E|)` for sorting edges
- `O(|E|×|V|)` for iterating through edges and merging clouds naively
- `O(|V|+|E|log|E|+|E|×|V|)`
  - `= O(|E|×|V|)`
- Can we do better?

Union-Find

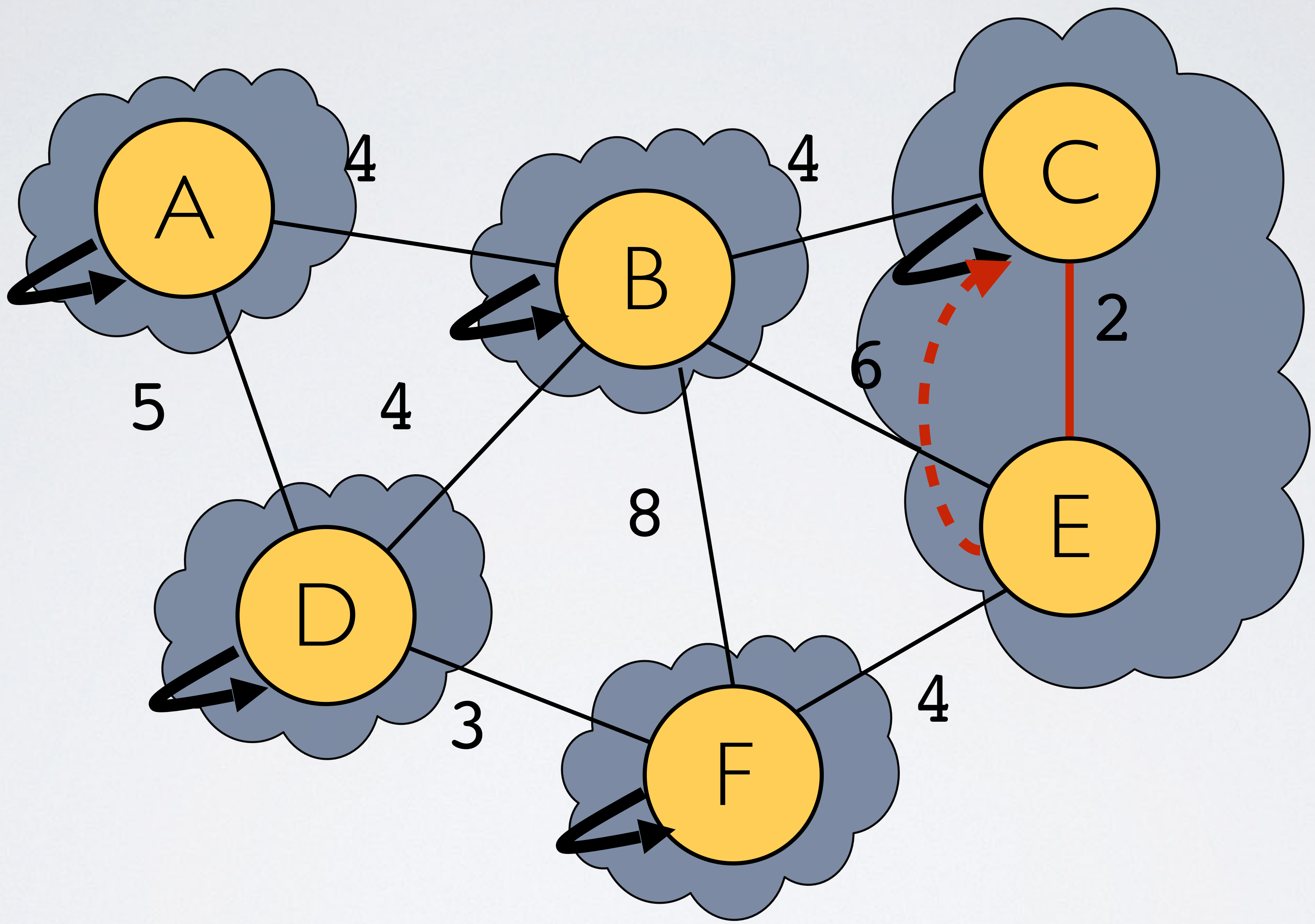
- Let's rethink notion of clouds
  - instead of labeling vertices w/ cloud numbers
  - think of clouds as small trees
- Every vertex in these trees has
  - a parent pointer that leads up to root of the tree
  - a rank that measures how deep the tree is

