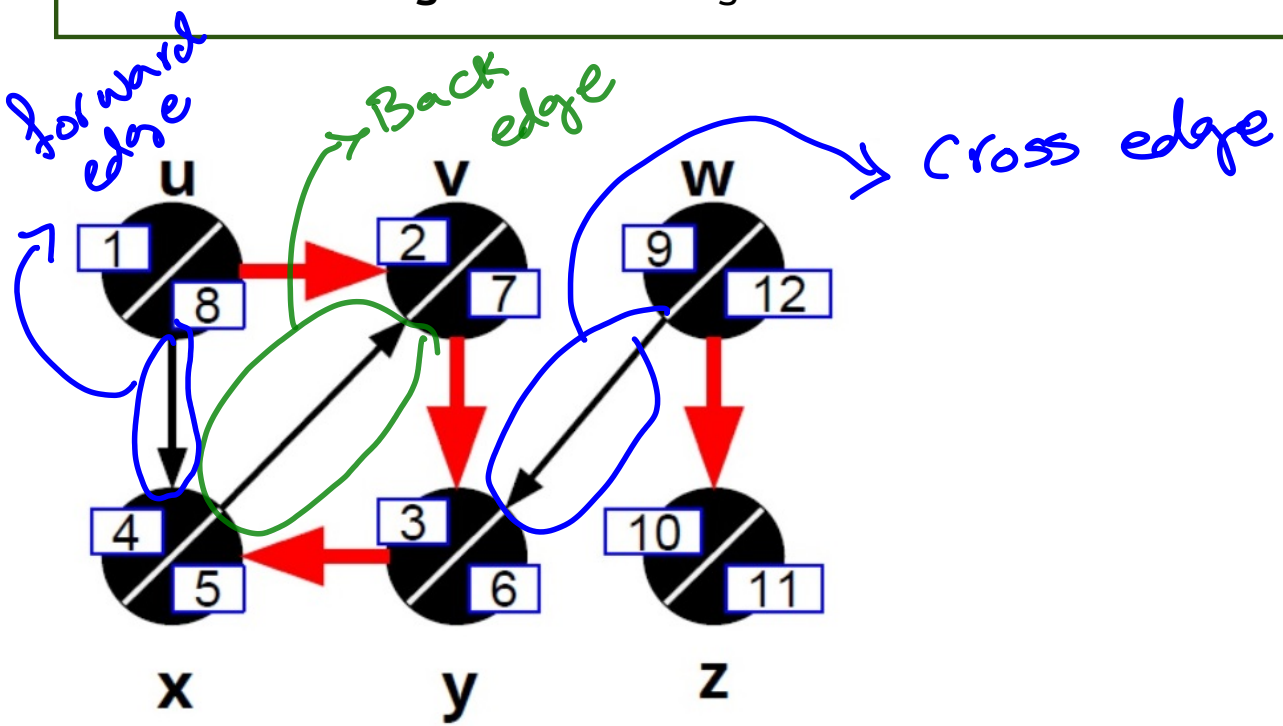
CSC263H
Data Structures and Analysis

Prof. Bahar Aameri & Prof. Marsha Chechik

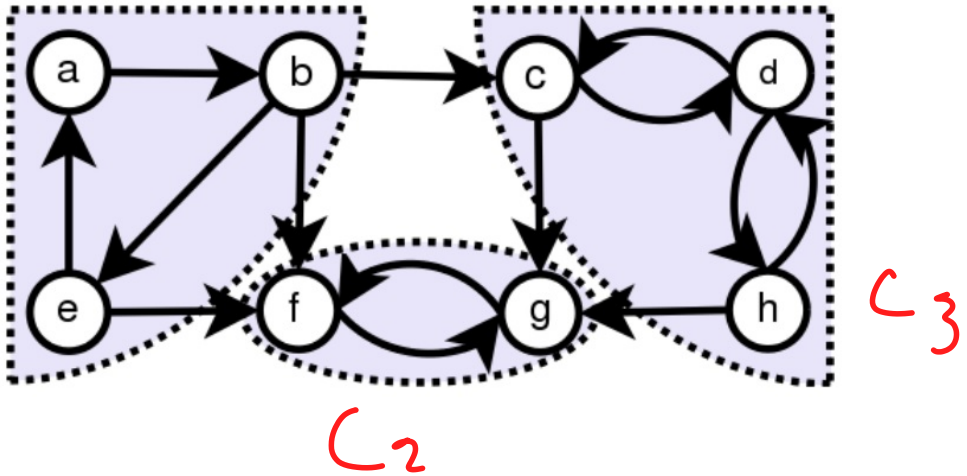
Winter 2024 – Week 10

- **Tree edge**: an edge in the DFS-forest
- **Back edge**: a **non-tree** edge pointing from a vertex to its **ancestor** in the DFS forest.
- **Forward edge:** a **non-tree** edge pointing from a vertex to its **descendant** in the DFS forest.
- **Cross edge:** all other edges.



Finding Strongly Connected Components

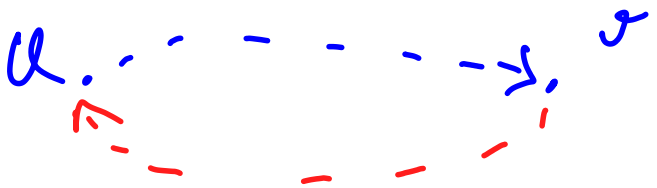
A **strongly connected component** of a directed graph $G = \langle V, E \rangle$ is a maximal set of vertices C, such that for every pair of vertices u and v, there's path from u to v and a path from v to u, i.e. vertices v and u are reachable from each other.



