Parallel And Sequential Data Structures and Algorithms

Star Contraction, Connectivity, Intro to MST

Announcements

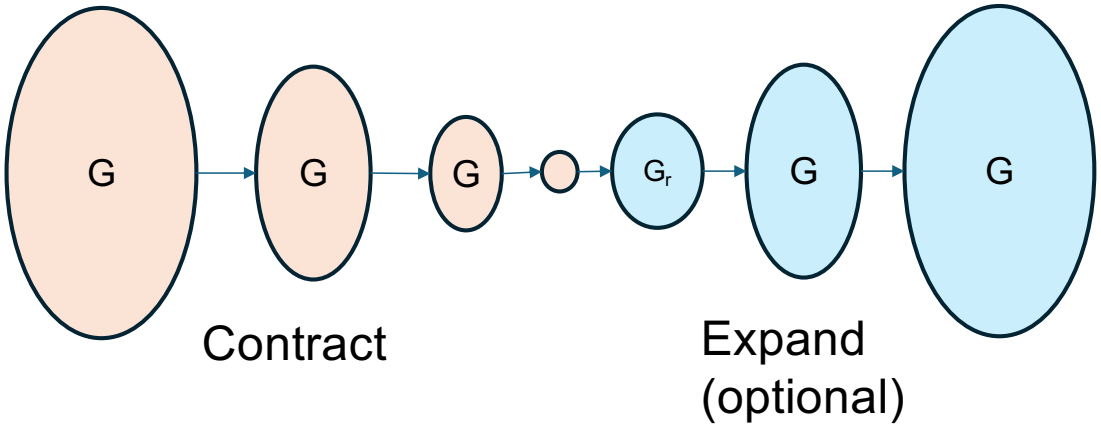
- ShortLab due today at 11:59pm
- PowerLab out tomorrow, due Monday 23rd
- **Quiz 3 on Friday**

Learning Objectives

- Implement Star Contraction
- Apply Star Contraction to Connected Components
- Introduction to Minimum Spanning Trees (MSTs)



Graph contraction: Review




```
fun graphContraction(G) :
  if base : return base(G)
  else :
    Gc = contractGraph(G)
    Gr = graphContraction(Gc)
    Ge = expandGraph(Gr)
    return Ge
```

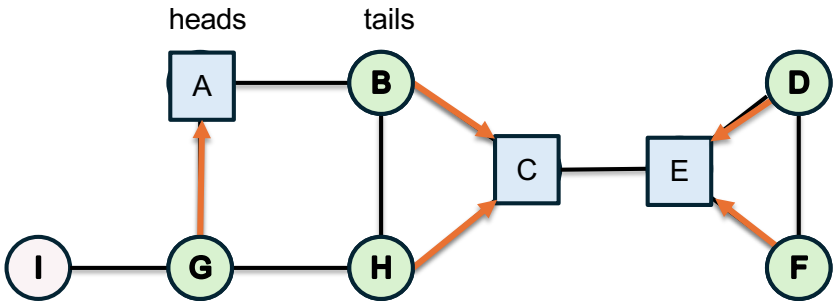
Mostly used on **undirected** graphs

Goals:

- Reduce size by constant factor
- Maintain connectivity

Star Contraction: Review

Star Idea:
1. Select a set of edges that form **independent stars**
2. Contract each star



1. Flip a coin on each vertex. A vertex is a **star center** if heads.

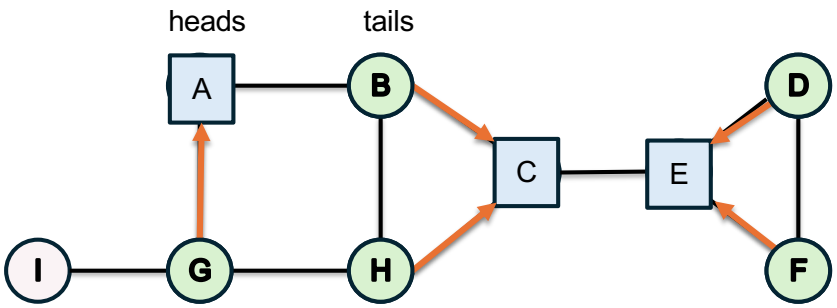
2. For each vertex that flips tails pick a neighboring heads (center) if there is one.

