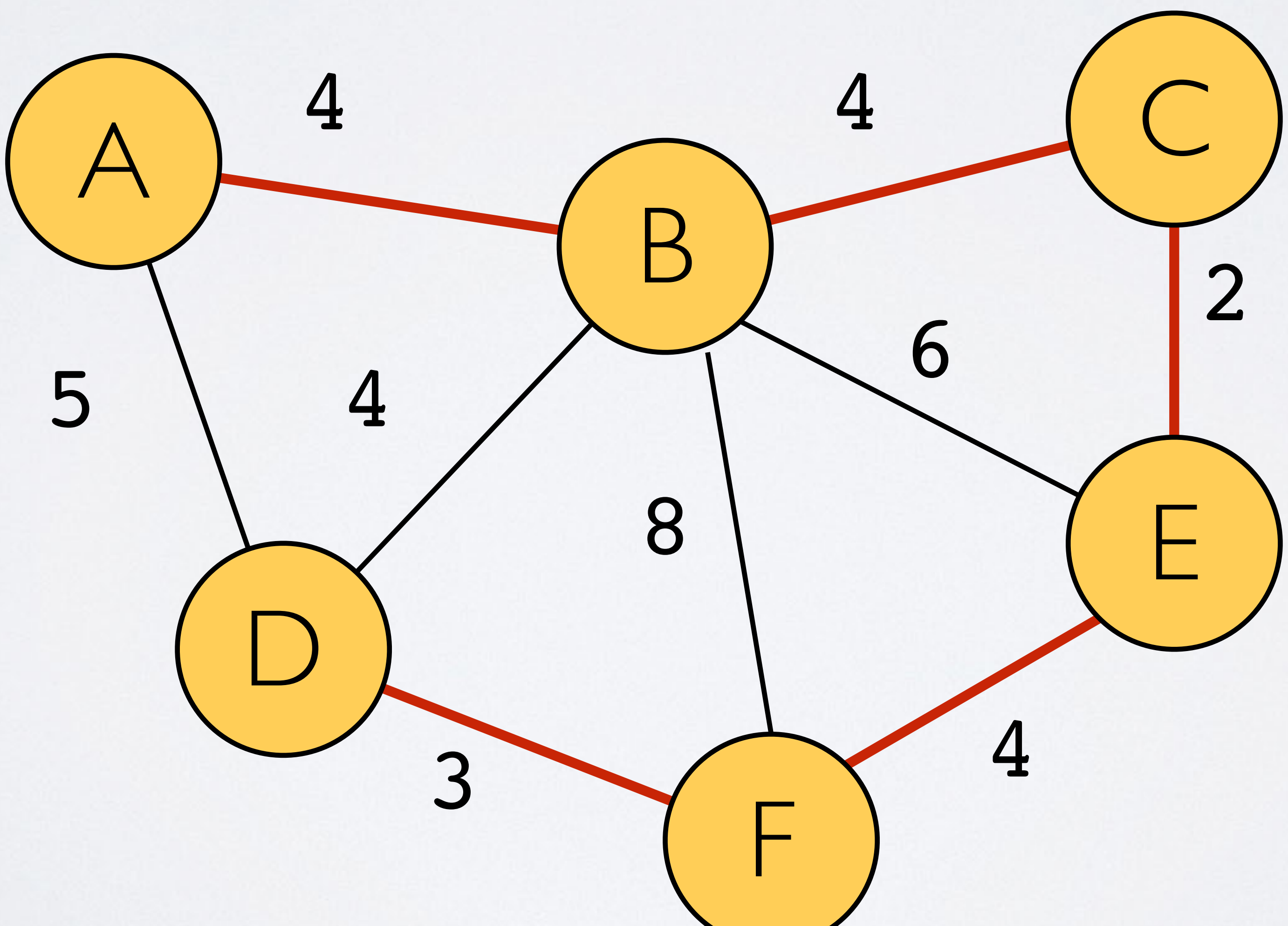
Minimum Spanning Trees: Implementing Kruskal

CS16: Introduction to Data Structures & Algorithms
Spring 2020

Minimum Spanning Trees

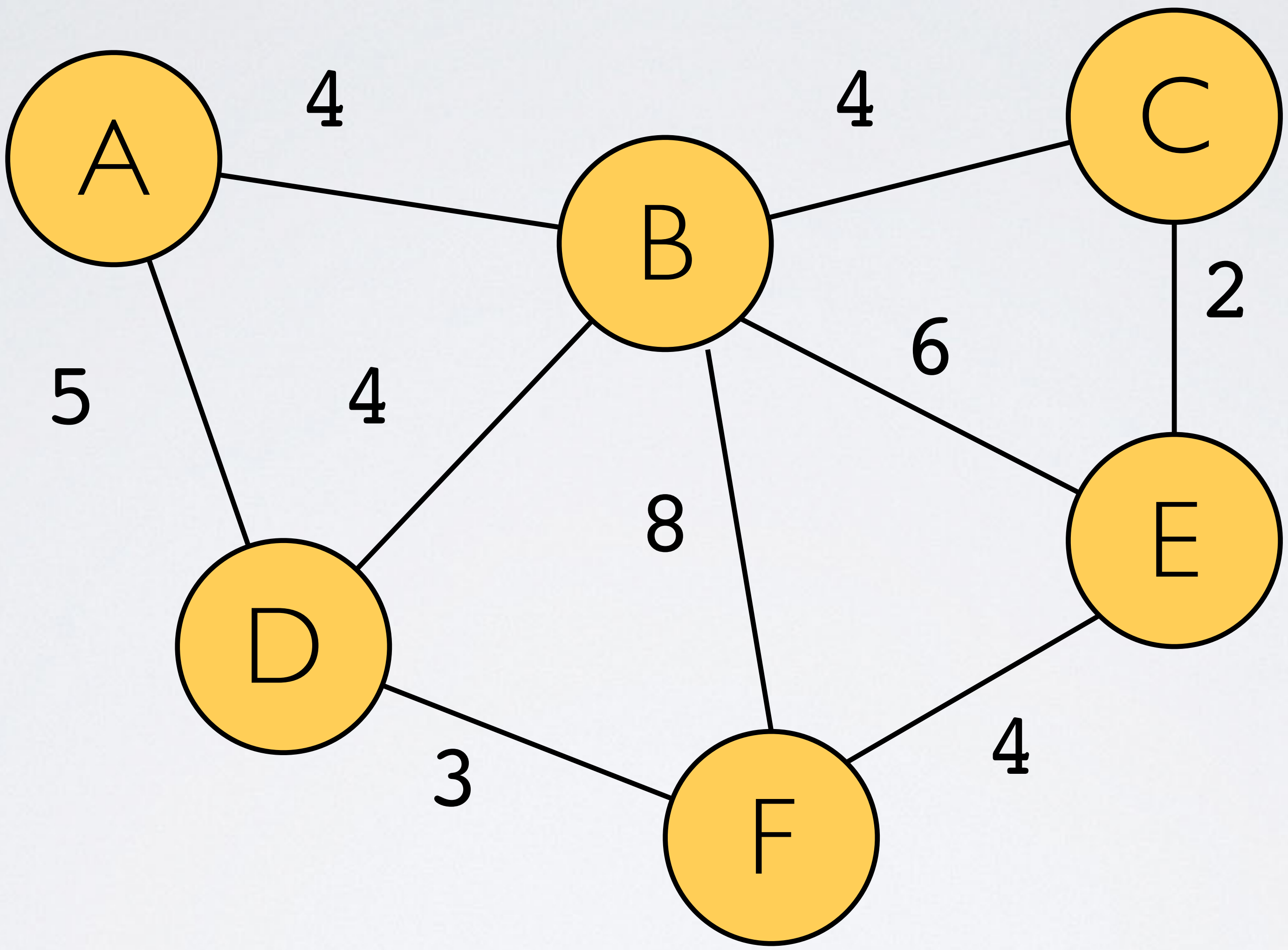
- A **minimum spanning tree** (MST) is
  - spanning tree with minimum total edge weight