C_1

C_2

C_3



$C_1 = \{a, b, e\}$

$C_2 = \{f, g\}$

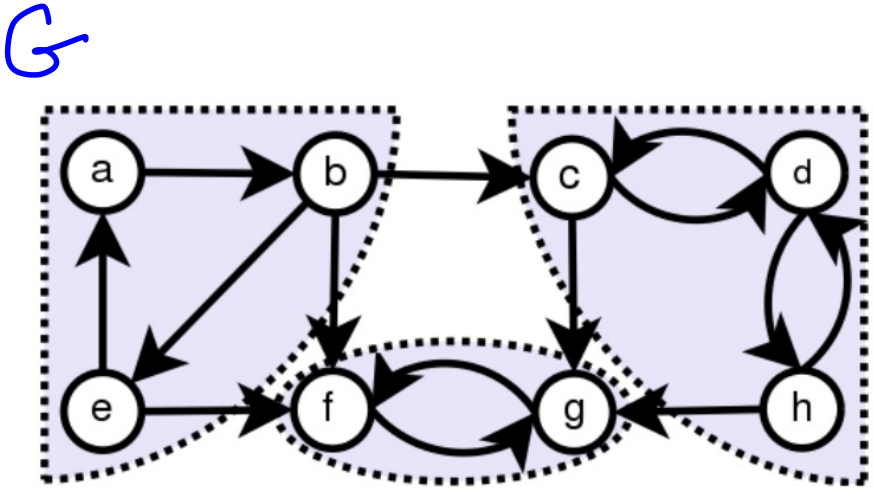
Finding Strongly Connected Components

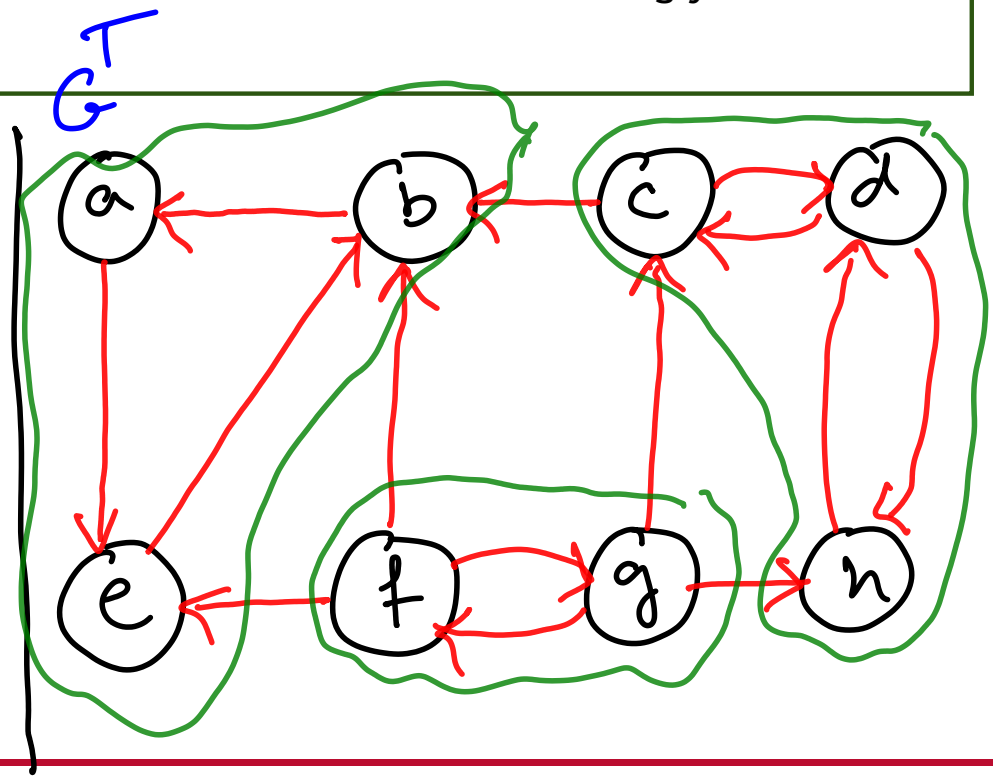
- The transpose of be a directed graph $G = (V, E)$ is the graph $G^{reverse} = (V, E^{r}everse)$, where