Example



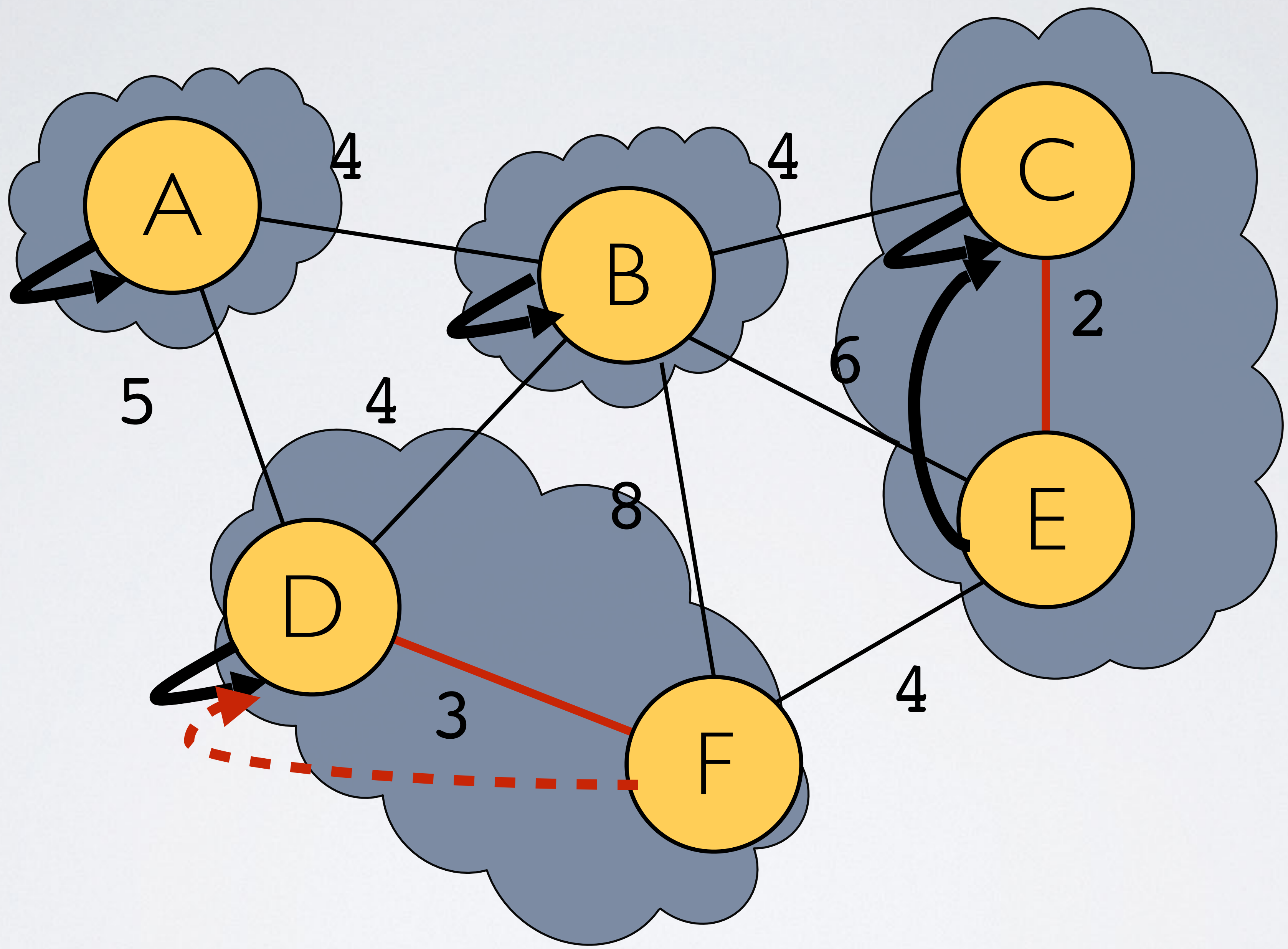

```
edges = [(C,E),(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



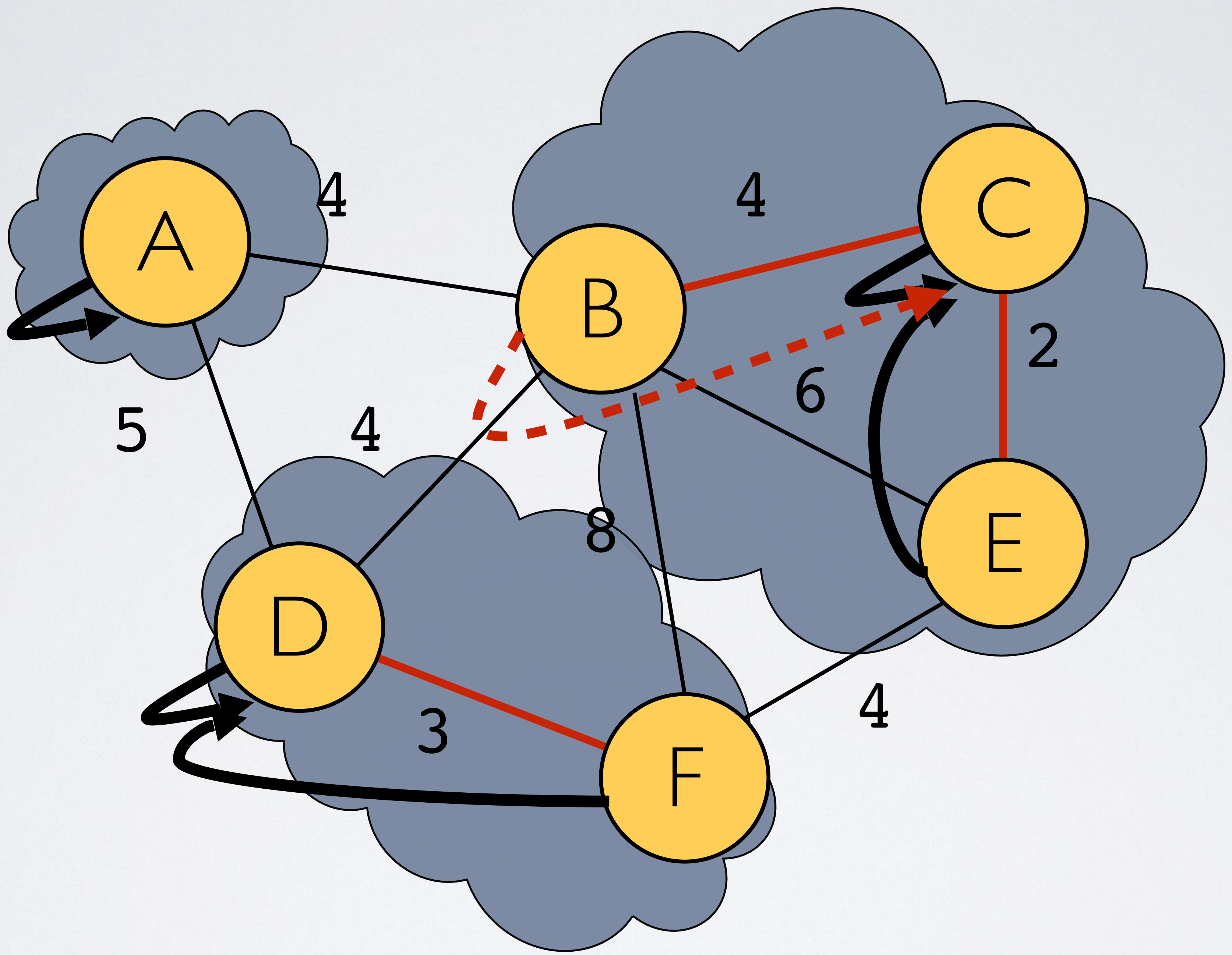