3. All tails that found a head are **satellites**

Star centers — Star edges — Satellite

Each satellite will be removed in contraction.

Star Contraction: Review



1. ½ probability that v flips tails
2. $1 - \left(\frac{1}{2}\right)^{\mathrm{d}}$ probability that v has a neighboring head

Hence probability that v is a satellite is:

$$\frac{1}{2}\left(1 - \left(\frac{1}{2}\right)^{\mathrm{d}}\right) \geq \frac{1}{4}$$



Implies $\geq |\mathrm{E}|/4$ edges are removed in expectation
Hence by the skittles lemma there are $\mathrm{O}(\log |\mathrm{V}|)$ rounds w.h.p.

Star Contraction : Review

Algorithm: Generic Star Contraction without expansion

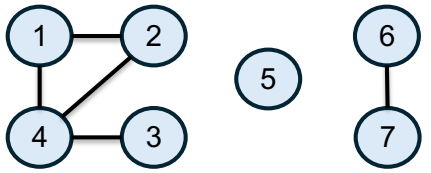
```
fun starContraction(G = (V,E)) :
  if |E| = 0 : return ?
  else :
    each vertex in V flips a coin
    each tail picks neighboring head:
       if there is one it is a satellite
    V’ = all but the satellites
    E’ = relabel edges to point to centers and remove self edges
    return starContraction(G’ = (V’,E’))
```

Graph Connectivity

Applications of Graph Contraction

- Graph connectivity (undirected graphs)

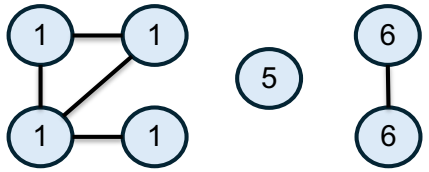
Definition (Connected Components): The maximal subgraphs that are connected.



Problem (Component Count): Return the number of connected components.

= 3

Problem (Component Labeling): Give all vertices a label such that two vertices have the same label if and only if they are in the same component.



Some choices:

Graph representation:

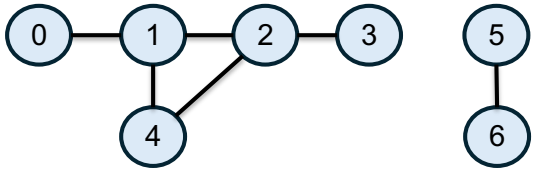
- Adjacency lists
- Sequences of edges

Vertex labels

- General (e.g. strings)
- Integers

We will use a sequence of edges, with integer labels.

Example Graph:




```
V = [0, 1, 2, 3, 4, 5, 6]
E = [(0,1),(1,2),(1,4),(2,3),(2,4),(5,6),
     (1,0),(2,1),(4,1),(3,2),(4,2),(6,5)]
```

We store the edges in both directions.

Inject: Another sequence primitive

Sidebar

Interface (Sequence): one more function

- **`inject(`**`S : sequence<T>, IVPairs : sequence<int,T>) -> sequence<T>`: return copy of `S` s.t. for each `(i, v)` in `IVPairs`, location `i` is updated with `v`.

Example:

```
inject([a, b, c, d, e], [(3,x),(0,y)]) => [y, b, c, x, e]
```

Inject: Another sequence primitive

Interface (Sequence): one more function

- **`inject(`**`S : sequence<T>, IVPairs : sequence<int,T>) -> sequence<T>`: return copy of `S` s.t. for each `(i, v)` in `IVPairs`, location `i` is updated with `v`.

```
fun inject(S : sequence<T>, ivPairs : sequence<int,T>) -> sequence<T>:
  R = allocate<T>(n)
  parallel for i in 0…(|S|-1):        // make a copy
    R[i] ← S[i]
  parallel for (i,v) in ivPairs:      // "inject" the values
    R[i] ← v
  return {R, 0, n}
```



? What is the problem with this code?