$$E^{reverse} = \{(v, u) \in V \times V : (u, v) \in E\}$$

That is, $G^{reverse}$ is G with all edges reversed.

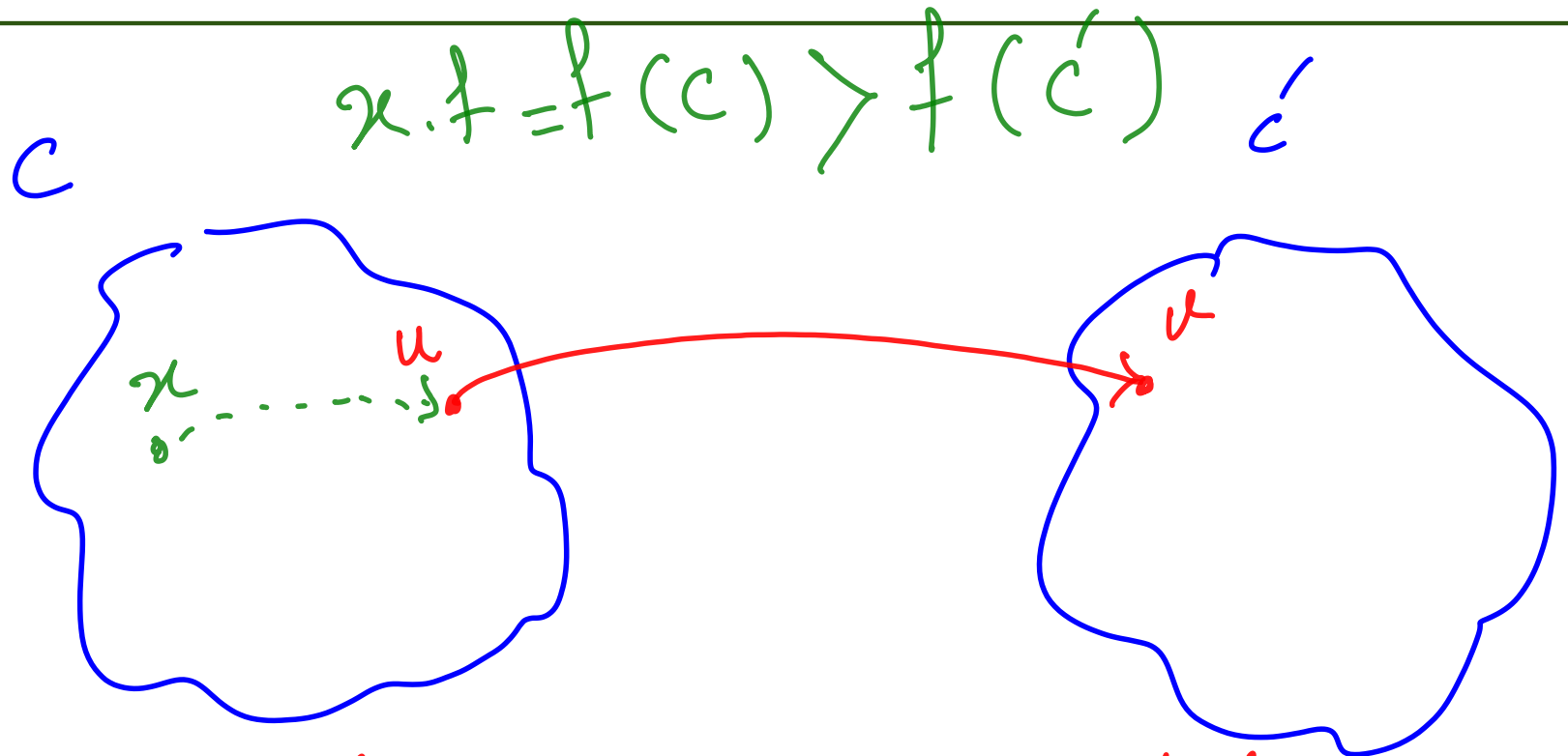
- A directed graph G and its transpose $G^{reverse}$ have the same Strongly Connected Components.