```
edges = [(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



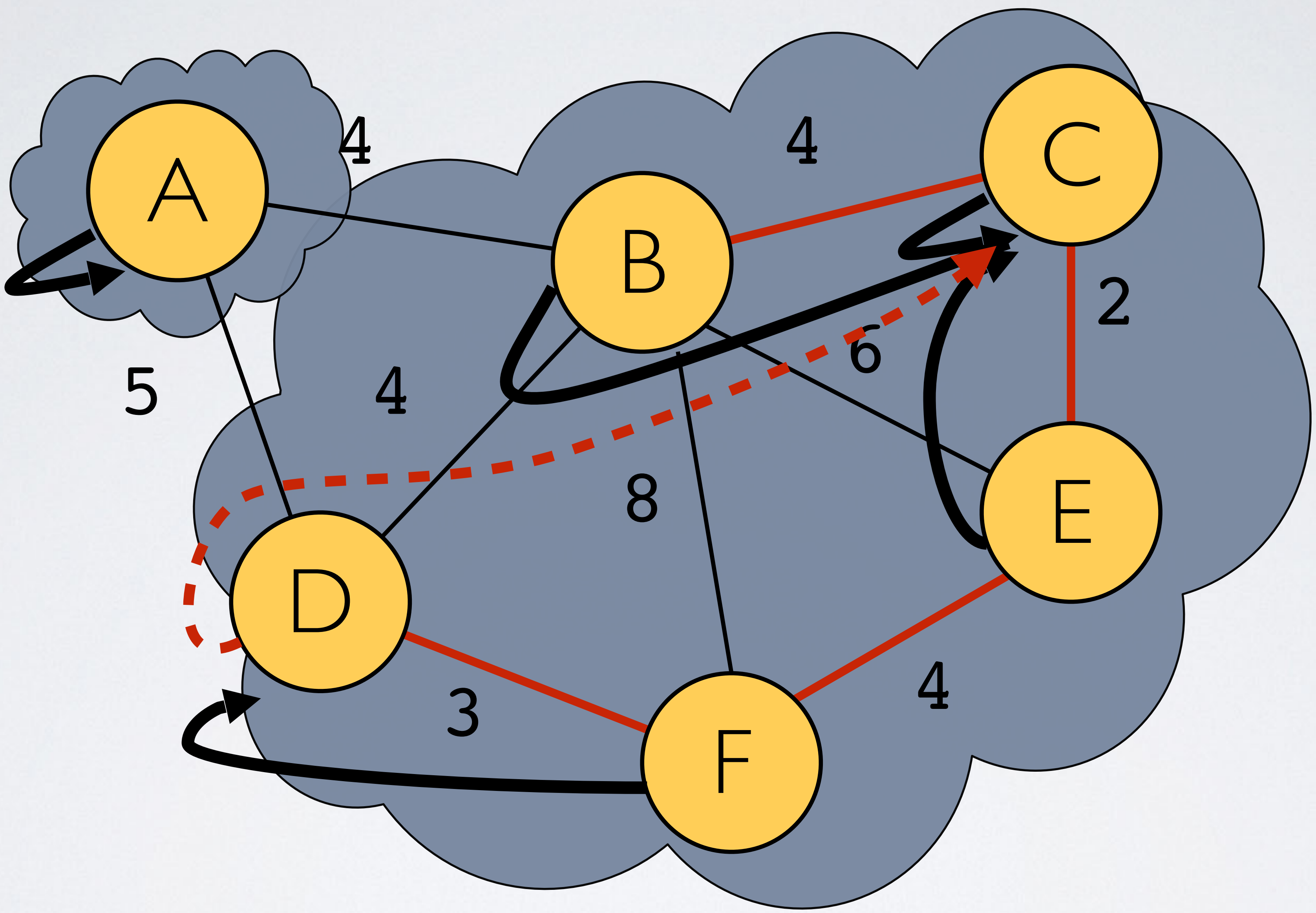

```
edges = [(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