If there are duplicate indices, `R[i] ← v` is a **data race** (undefined behavior)

Inject: Another sequence primitive

```
fun inject(S : sequence<T>, ivPairs : sequence<int,T>) -> sequence<T>:
  R = allocate<T>(n)
  parallel for i in 0...(|S|-1):          // make a copy
    R[i] ← S[i]
  parallel for (i,v) in ivPairs:        // "inject" the
    atomic_write(R[i], v)               //  values
  return {R, 0, n}
```



Example (race, leads to non-determinism):

```
inject([a, b, c, d, e], [(3,x),(0,y),(0,z)]) ->  [y, b, c, x, e] or?
                                             ->  [z, b, c, x, e]
```

Inject: Another sequence primitive

Interface (Sequence): one more function

- **`inject(`**`S : sequence<T>, IVPairs : sequence<int,T>) -> sequence<T>`: return copy of `S` s.t. for each `(i, v)` in `IVPairs`, location `i` is updated with `v`.

We will allow this for this one primitive since the race is "self contained", but it means **inject** is not a "function".

In C++: the racy parallel loop:

```
parallel_for(0, ivpairs.size(), [&](int j) {
  auto&& [i, v] = ivpairs[j];
  std::atomic_ref(S[i]).store(v);  // S[i] = v; has undefined behavior
}
```

Flipping Coins

Will use the function `Heads(v, i)` to "flip coins". Here v is a vertex and i is a round number.

- Can be thought as a hash function to true, false
- Or the coin that gets flipped at v on round i

CountComponents with star contraction

```
fun countComponents(V : sequence<int>, E : sequence<int,int>,
                    P : sequence<int>, i : int) -> int :
  if |E| == 0 : |V|
  else :
     hooks = [(u,v) for (u,v) in E if not Heads(u,i) and Heads(v,i)]
     P' = inject(P, hooks)
     V' = [v for v in V if P'[v] = v]
     E' = [(P'(u),P'(v)) for (u, v) in E if P'(u) ≠ P'(v)]
  return countComponents(V', E', P', i+1)
```

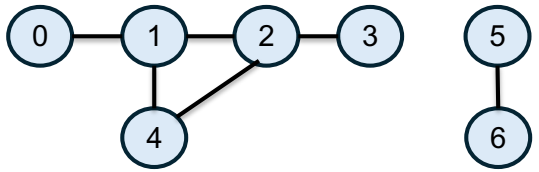
From tail to head

Find Stars

Contract

Recurse

```
V = [0, 1, 2, 3, 4, 5, 6]
P = [0, 1, 2, 3, 4, 5, 6]   // "centers"
E = [(0,1),(1,2),(1,4),(2,3),(2,4),(5,6),
     (1,0),(2,1),(4,1),(3,2),(4,2),(6,5)]
```



CountComponents with star contraction

```
fun countComponents(V, E, P, i) :
  if |E| == 0 : |V|
  else :
    hooks = [(u,v) for (u,v) in E if
              not Heads(u,i) and Heads(v,i)]
    P' = inject(P, hooks)
    V' = [v for v in V if P'[v] = v]
    E' = [(P'(u),P'(v)) for
          (u, v) in E if P'(u) ≠ P'(v)]
  return countComponents(V', E', P', i+1)
```