Finding Strongly Connected Components

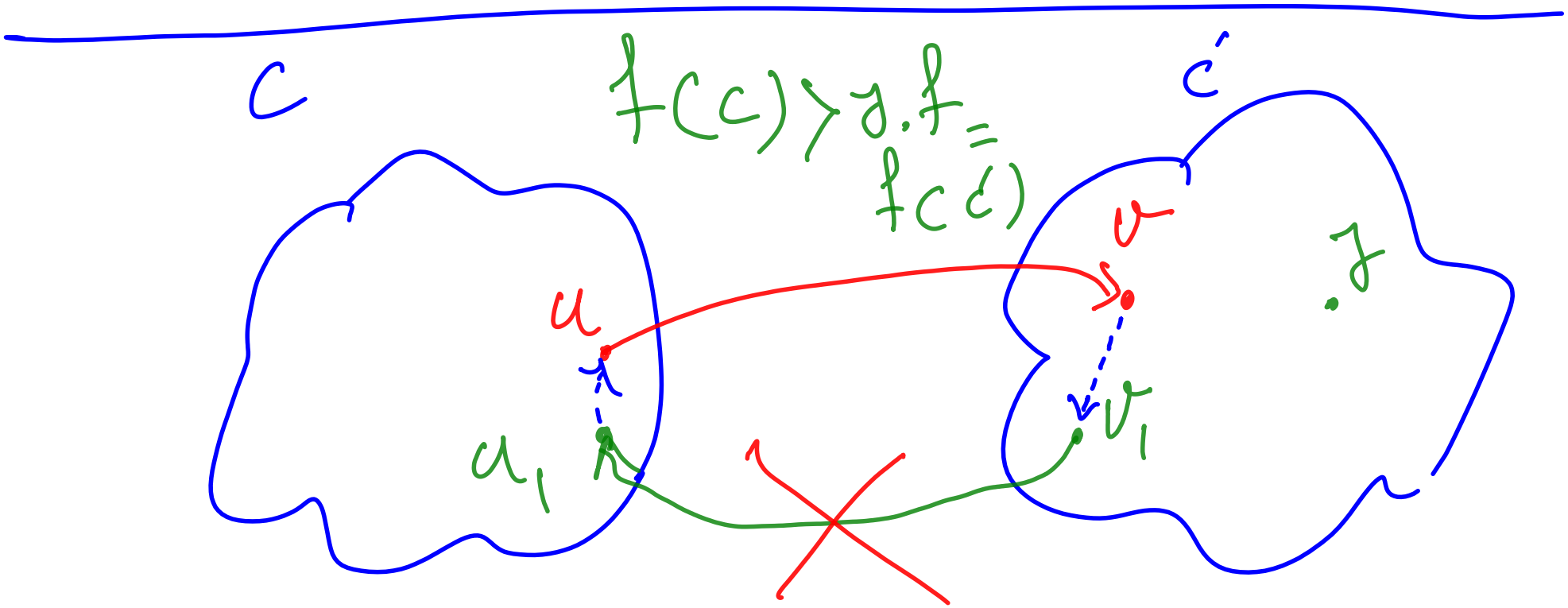
Let C and C' be distinct SCC's in $G = \langle V, E \rangle$. Suppose there is an edge $(u, v) \in$ E such that $u \in C$ and $v \in C'$.
Then $f(C) > f(C')$, where $f(U) = max_{u \in U}\{u.f\}$
(latest finishing time of any vertex in U).