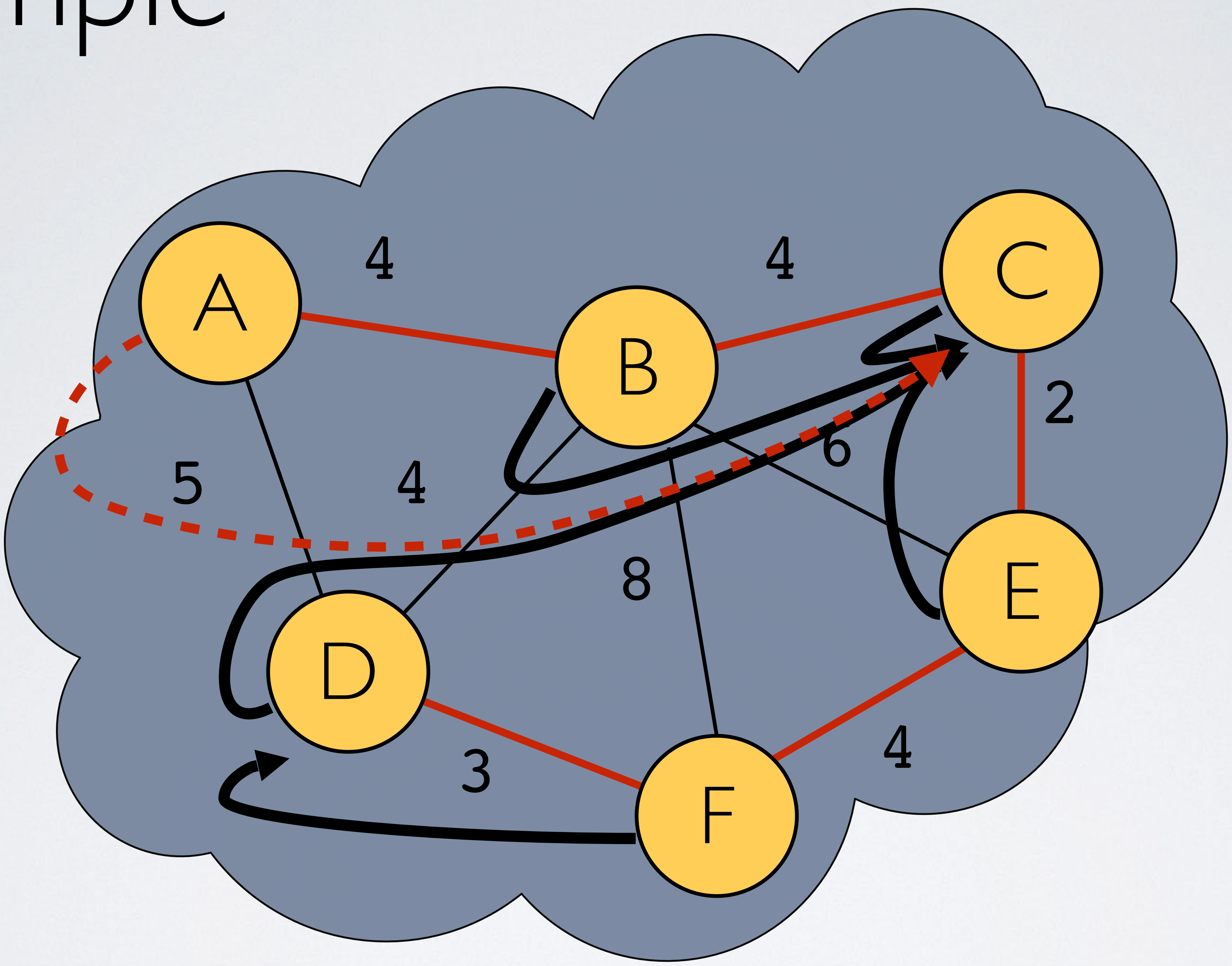
`edges = [(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]`

Example



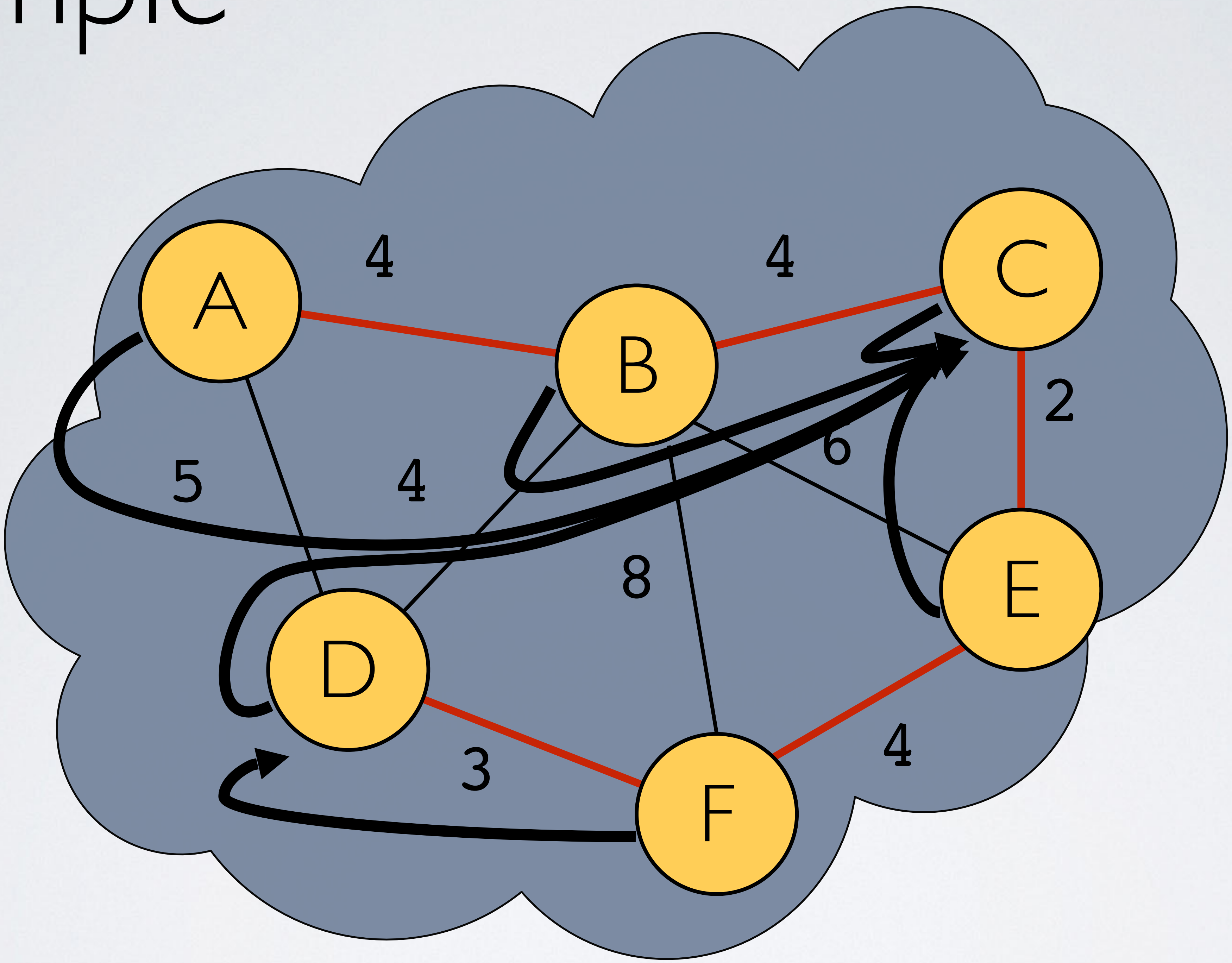