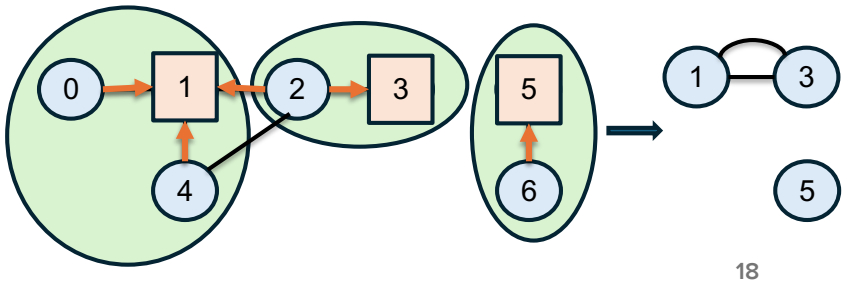
Find Stars

Contract

From tail to head

```
heads = [1,3,5]
hooks = [(0,1),(2,3),(2,1),(4,1),(6,5)]
P' = [1, 1, 3, 3, 1, 5, 5]
V' = [1, 3, 5]
E' = [(1,3),(3,1),(1,3),(3,1)]
```

```
V = [0, 1, 2, 3, 4, 5, 6]
P = [0, 1, 2, 3, 4, 5, 6]   // "centers"
E = [(0,1),(1,2),(1,4),(2,3),(2,4),(5,6),
     (1,0),(2,1),(4,1),(3,2),(4,2),(6,5)]
```



CountComponents with star contraction

```
fun countComponents(V, E, P, i) :
  if |E| == 0 : |V|
  else :
    hooks = [(u,v) for (u,v) in E if
              not Heads(u,i) and Heads(v,i)]
    P' = inject(P, hooks)
    V' = [v for v in V if P'[v] == v]
    E' = [(P'(u),P'(v)) for
          (u, v) in E if P'(u) ≠ P'(v)]
  return countComponents(V', E', P', i+1)
```

Find Stars

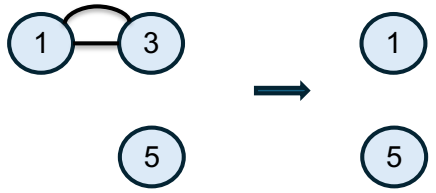
Contract

From tail to head

```
heads = [1]
hooks = [(3,1),(3,1)]
P' = [1, 1, 3, 1, 1, 5, 5]
V' = [1, 5]
E' = []
```

Second round:

```
V = [1, 3, 5]
P = [1, 1, 3, 3, 1, 5, 5]   // "centers"
E = [(1,3),(3,1),(1,3),(3,1)]
```



Done!

CountComponents with star contraction

```
fun labelComponents(V : sequence<int>, E : sequence<int,int>,
                    P : sequence<int>, i : int) -> int :
  if |E| == 0 : P
  else :
     hooks = [(u,v) for (u,v) in E if not Heads(u,i) and Heads(v,i)]
     P' = inject(P, hooks)
     V' = [v for v in V if P'[v] = v]
     E' = [(P'(u),P'(v)) for (u, v) in E if P'(u) ≠ P'(v)]
     R  = labelComponents(V', E', P', i+1)
  return [R[p] for p in P']
```

Each satellite reads R from its center

Find Stars

Contract

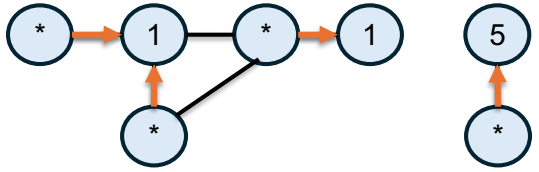
Recurse

Expand

```
R = [*, 1, *, 1, *, 5, *]
```

```
P' = [1, 1, 3, 3, 1, 5, 5]
```

```
Return= [1, 1, 1, 1, 1, 5, 5]
```



Cost of Count/Label Components

Cost of one round

Step	Work	Span
hooks = filter	$O(\|E\|)$	$O(\log \|E\|)$
P' = inject	$O(\|P\|)$	$O(1)$
V' = filter	$O(\|V\|)$	$O(\log \|V\|)$
E' = filter	$O(\|E\|)$	$O(\log \|E\|)$
Expand	$O(\|P\|)$	$O(1)$
Total:	$O(\|V\| + \|P\| + \|E\|)$	$O(\log(\|V\|)$

|V| decreases geometrically
|P| stays at original n
|E| ?

|E| only goes down by ¼$|V|$

Cost across rounds

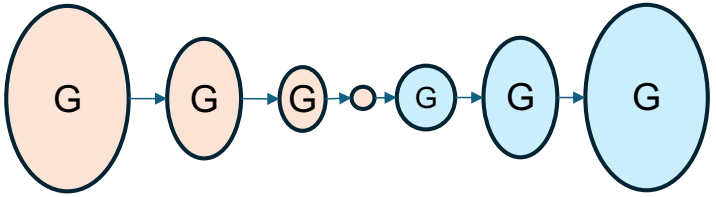
Round	Work	Span
0	$O(m+n)$	$O(\log n)$
1	$O(m+n+¾n)$	$O(\log n)$
2	$O(m+n+(3/4)^2 n)$	$O(\log n)$
…		
Total:	$O\big((n+m)\log n\big)$	$O(\log^2 n)$

How about a forest (no cycles)? Can we argue the number of edges goes down?