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

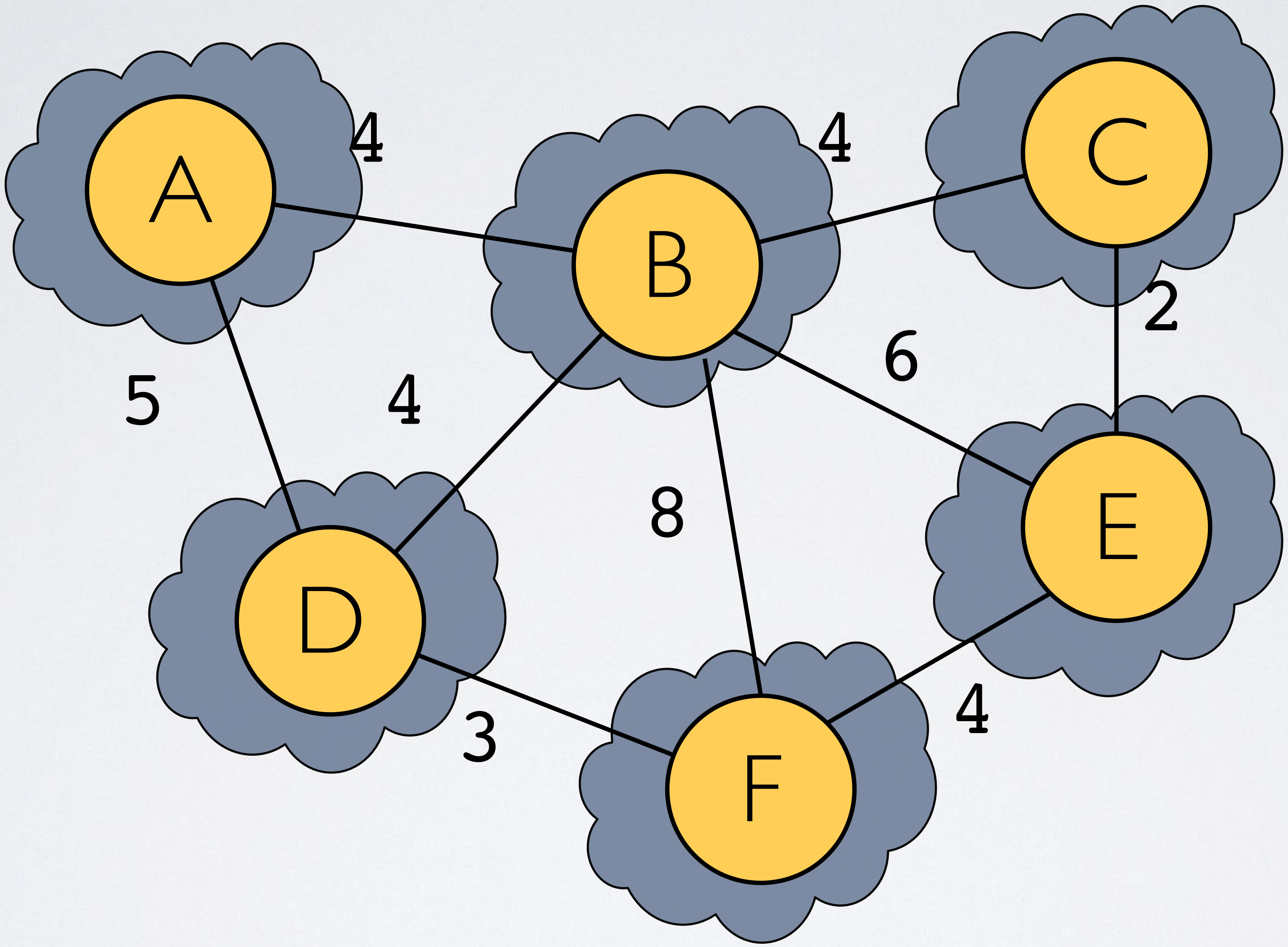



```
edges = [(C,E),(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Kruskal

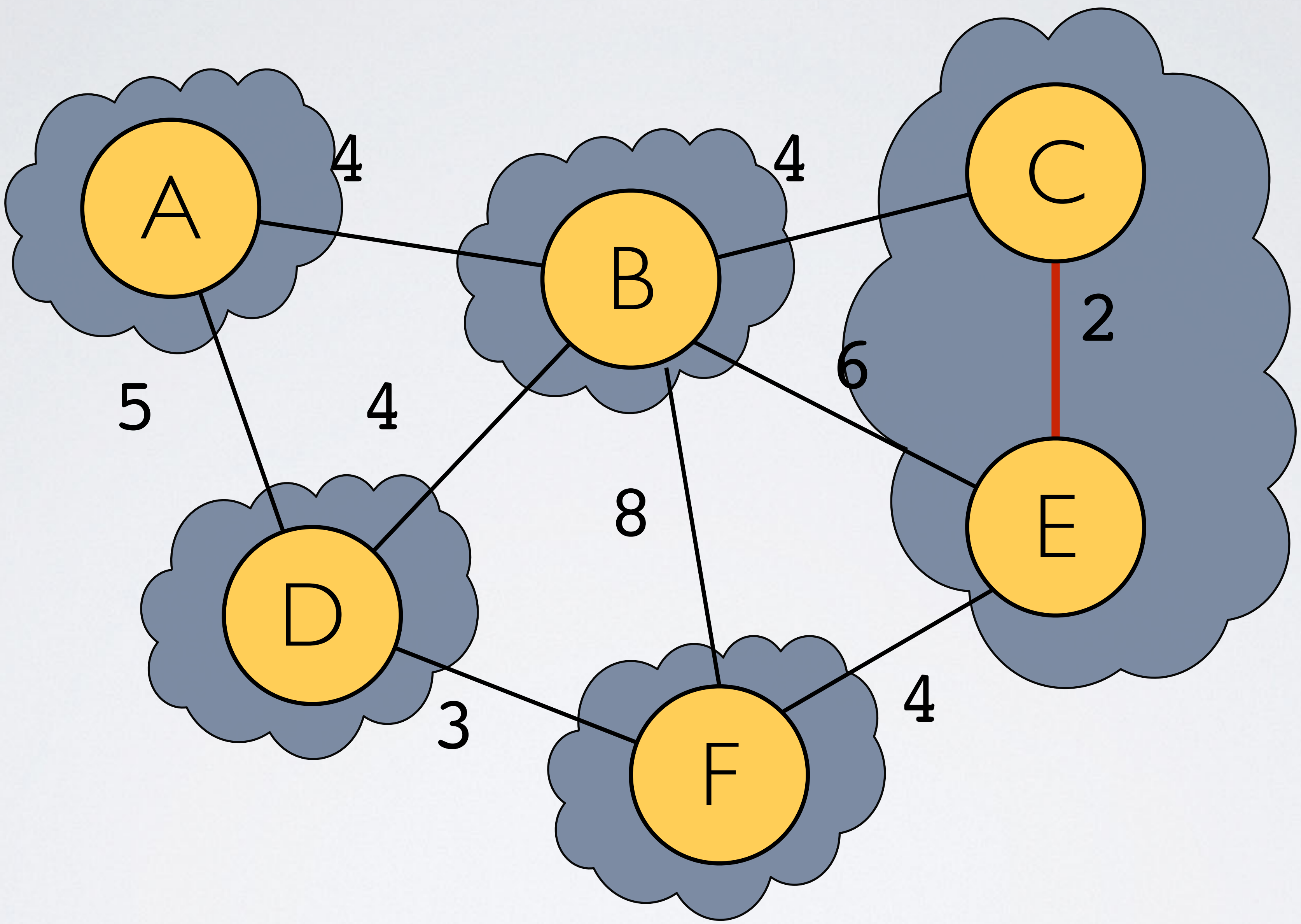
- How can we tell if adding edge will create cycle?
- Start by giving each vertex its own “cloud”
- If both ends of lowest-cost edge are in same cloud
  - we know that adding the edge will create a cycle!
- When edge is added to MST
  - merge clouds of the endpoints

Example



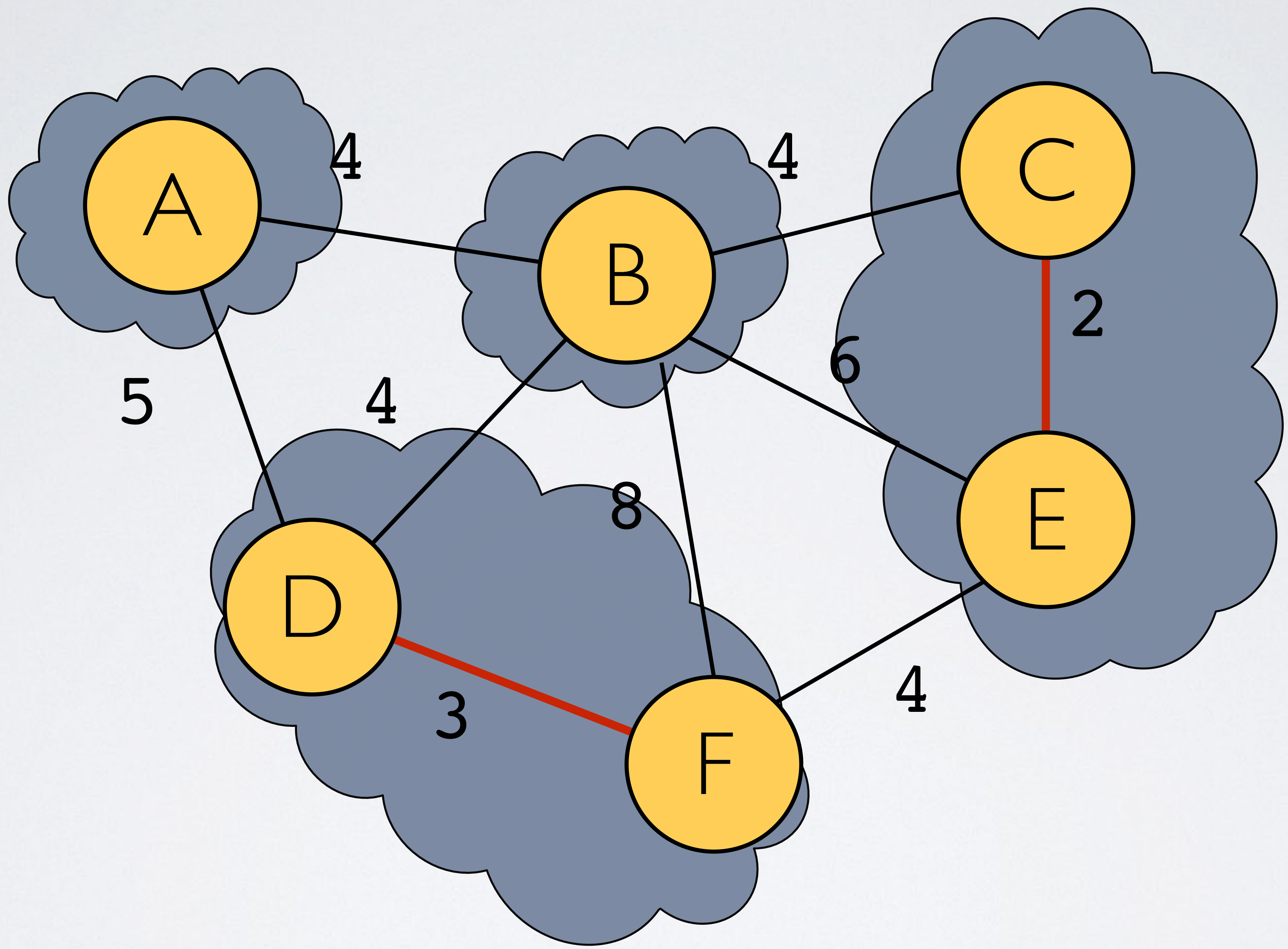

```
edges = [(C,E),(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



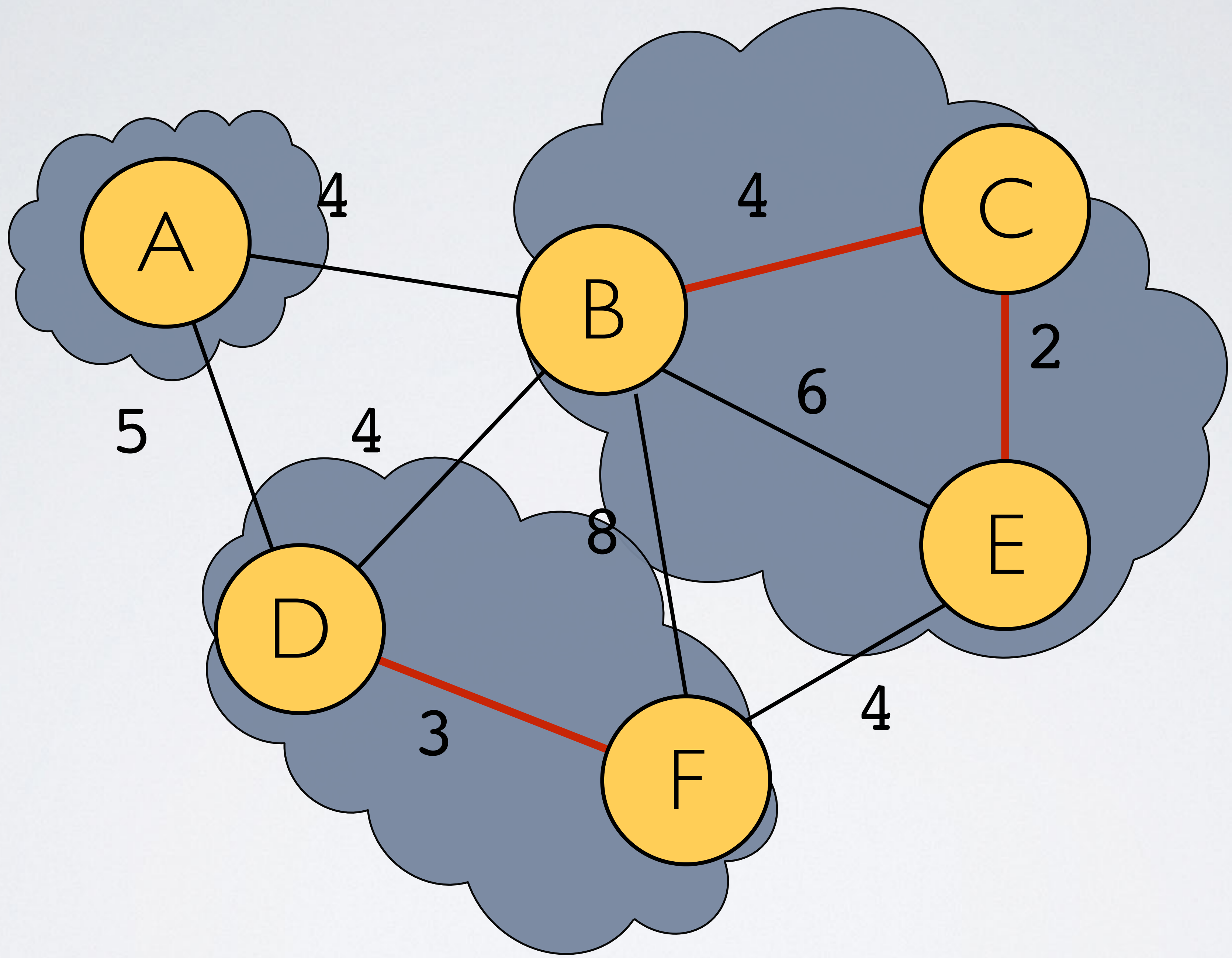

```
edges = [(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



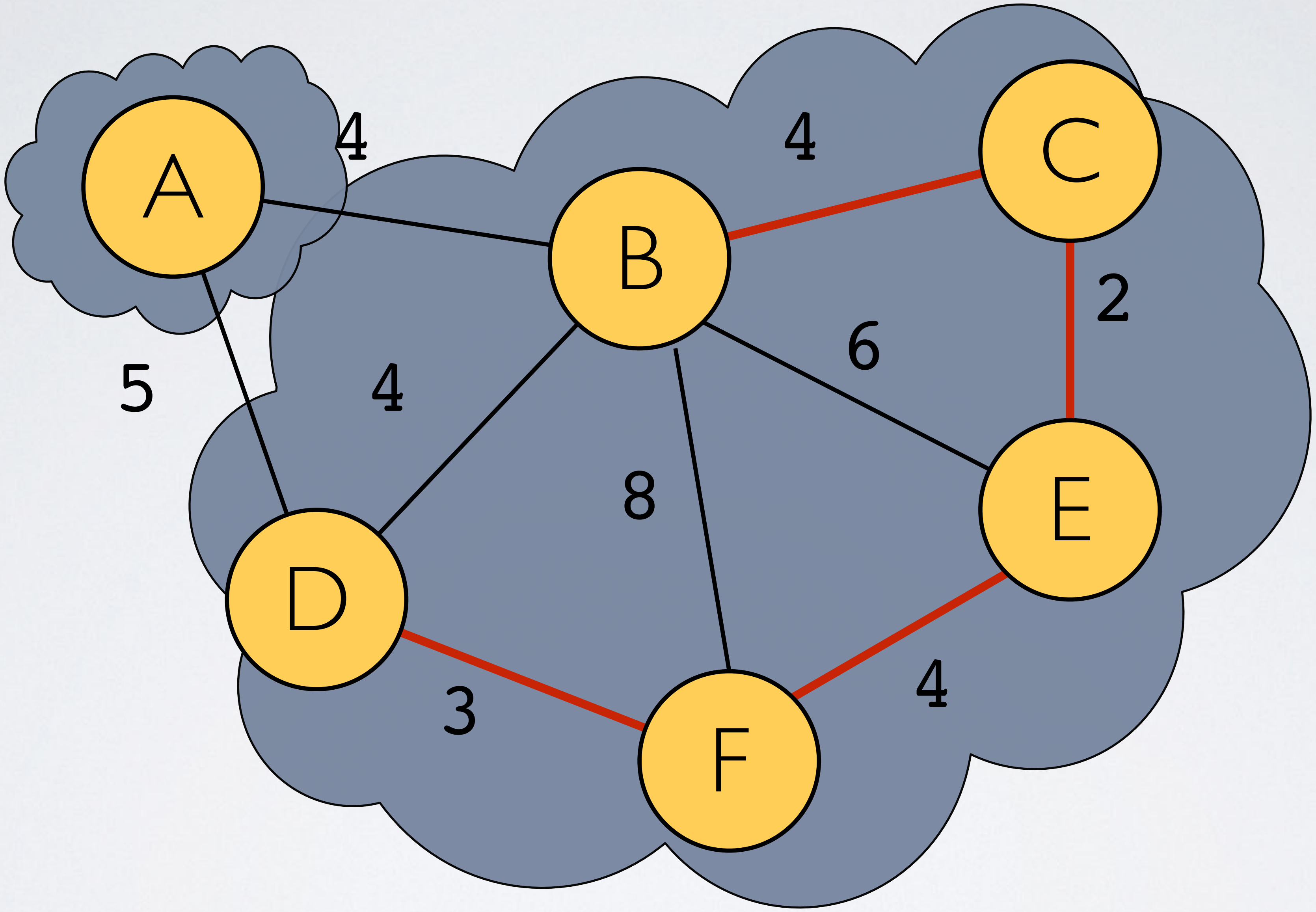