```
edges = [(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example




```
edges = [(A,D),(B,E),(B,F)]
```

Example

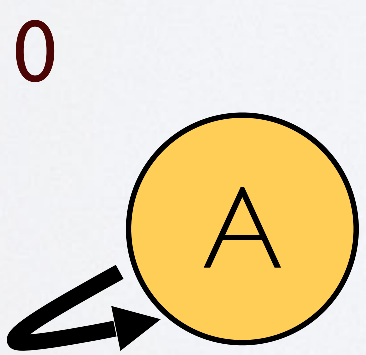



```
edges = [(A,D),(B,E),(B,F)]
```

Implementing Union-Find

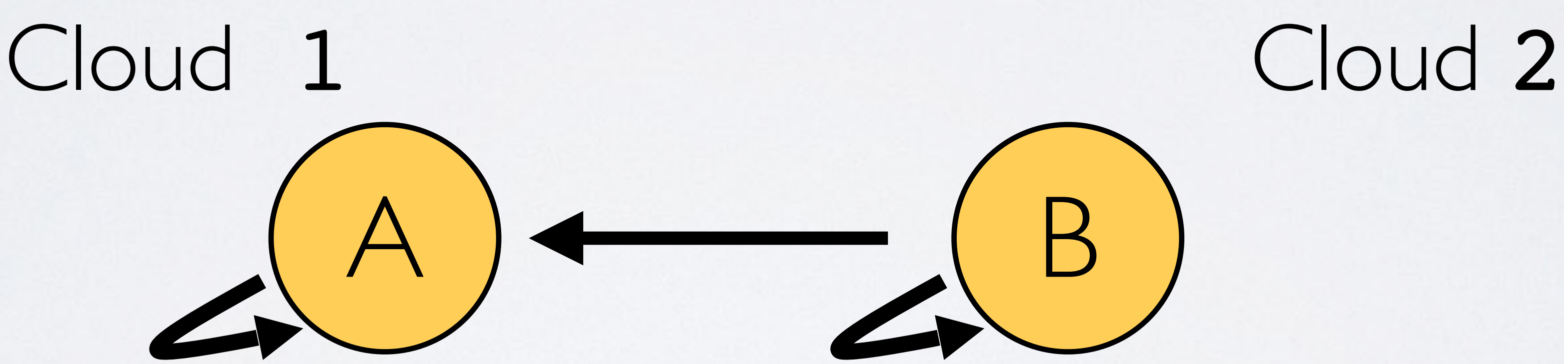
- At start of Kruskal
  - every node is put into own cloud

```
// Decorates every vertex with its parent ptr & rank
function makeCloud(x):
    x.parent = x
    x.rank = 0
```



Implementing Union-Find

- Suppose `A` is in cloud `1` and `B` is in cloud `2`
- Instead of relabeling `B` as cloud `1` make `B` point to `A`
  - Think of this as the union of two clouds



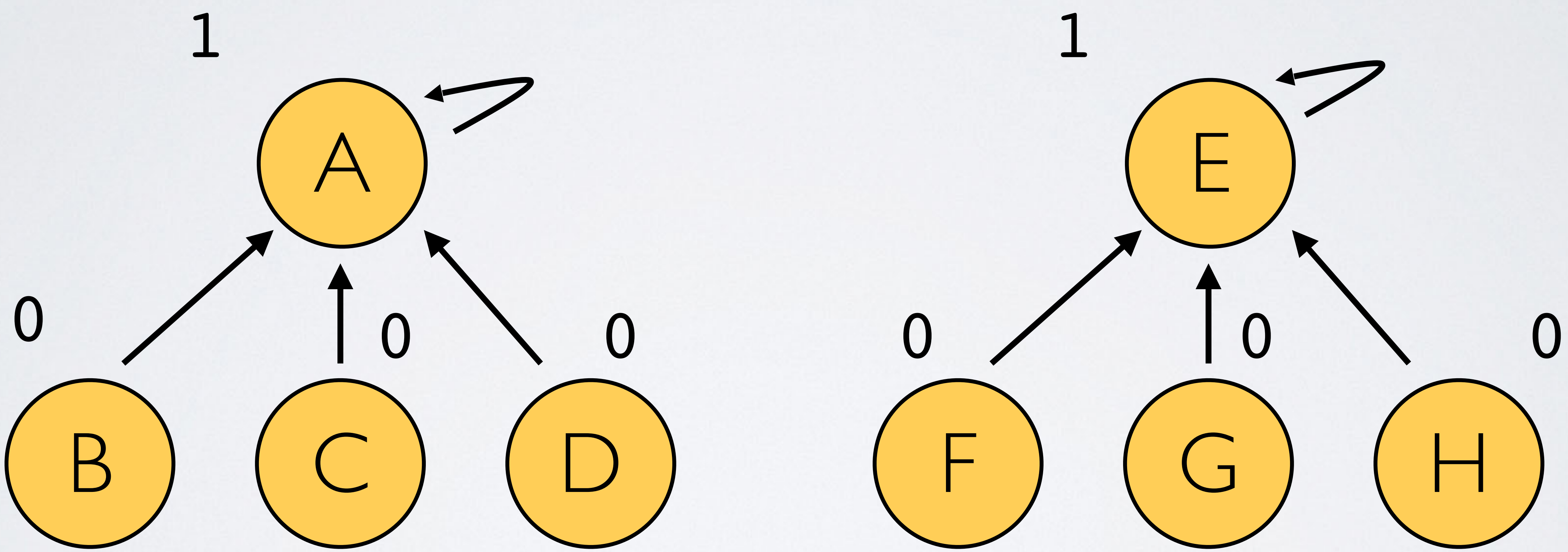
  - Given two clouds which one should point to the other?