Yes: Total work can be $O(n)$

Summary on Graph Contraction

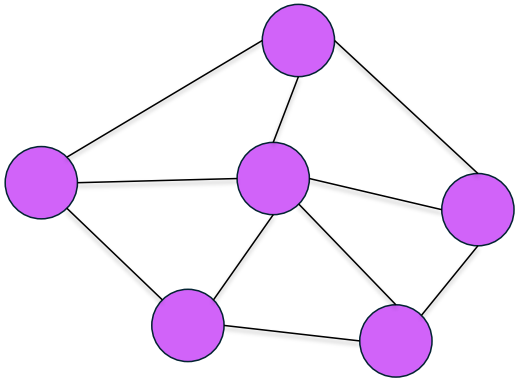
- Graph contraction identifies vertex disjoint connected pieces of a graph and contracts each into a single vertex.
- This is repeated until no edges remain.
- Has many applications
- Edge contraction is good for certain graphs: e.g $O(\mathrm{m})$ work and $O(\log^2 n)$ span on a cycle graph
- Star contraction works with all graphs, but is $O\big((n+m)\log n\big)$ work (also $O(\log^2 n)$ span).
- Sometimes use expansion

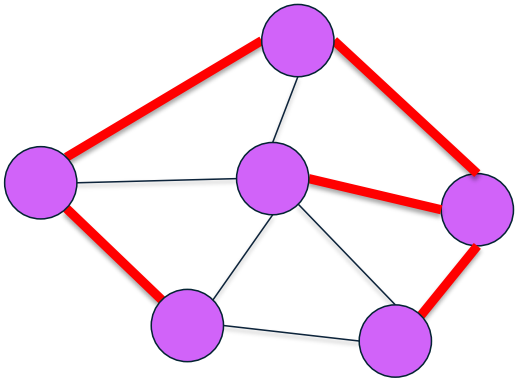


Minimum Spanning Trees

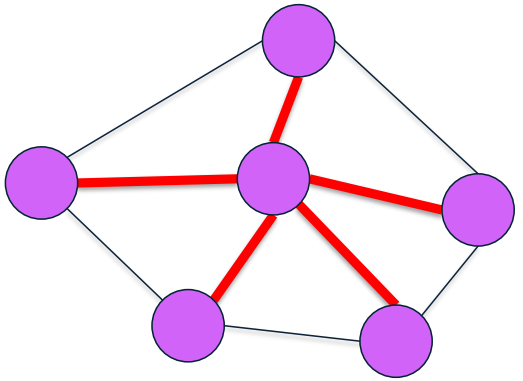
Spanning Trees

Definition (Spanning Tree): A spanning tree of an undirected, connected graph $G = (V, E)$ is a tree on V, i.e., a connected graph $T = (V, E')$ where $E' \subseteq E$.