```
edges = [(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



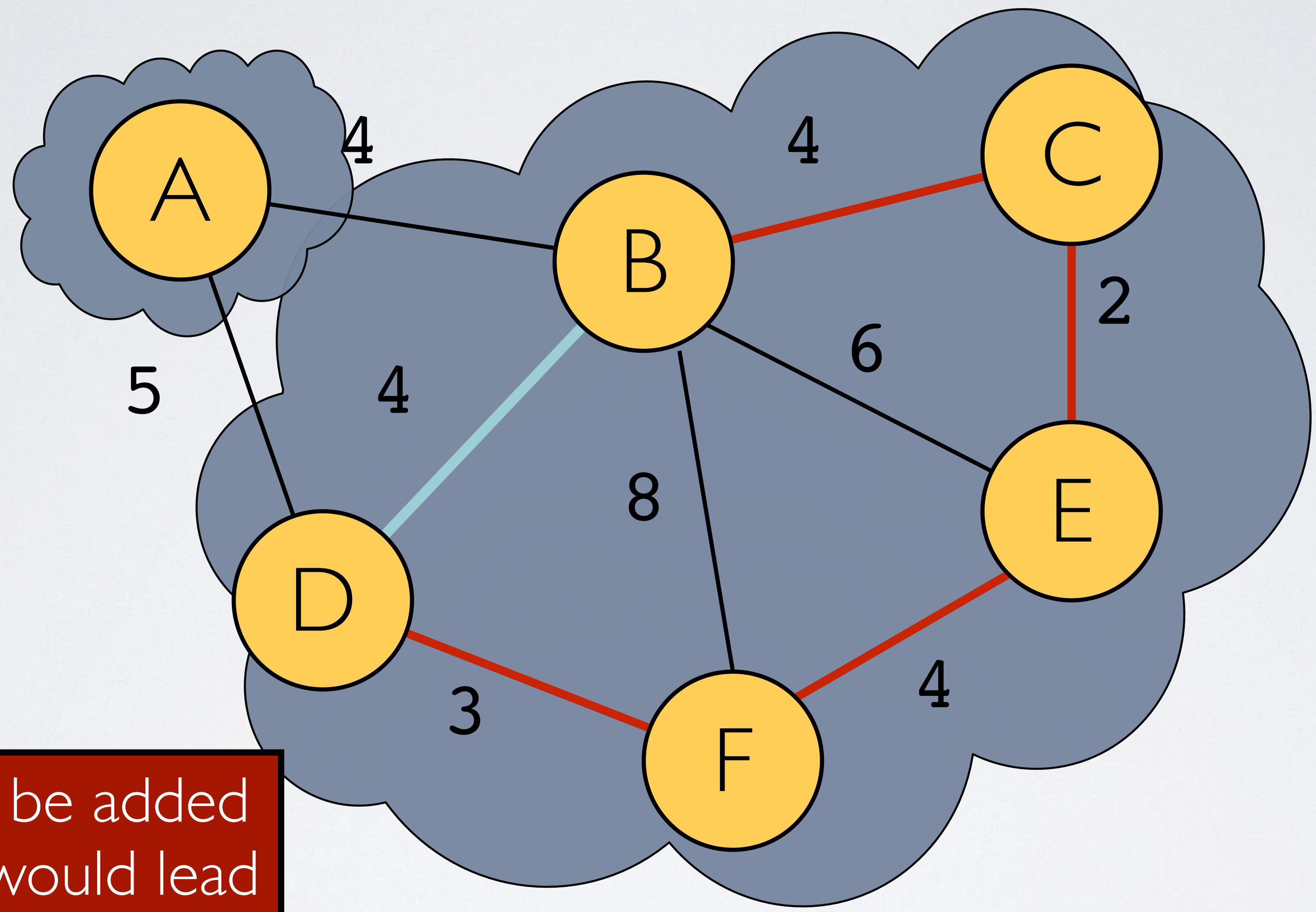

```
edges = [(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



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

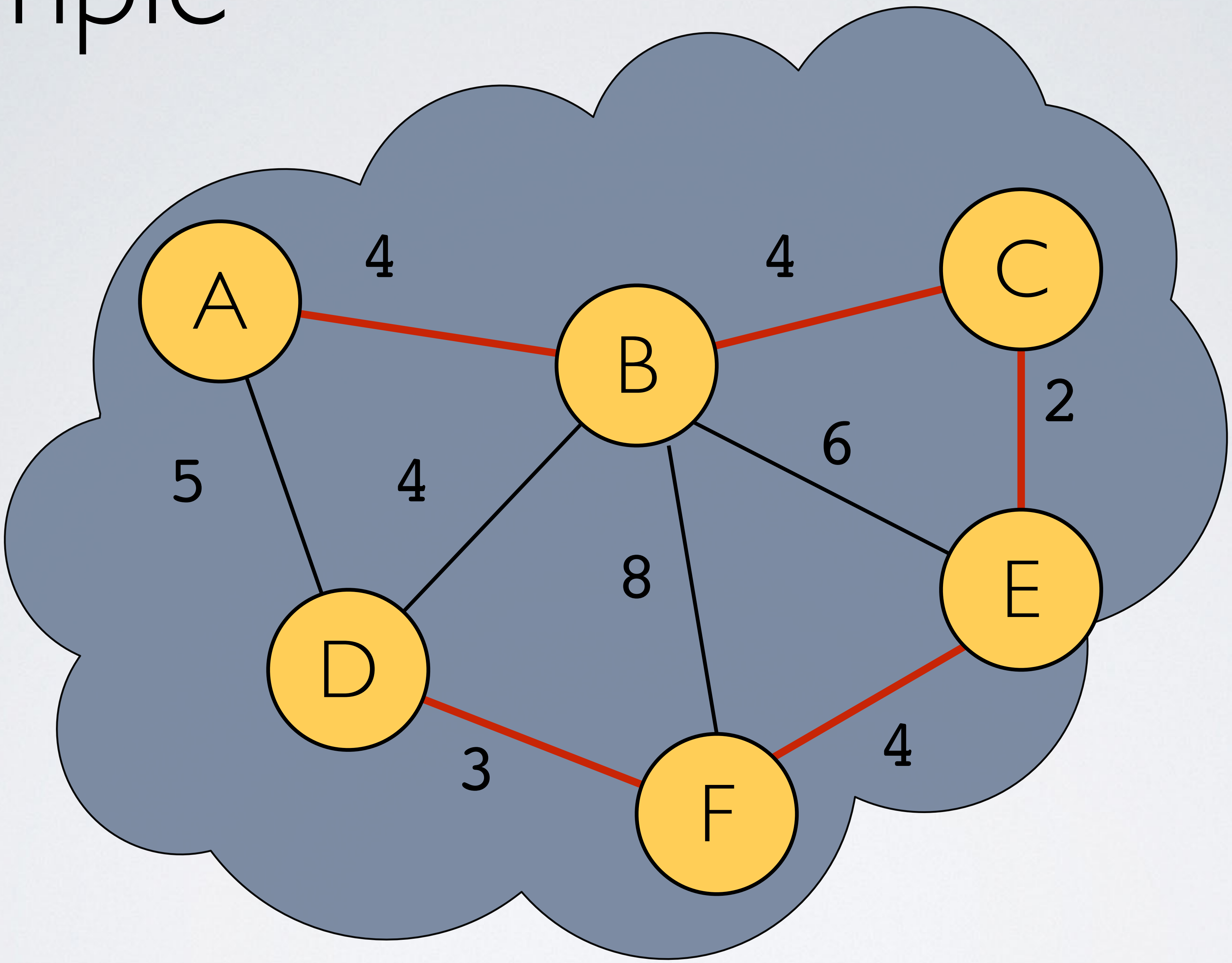
Example



BD cannot be added because it would lead to a cycle

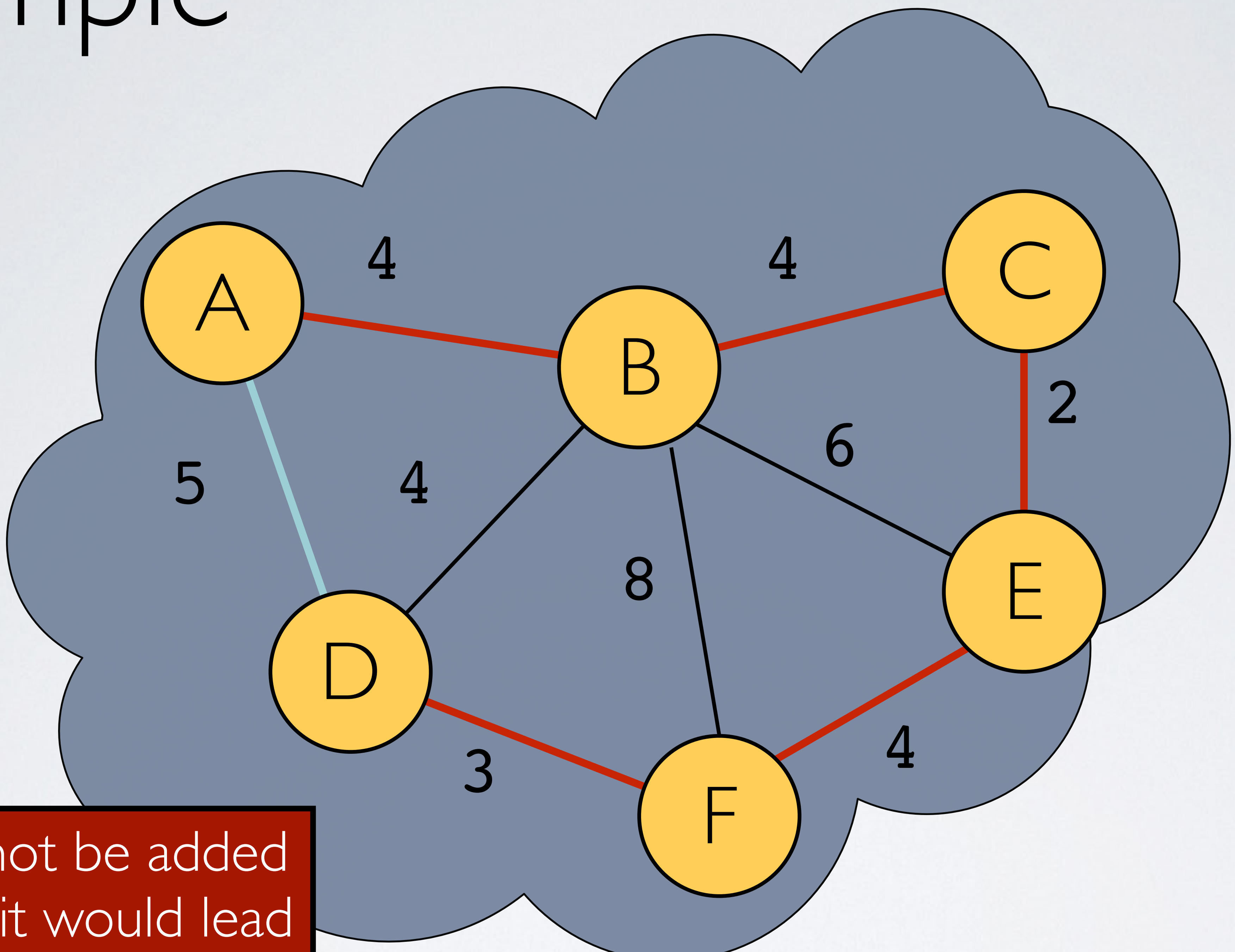
```
edges = [(A,B),(A,D),(B,E),(B,F)]
```

Example



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

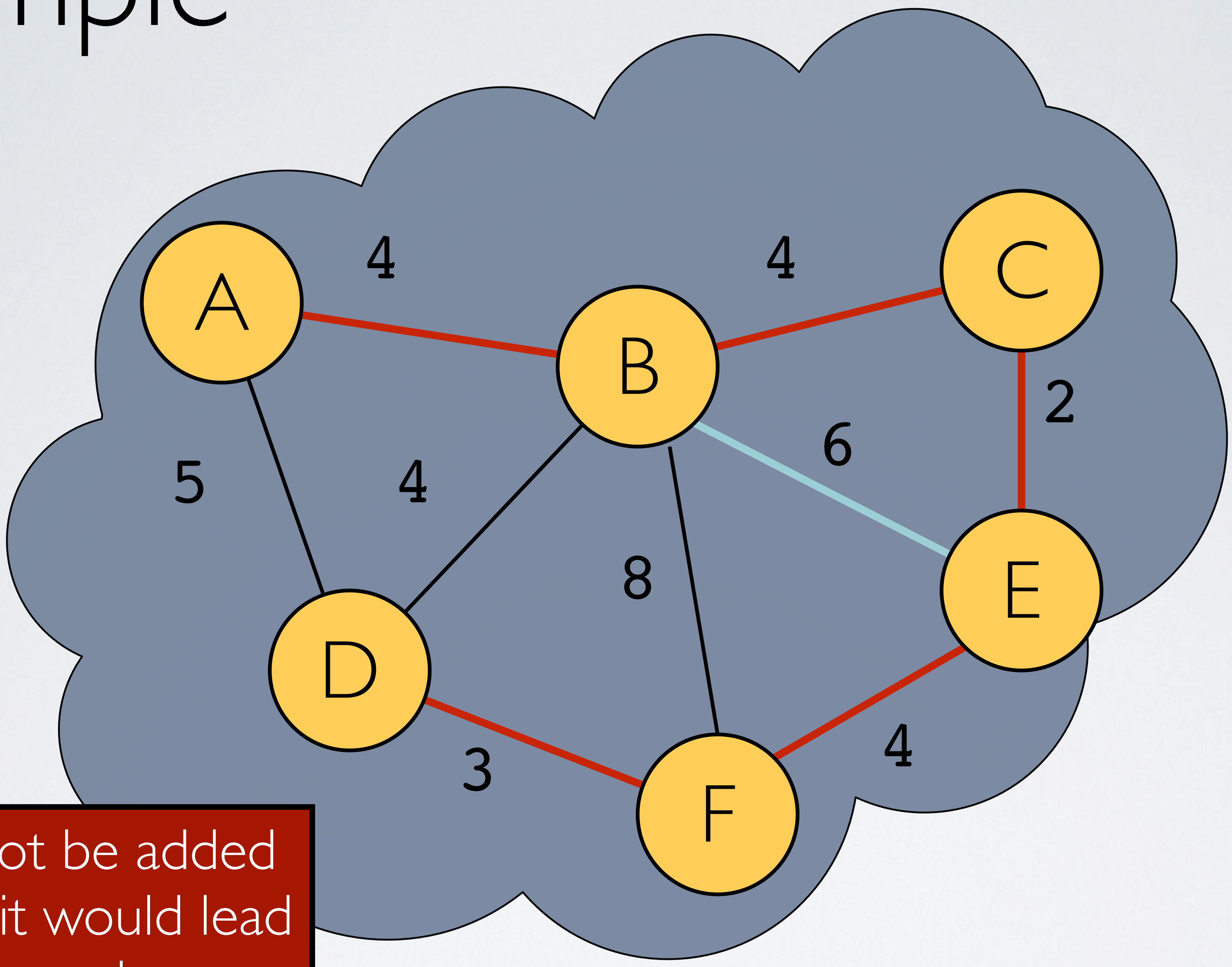
Example



AD cannot be added because it would lead to a cycle

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

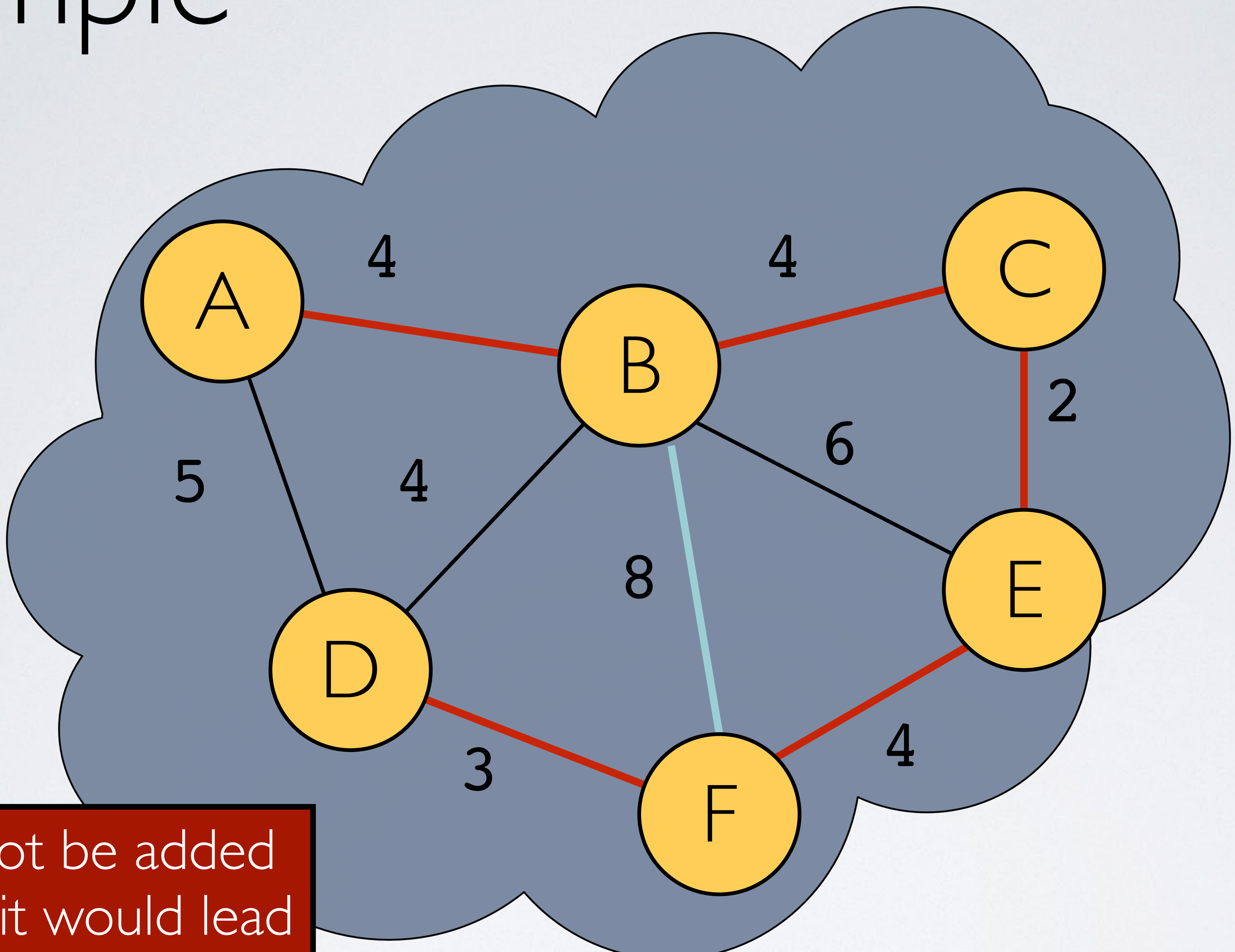
Example



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|)`

Runtime of Naive Kruskal

- 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 vertices in smaller cloud is `O(|V|)`
  - because cloud could be as big as `|V|/2` vertices
- Merge Runtime
  - `O(|V|) + O(1) + O(|V|) = O(|V|)`

Activity #4

2 min

Runtime of Naive Kruskal

- 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 vertices in smaller cloud is `O(|V|)`
  - because cloud could be as big as `|V|/2` vertices
- Merge Runtime
  - `O(|V|) + O(1) + O(|V|) = O(|V|)`

Activity #4

2 min

Runtime of Naive Kruskal

- 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 vertices in smaller cloud is `O(|V|)`
  - because cloud could be as big as `|V|/2` vertices
- Merge Runtime
  - `O(|V|) + O(1) + O(|V|) = O(|V|)`

Activity #4

1 min

Runtime of Naive Kruskal

- 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 vertices in smaller cloud is `O(|V|)`
  - because cloud could be as big as `|V|/2` vertices
- Merge Runtime
  - `O(|V|) + O(1) + O(|V|) = O(|V|)`

Activity #4

0 min

Kruskal Runtime w/ Naive Clouds