Implementing Union-Find

- We use the rank to decide
  - make lower-ranked root point to higher-ranked root
  - then update rank
- How do we update ranks?
  - For clouds of size `1` root always has rank `0`
  - For clouds of size larger than `1` we increment rank only when merging clouds of same rank

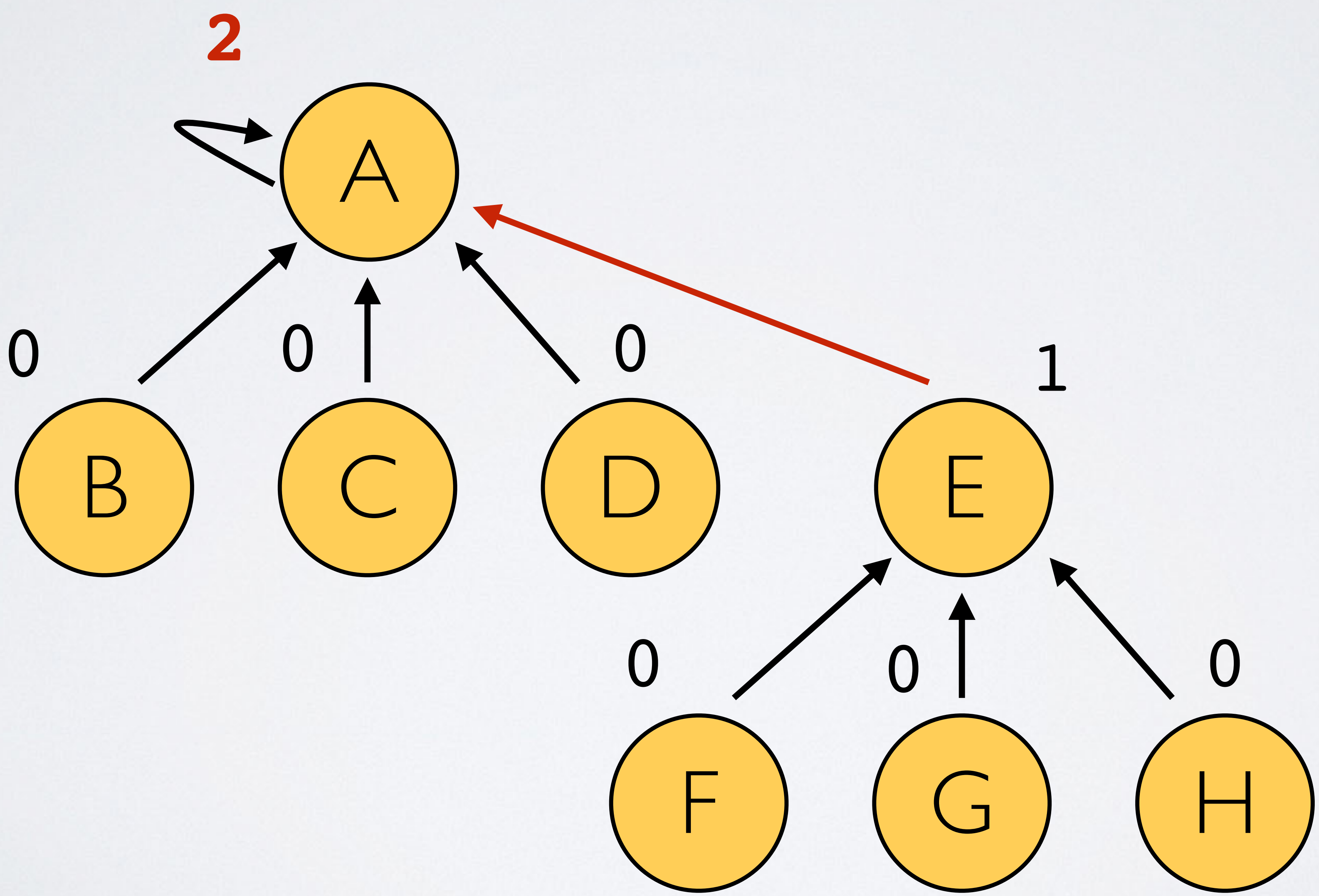
Implementing Union-Find

- Merging trees with same rank



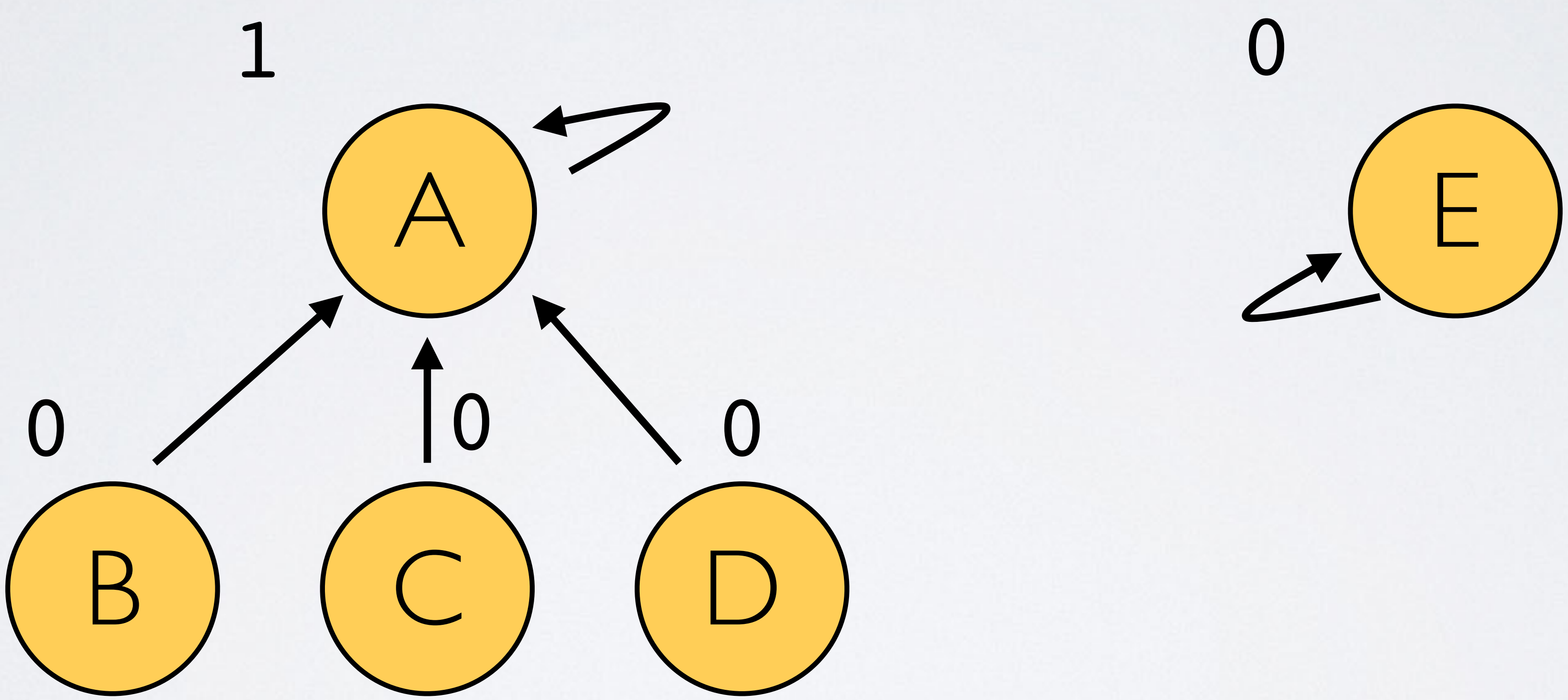
Implementing Union-Find

- Merging trees with same rank