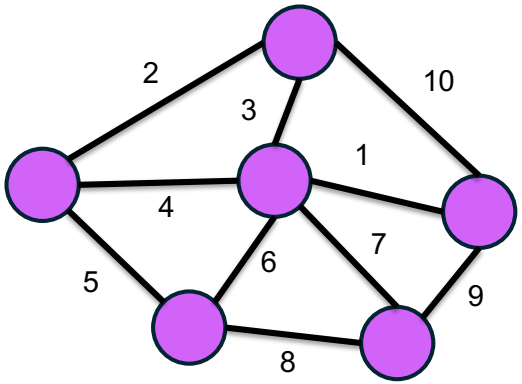
Spanning Tree 1

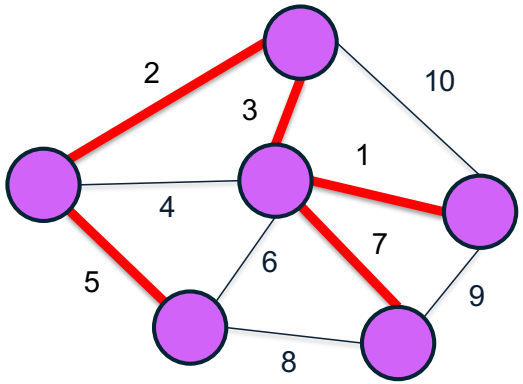


Spanning Tree 2

Minimum Spanning Tree

Definition (Minimum Spanning Tree): A minimum spanning tree (MST) of a weighted, undirected, connected graph is a spanning tree of minimum possible total weight





Uniqueness of Edge Weights

- **MSTs are easier if we assume the edge weights are unique**
- **But this sounds like a substantial loss of generality!**

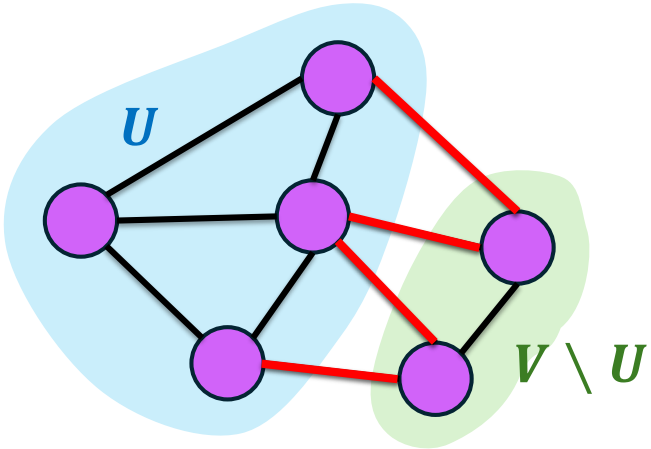
We can assume the edge weights are unique **without loss of generality**

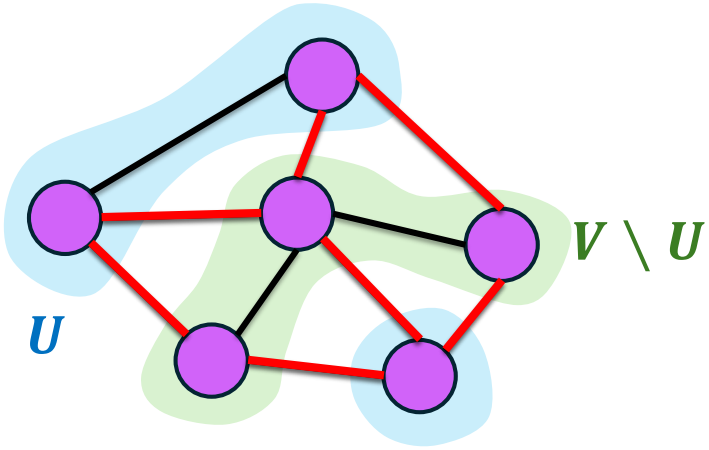
- Just introduce a tiebreaker for edge weights
- For example, give each edge a label from 0 to $m - 1$
- Tiebreak equal weights by considering the label

Theorem (Unique MST): In a weighted undirected graph with unique edge weights, there exists a unique MST