```
function kruskal(G):
    // Input: undirected, weighted graph G
    // Output: list of edges in MST
    for vertices v in G:
        makeCloud(v)
    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
```

O(|V|)

O(|E|log|E|)

O(|E|)

O(|V|)

Naive Kruskal Runtime

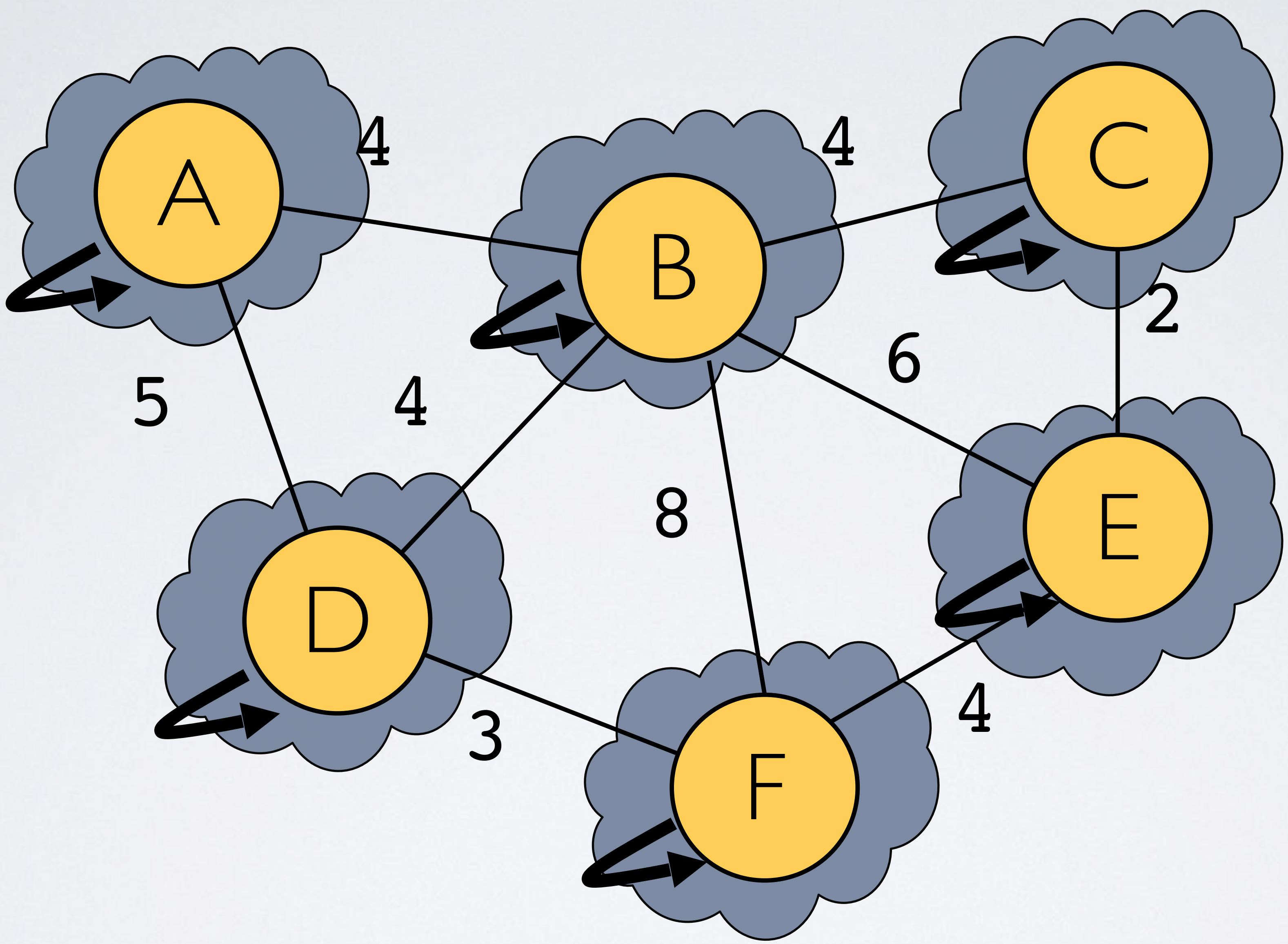
- $O(|V|)$ for iterating through vertices
- $O(|E|\log|E|)$ for sorting edges
- $O(|E|\times|V|)$ for iterating through edges and merging clouds naively
- $O(|V|+|E|\log|E|+|E|\times|V|)$
  - $= O(|E|\times|V|) = O(|V|^2 \times|V|)= O(|V|^3)$
- Can we do better?

since $|E| \leq |V|^2$

Union-Find

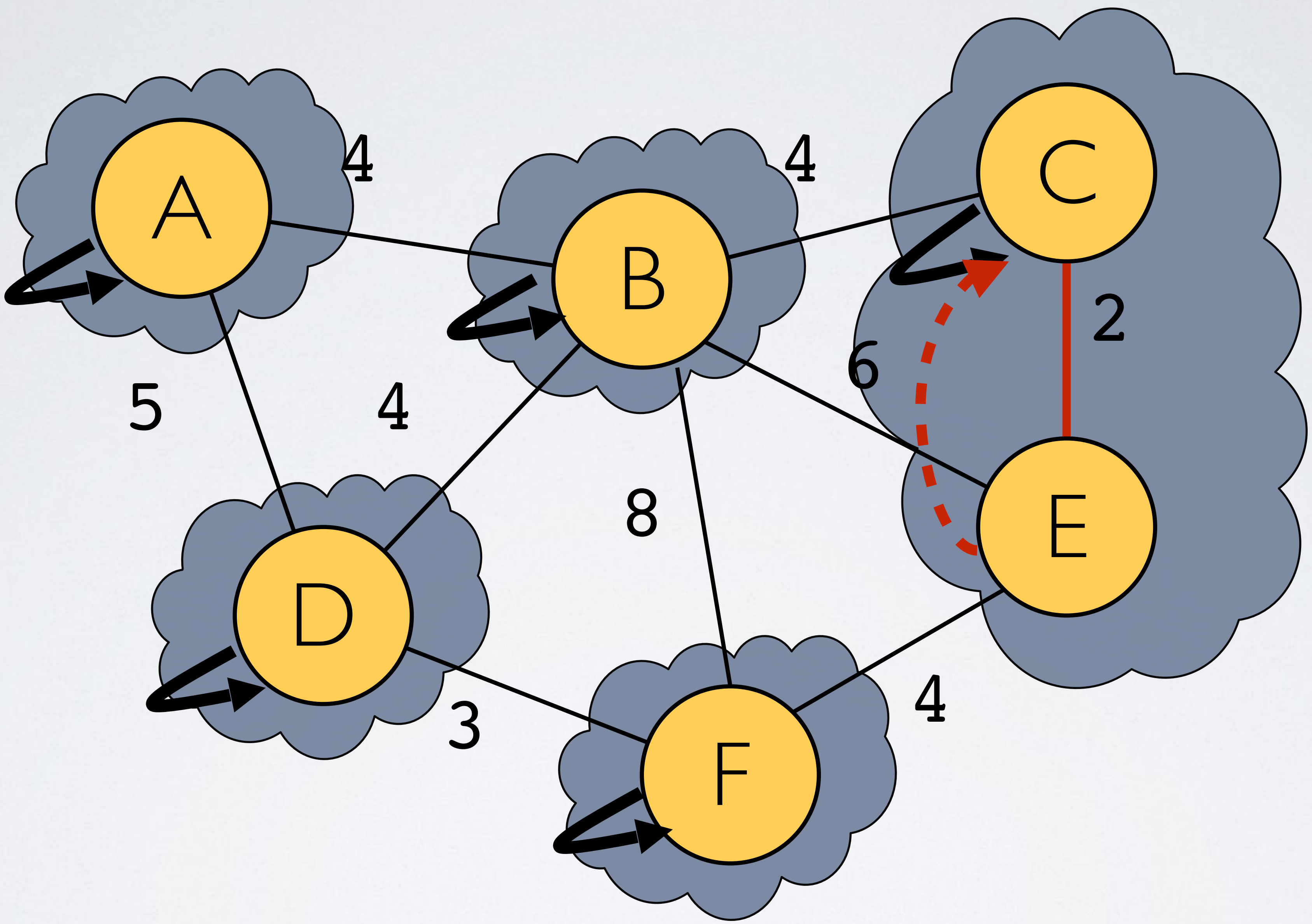
- Let's rethink notion of clouds
  - instead of labeling vertices w/ cloud numbers
  - think of clouds as small trees
- Every vertex in these trees has
  - a parent pointer that leads up to root of the tree
  - a rank that measures how deep the tree is

Example



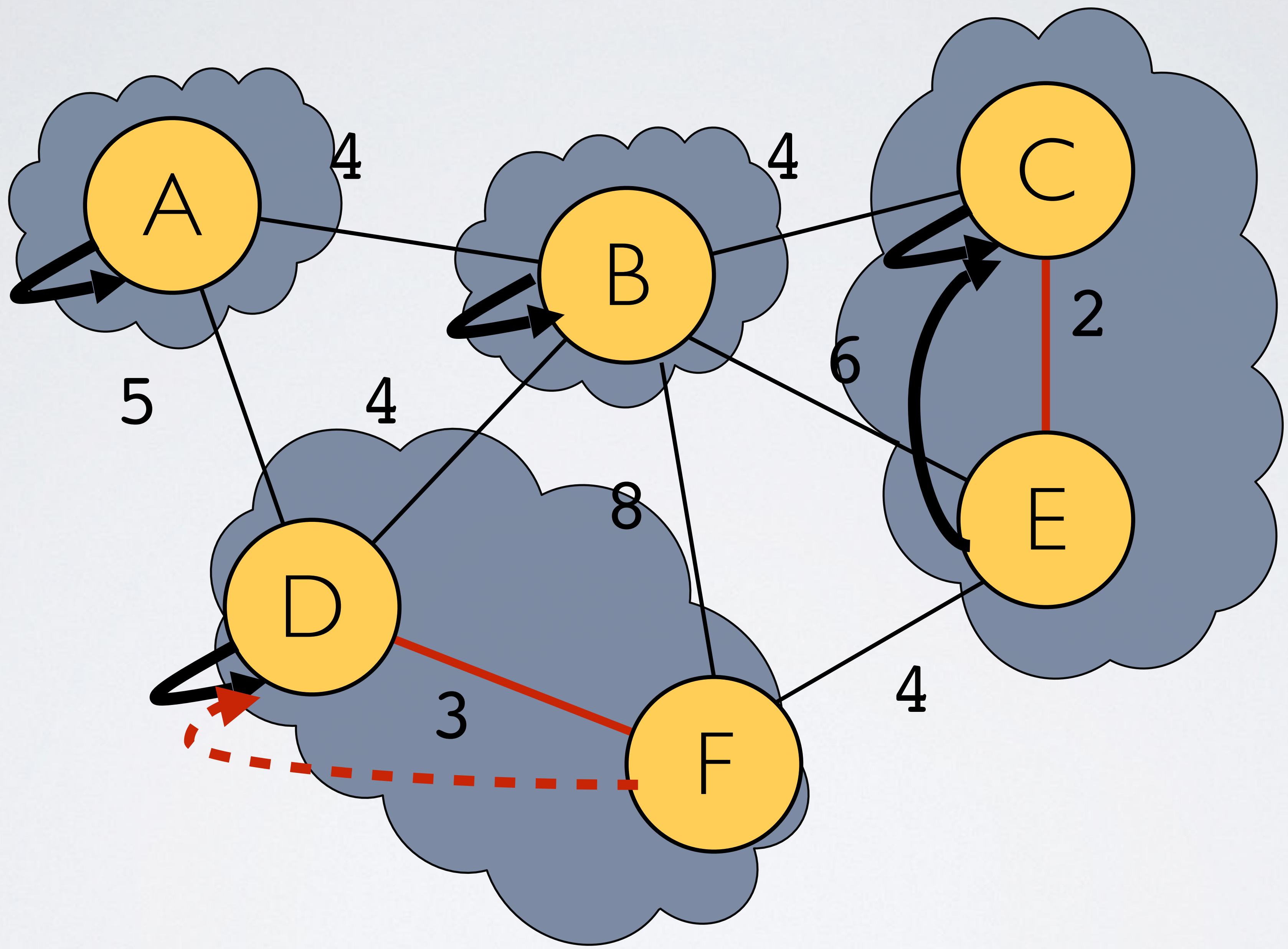