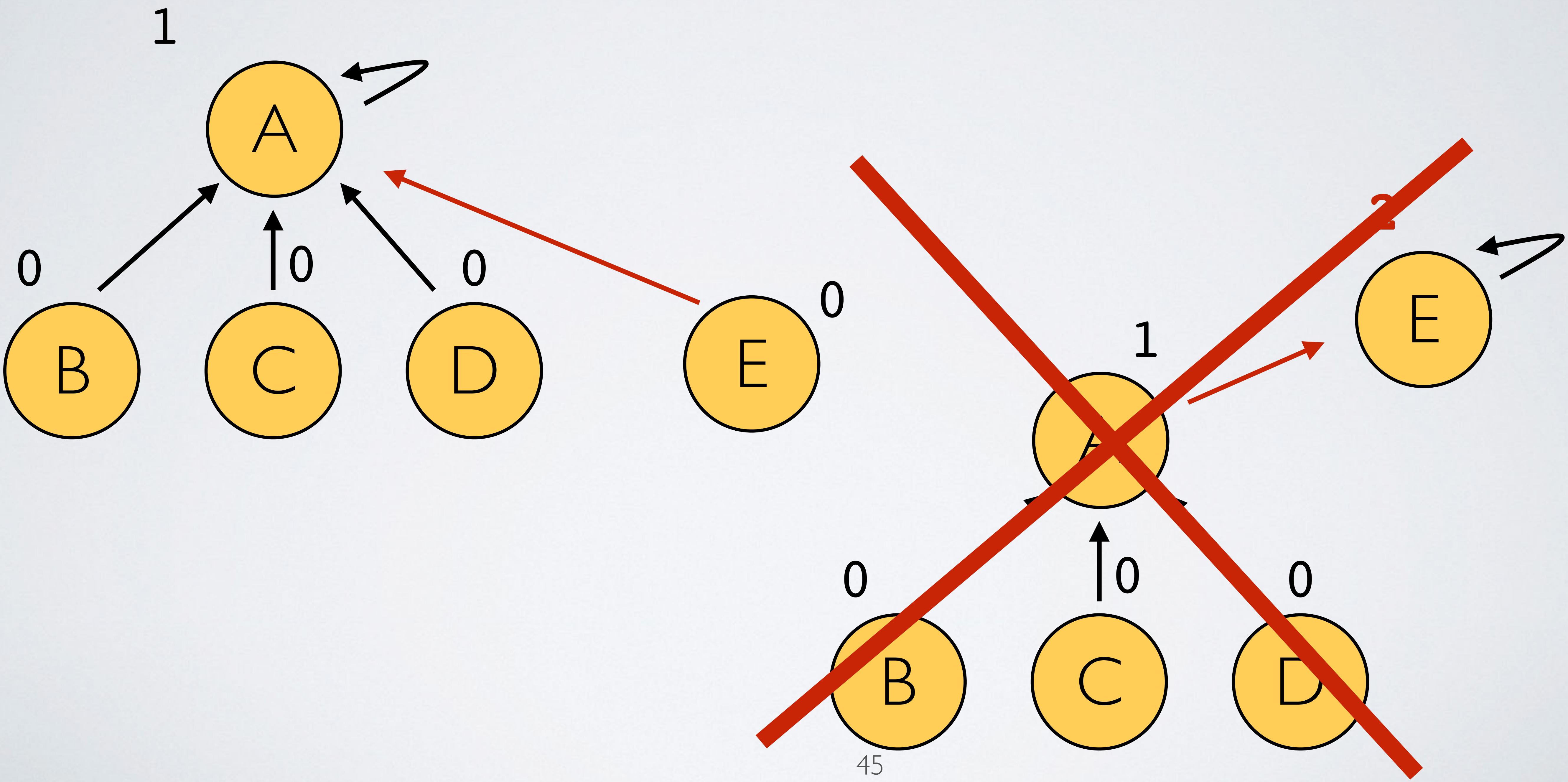
Implementing Union-Find

- Merging trees with different ranks



Implementing Union-Find

- Merging trees with different ranks



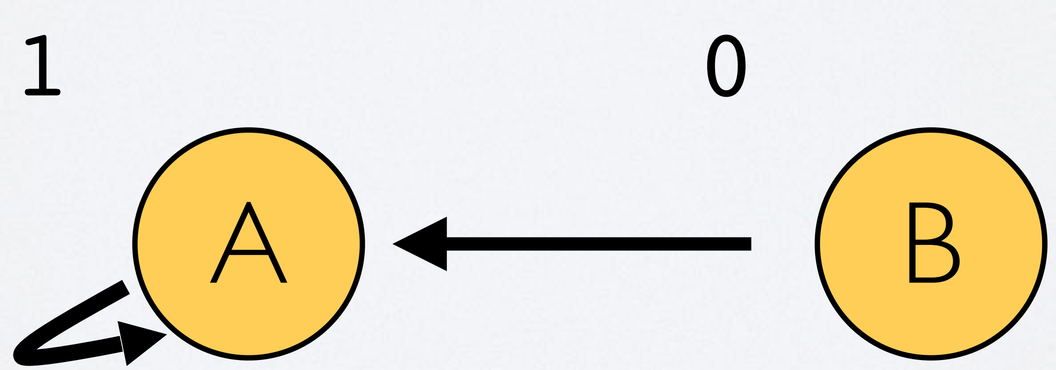
Implementing Union-Find

```
// Merges two clouds, given the root of each cloud
function union(root1, root2):
    if root1.rank > root2.rank:
        root2.parent = root1
    elif root1.rank < root2.rank:
        root1.parent = root2
    else:
        root2.parent = root1
        root1.rank++
```