Properties of MSTs

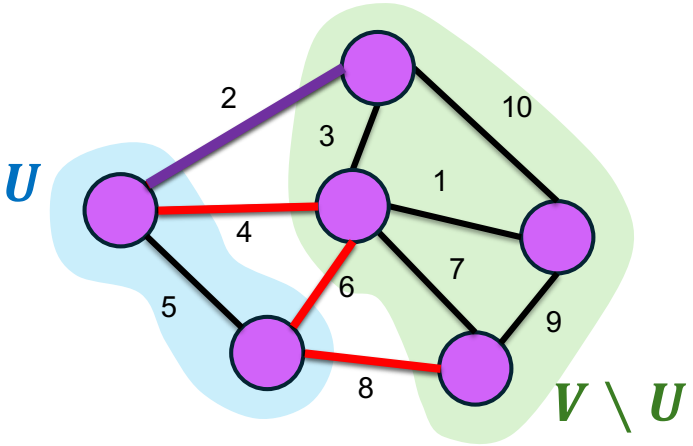
Definition (Cut): In an undirected graph $G = (V, E)$, a **cut** is a partition of the vertices V into two non-empty disjoint subsets U and $V \setminus U$. An edge $(u, v) \in E$ **crosses** the cut if $u \in U$ and $v \notin U$.





Properties of MSTs

Theorem (Light-Edge/Cut/Blue Rule): In a weighted undirected graph $G = (V, E)$, for **any** cut $(U, V \setminus U)$, the lightest edge crossing the cut must be in the minimum spanning tree of G.



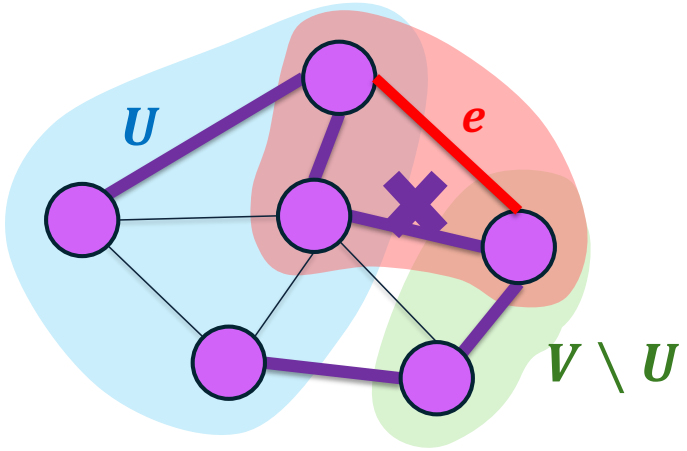
If there are duplicate edge weights, **for any lightest edge**, there exists an MST containing it.

Properties of MSTs

Theorem (Light-Edge/Cut/Blue Rule): In a weighted undirected graph $G = (V, E)$, for **any** cut $(U, V \setminus U)$, the lightest edge crossing the cut must be in the minimum spanning tree of G.

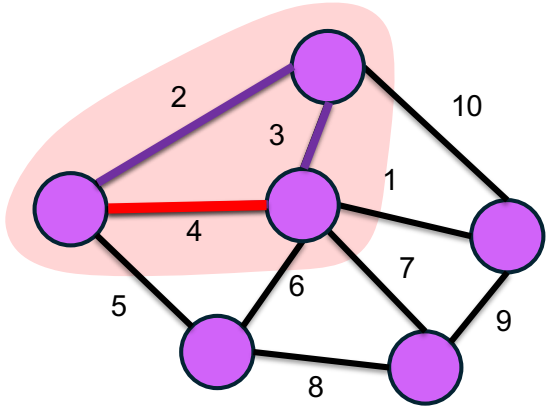
Proof: by contradiction

- *Assume* e *is the lightest edge across* $(U, V \setminus U)$*, and* T *is the MST and does not contain* e
- *Consider the graph* $T \cup \{e\}$*, it contains a cycle!*
- *Let* e' *in be an edge in the cycle that crosses the cut. There is at least one cycle edge that crosses the cut.*
- *Replace* e' *with* e *and we still have a spanning tree*
- *But since* $w(e) < w(e')$ *it has a lower weight*
- *Therefore, T is not an MST, which is a contradiction.*



Properties of MSTs

Theorem (Heavy-Edge/Cycle/Red Rule): In a weighted undirected graph $G = (V, E)$, for **any** cycle in G, the heaviest edge on the cycle is not in the MST of G.



If there are duplicate edge weights, **for any heaviest edge**, there exists an MST not containing it

Proof: Very similar to the light-edge rule

- *Also, you saw it in 15-122*