```
edges = [(C,E),(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



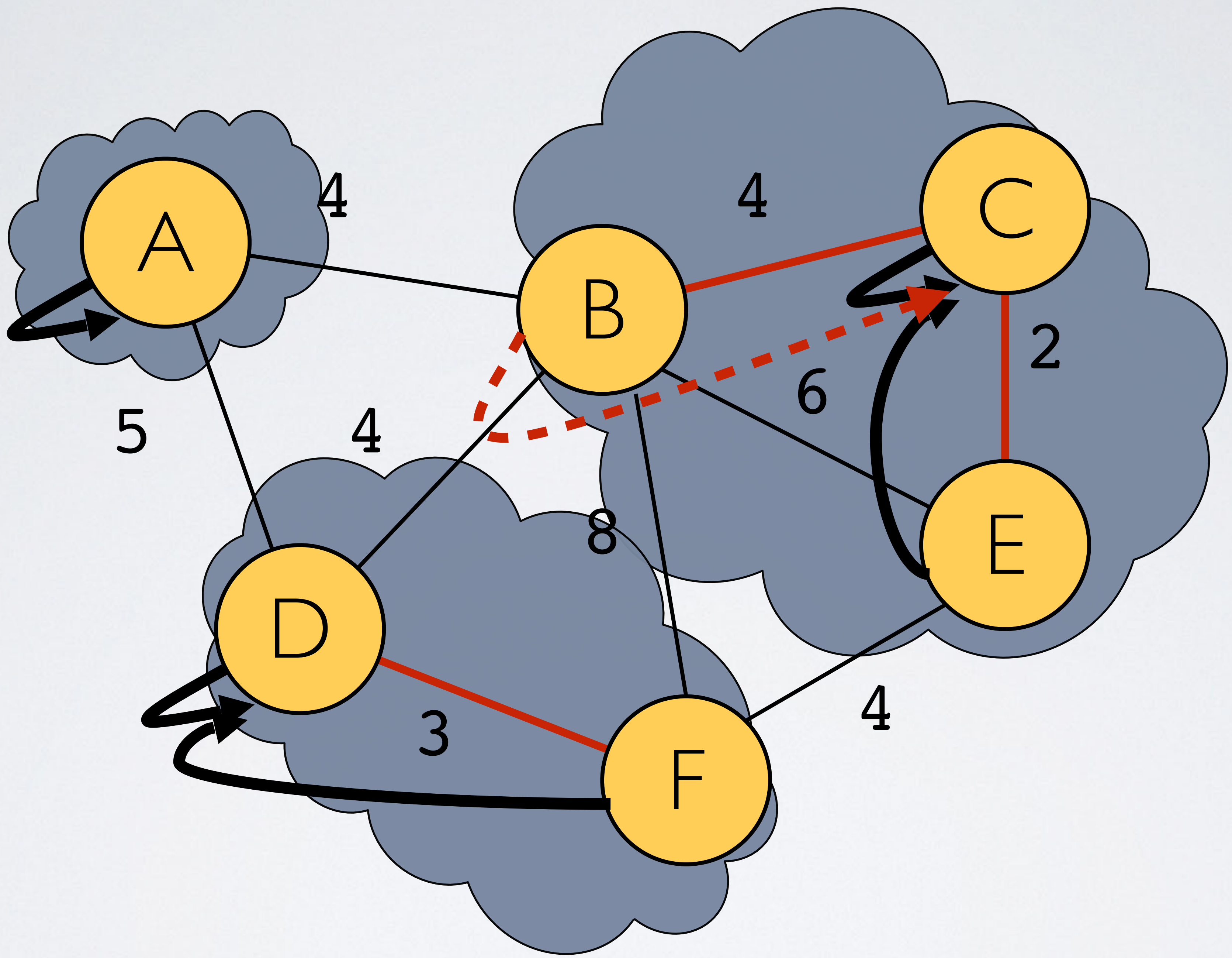

```
edges = [(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



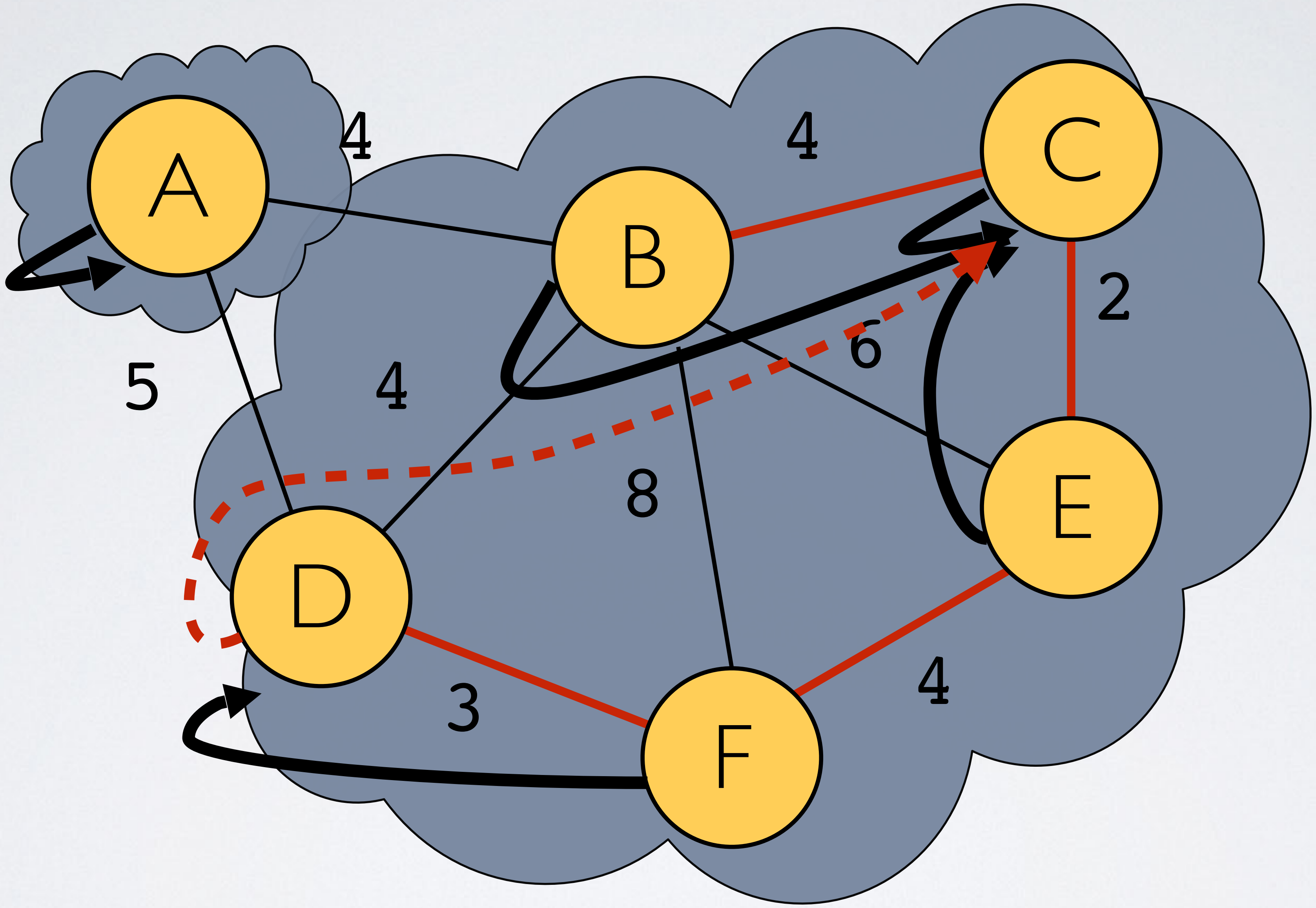

```
edges = [(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



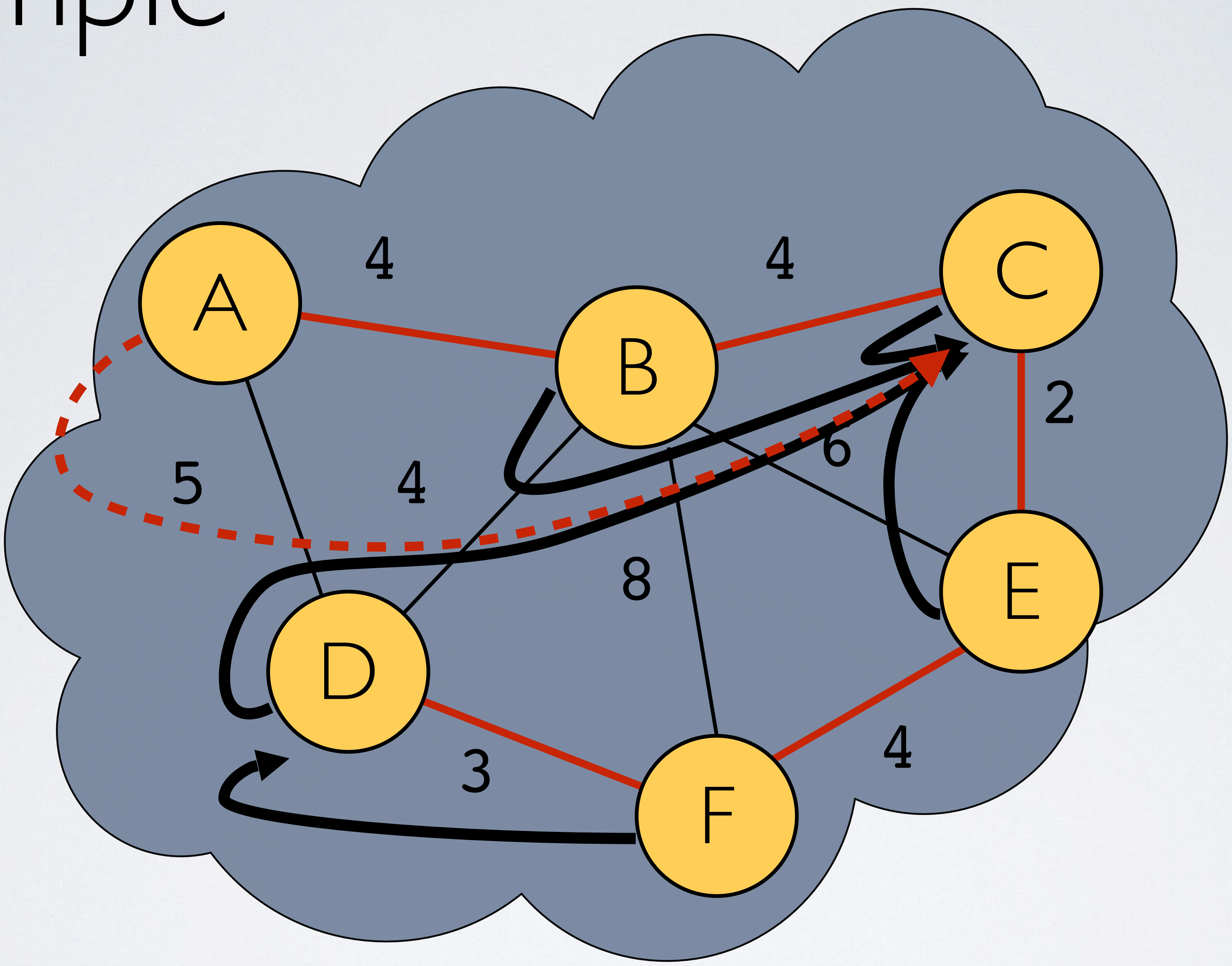
edges = [(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]

Example



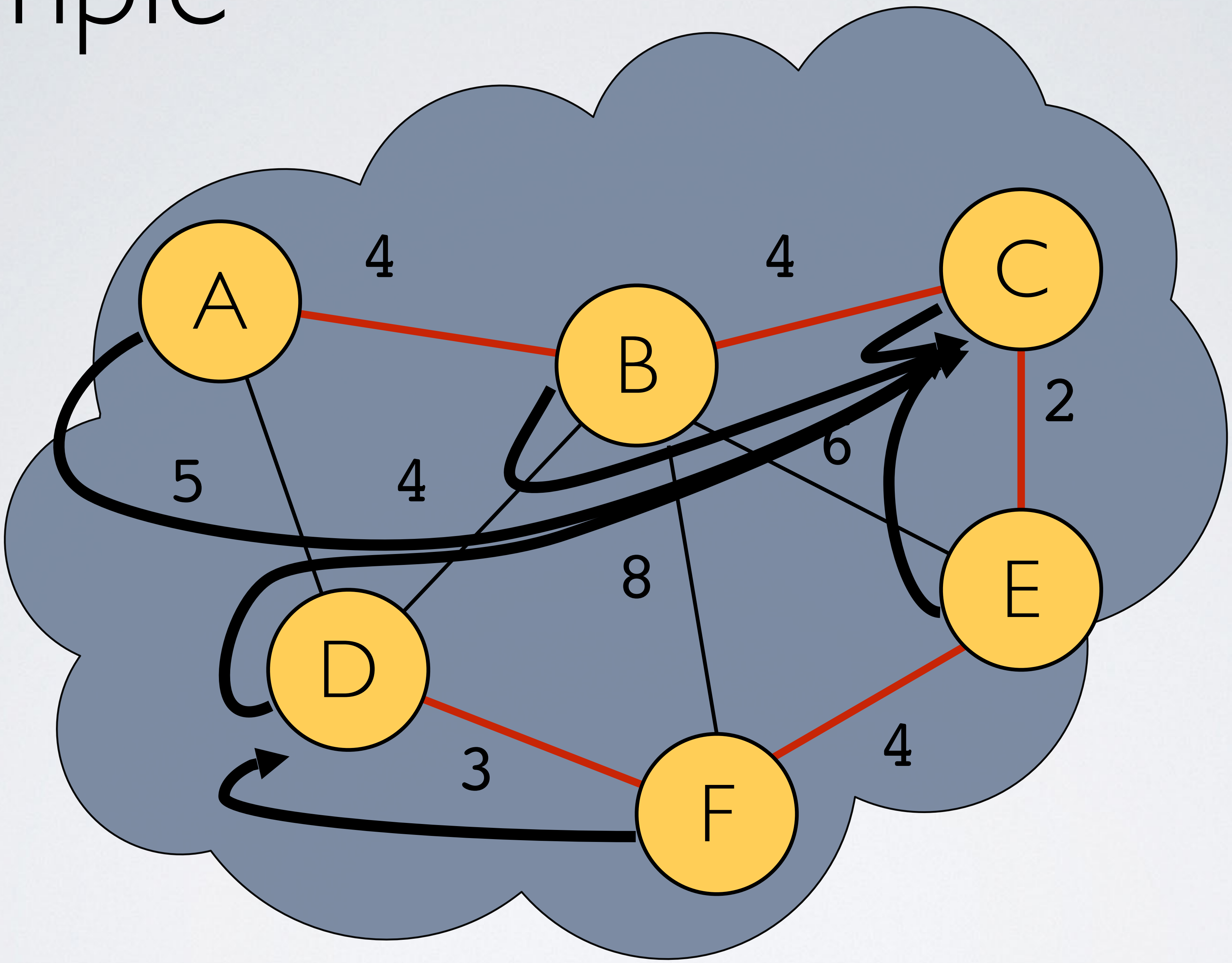
edges = [(B,D),(A,B),(A,D),(B,E),(B,F)]