Implementing Union-Find

- To find the cloud of **B**
  - follow **B**'s parent pointer all the way up to root

```
// Finds the cloud of a given vertex
function find_root(x):
    while x.parent != x:
        x = x.parent
    return x
```

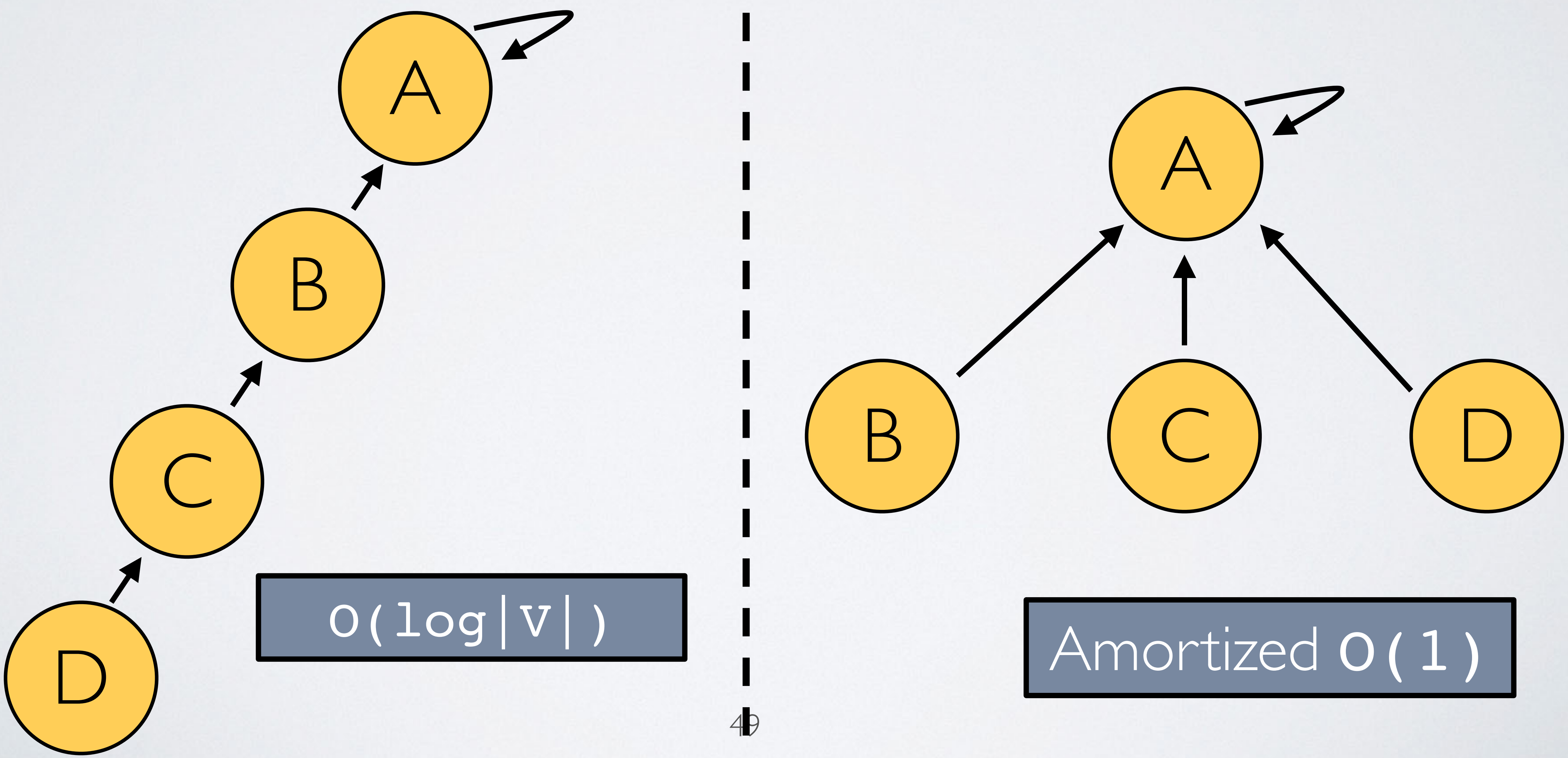


Path Compression

- This approach to implementing **`find`** runs in
  - **`O(log|V|)`**
  - not obvious to see why and proof beyond CS16
- We can bring this down to amortized **`O(1)*`**
  - with path compression…
  - …a way of flattening the structure of the tree…
  - …whenever **`find()`** is used on it
  - *not actually O(1) but *very* close—analysis goes beyond CS16 material

Path Compression

- Instead of traversing up tree every time **`D`**'s cloud is asked for
  - We only search for **`D`**'s root once
  - As we follow chain of parents to **`A`** we set parents of **`D`** & **`C`** to **`A`**



Path Compression Pseudo-code

```
function find_root(x):
    if x.parent != x:
        x.parent = find_root(x.parent)
    return x.parent
```

Runtime of Kruskal w/ Path Compression

```
function kruskal(G):
    // Input: undirected, weighted graph G
    // Output: list of edges in MST
    for vertices v in G:                              O(|V|)
        makeCloud(v)
    MST = []
    Sort all edges                                    O(|E|log|E|)
    for all edges (u,v) in G sorted by weight:        O(|E|)
        if u and v are not in same cloud:             O(1) amortized
            add (u,v) to MST
            merge clouds containing u and v           O(1) amortized
    return MST
```

Kruskal Runtime

- `O(|V|)` for iterating through vertices
- `O(|E|log|E|)` for sorting edges
- `O(|E|×1)` for iterating through edges and merging clouds with path compression
- `O(|V|+|E|log|E|+|E|×1)`
  - `= O(|V|+|E|log|E|)`
- `O(|V|+|E|log|E|)` better than `O(|V|×|E|)`

Readings

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