Example



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

Example




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

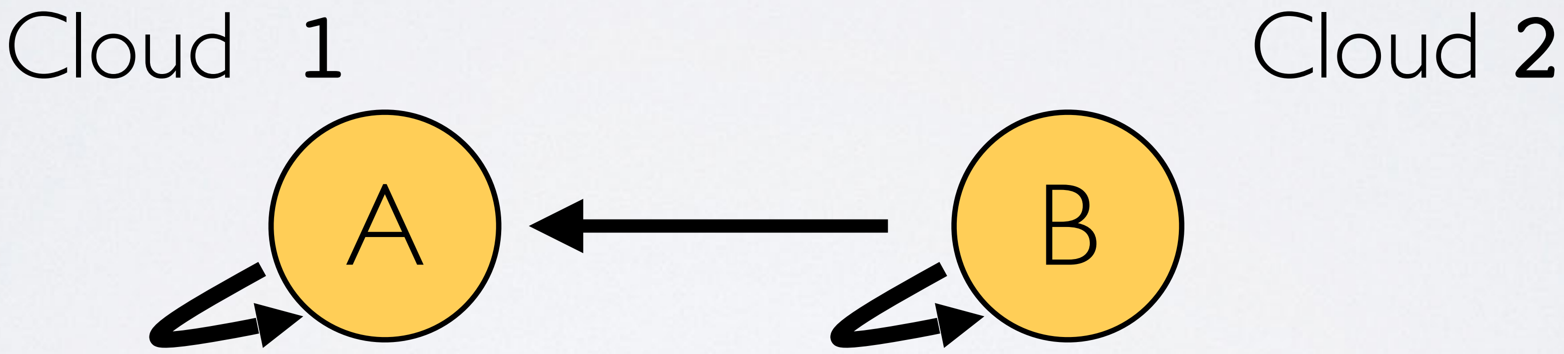
Implementing Union-Find

- 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



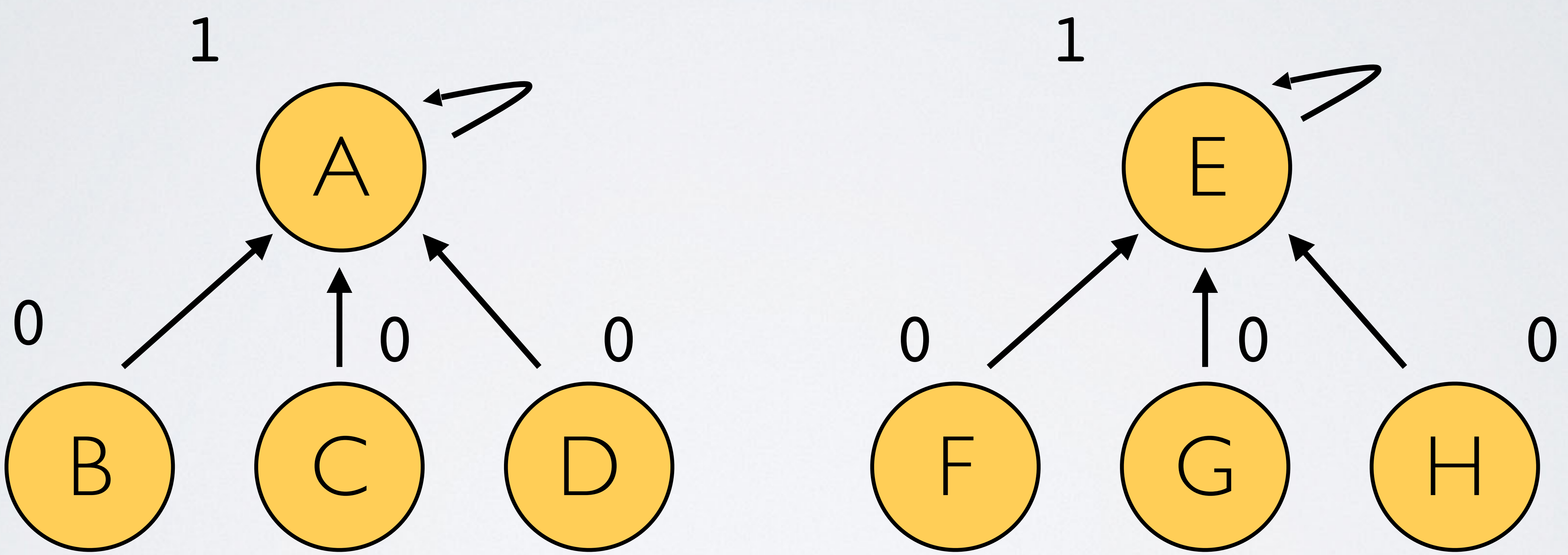
  - Given two clouds which one should point to the other?

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

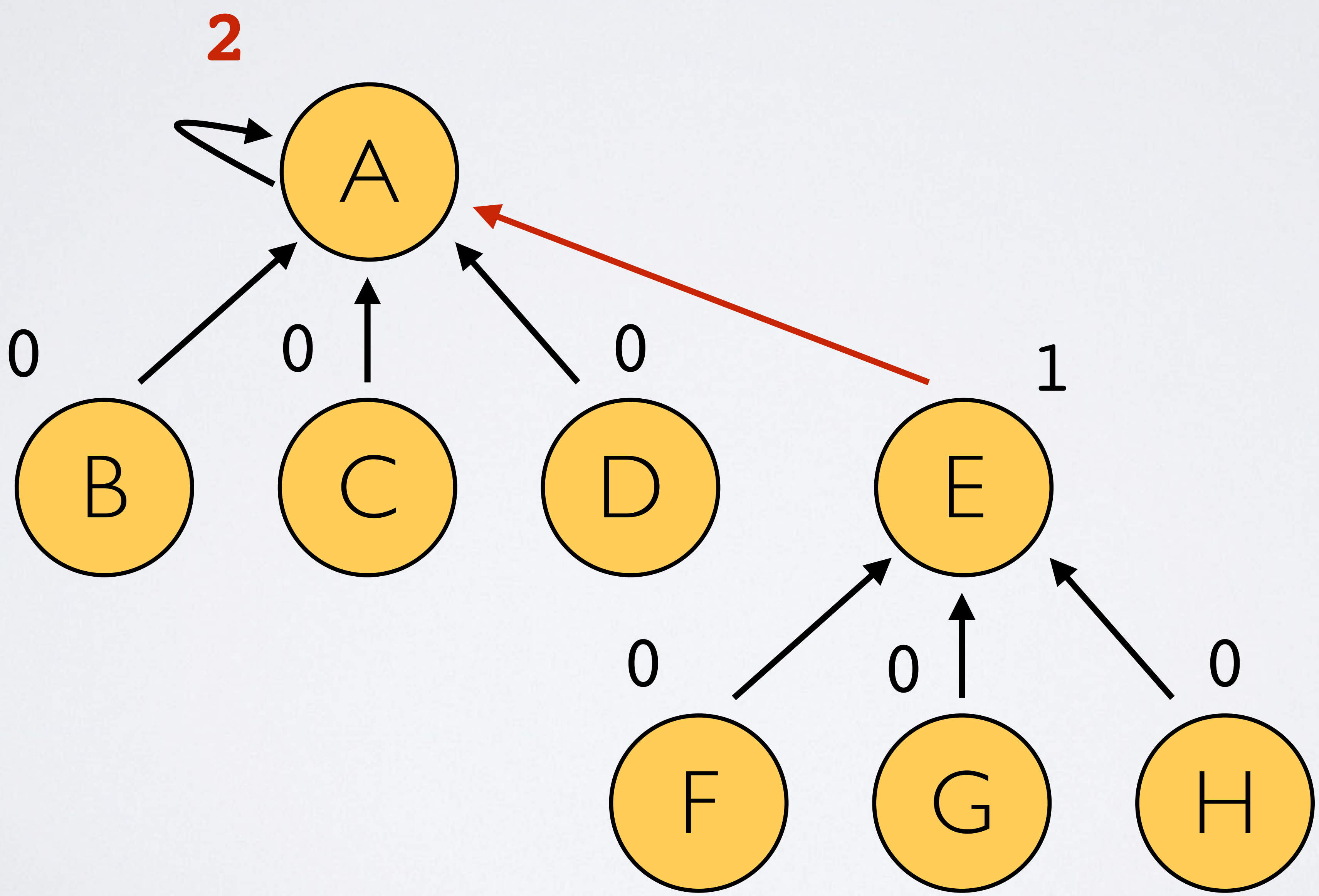
Implementing Union-Find

- Merging trees with same rank



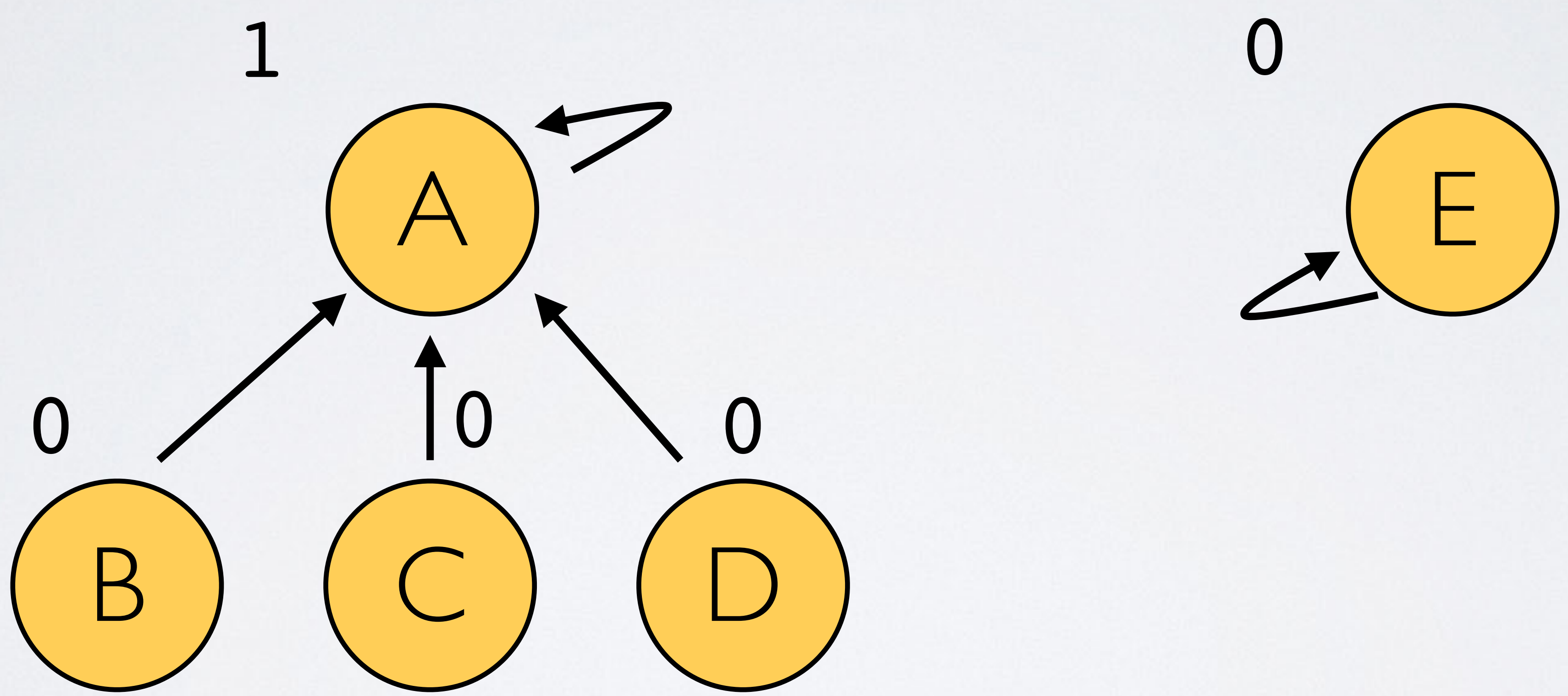
Implementing Union-Find

- Merging trees with same rank



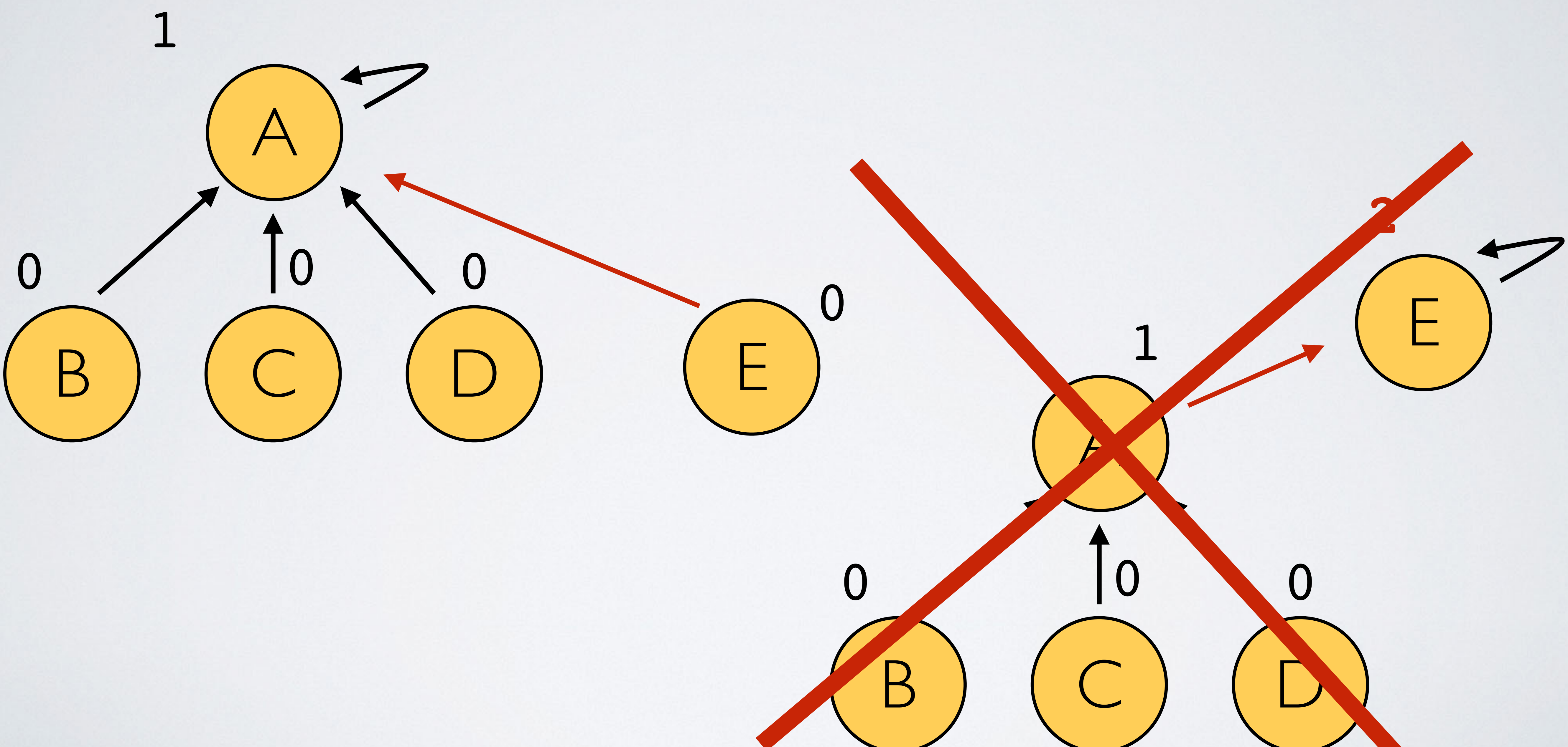
Implementing Union-Find

- Merging trees with different ranks



Implementing Union-Find

- Merging trees with different ranks



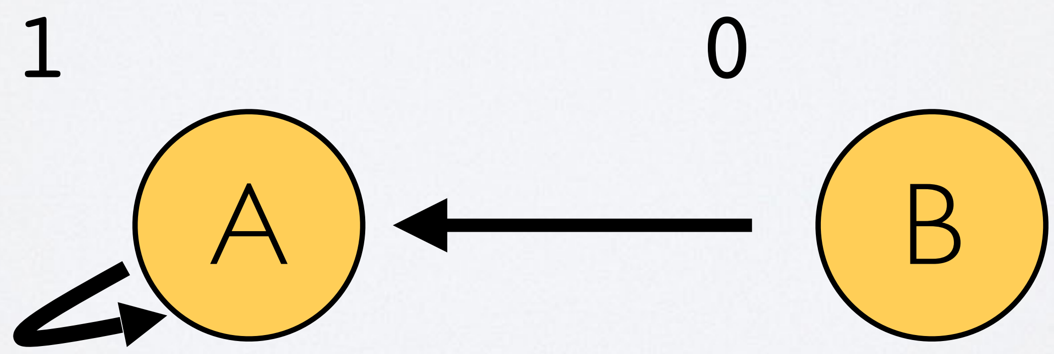
Implementing Union-Find

```
// 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
```

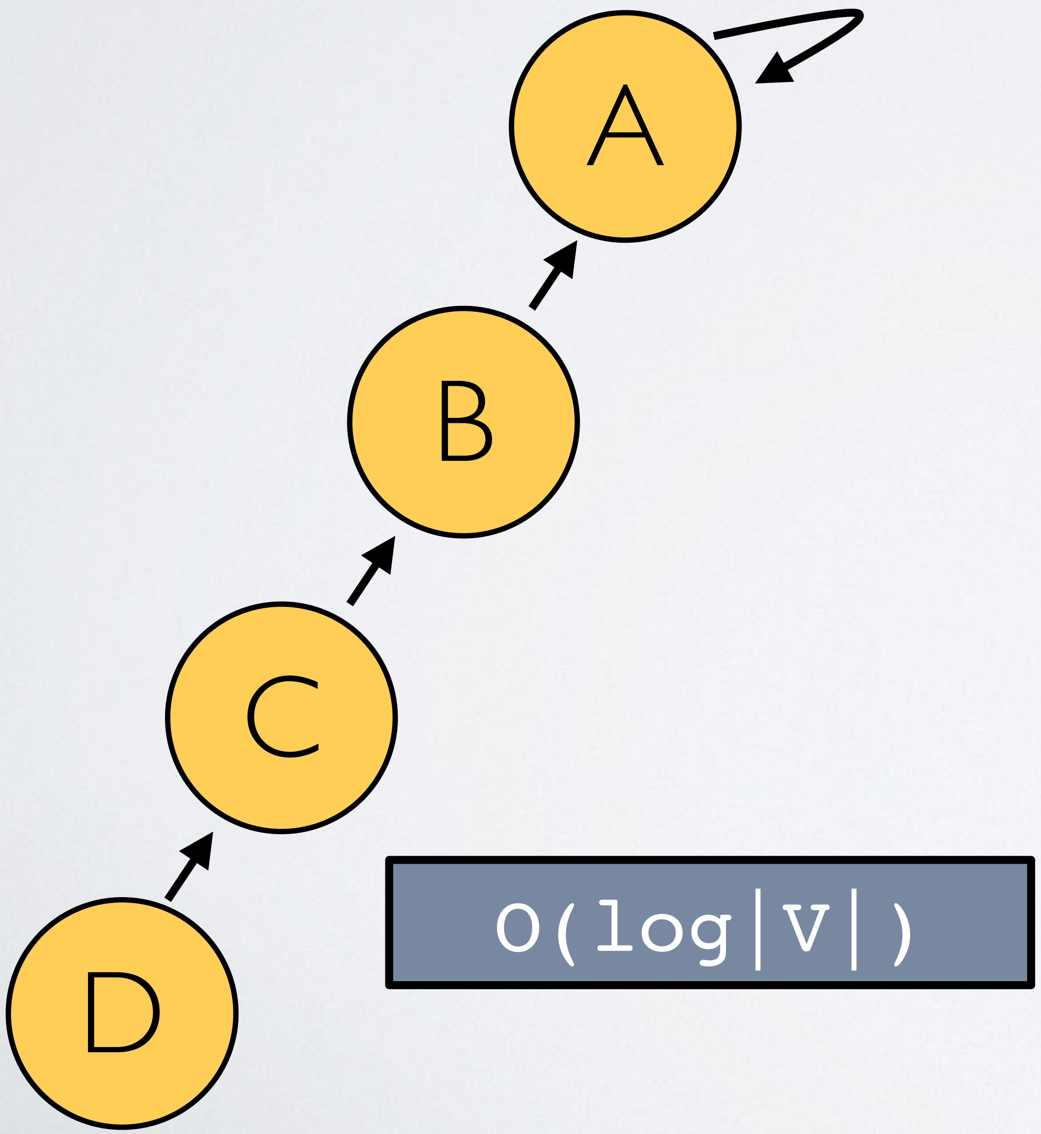


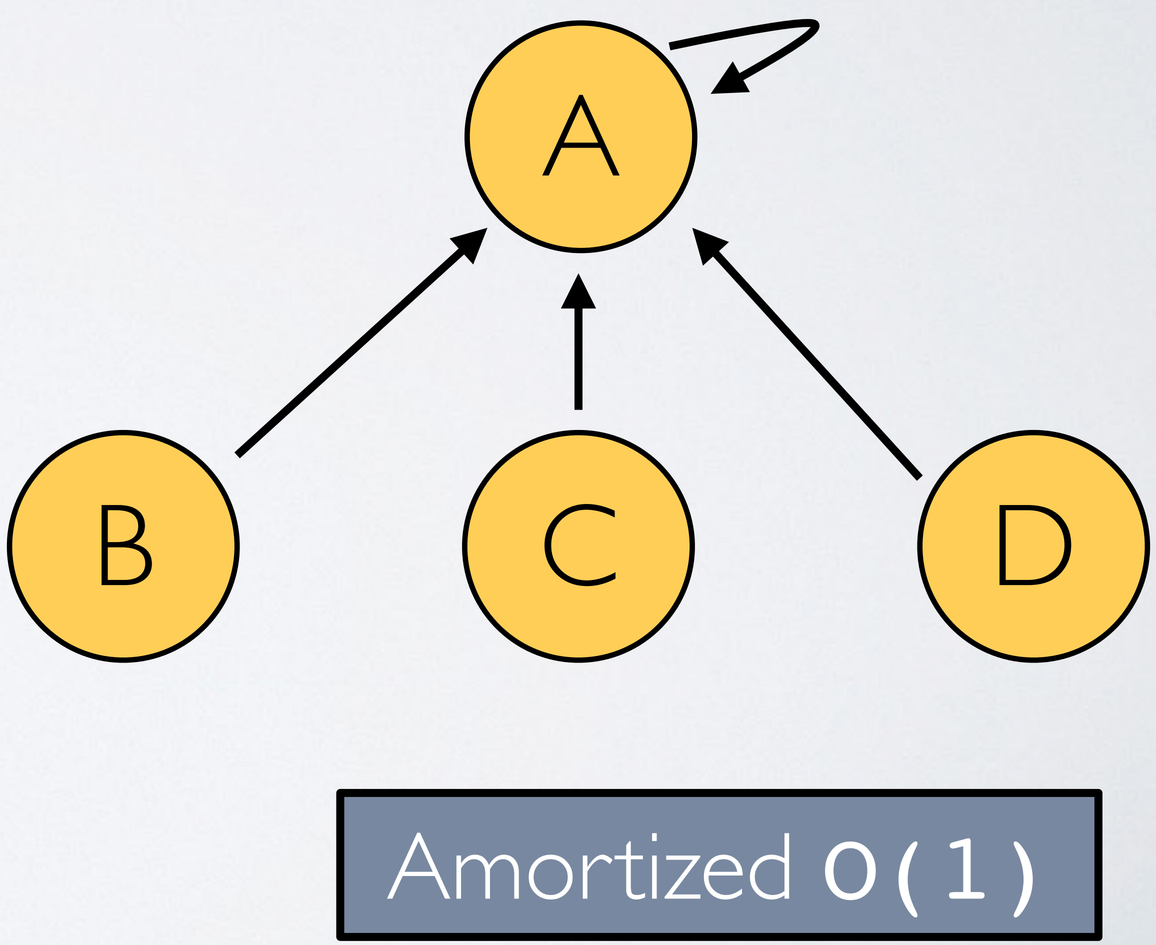
Path Compression

- 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

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

Activity #5

1 min

Runtime of Kruskal w/ Path Compression

Activity #5

1 min

Runtime of Kruskal w/ Path Compression

Activity #5

0 min

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:
            add (u,v) to MST
            merge clouds containing u and v                   O(1)
    return MST                                                amortized
```

Kruskal Runtime

- $O(|V|)$ for iterating through vertices
- $O(|E|\log|E|)$ for sorting edges
- $O(|E|\times 1)$ for iterating through edges and merging clouds with path compression
- $O(|V|+|E|\log|E|+|E|\times 1)$
  - $= O(|V|+|E|\log|E|)$
- $O(|V|+|E|\log|E|)$ better than $O(|V|^3)$

Prim’s and Kruskal’s

- Why learn two algorithms?
  - Two very different approaches for the same problem
  - MSTs are a basic building block, have been object of study for years
    - (like sorting)
- When to use?
  - Kruskals: dominated by sorting, use when already have sorted edges
  - Prim’s: better on dense graphs

Readings

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