Case 1: DFS discovers a vertex in C before discovering any vertex in C'

Case 2: N " " " " C' "

" " " " " C

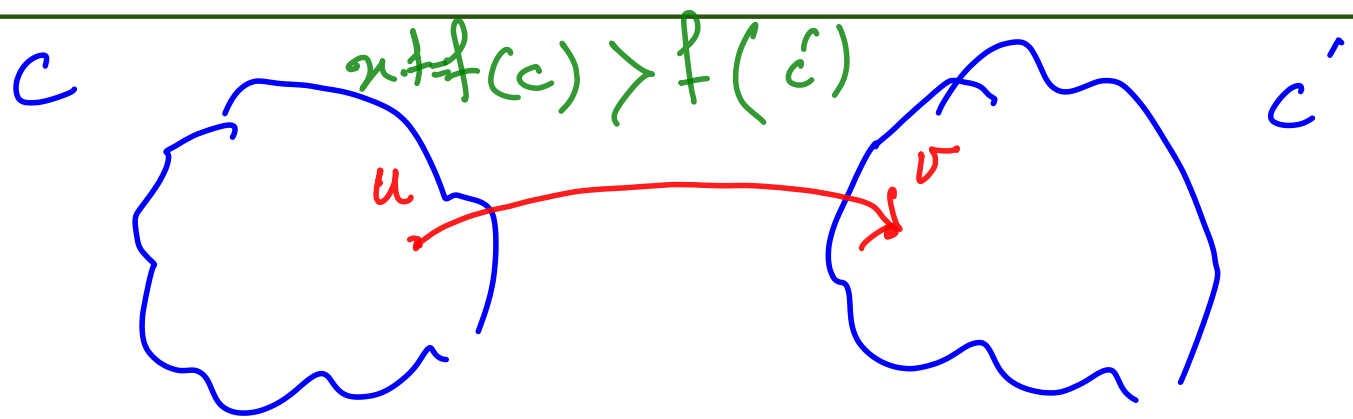


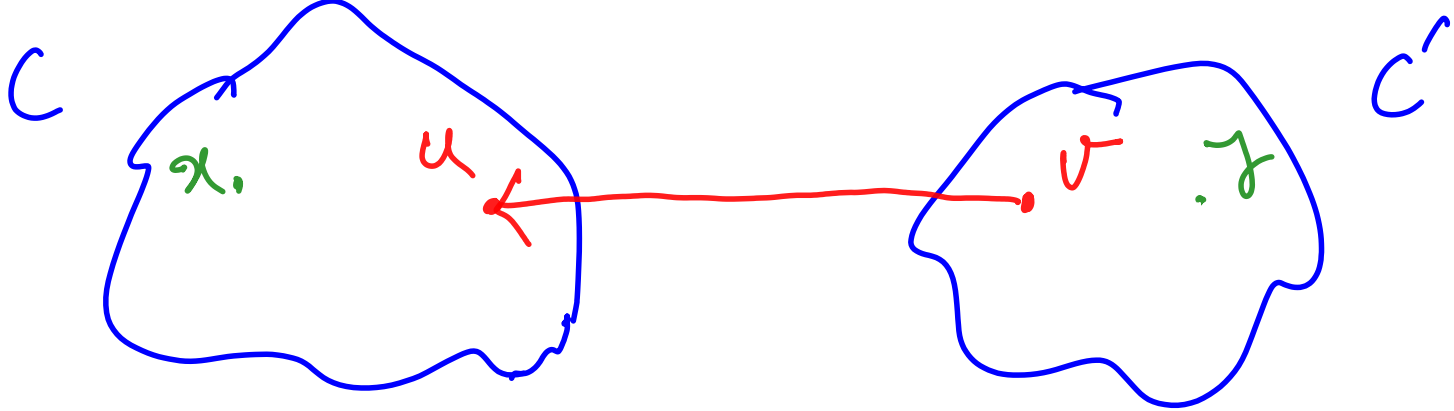
Complete proof in CLRS

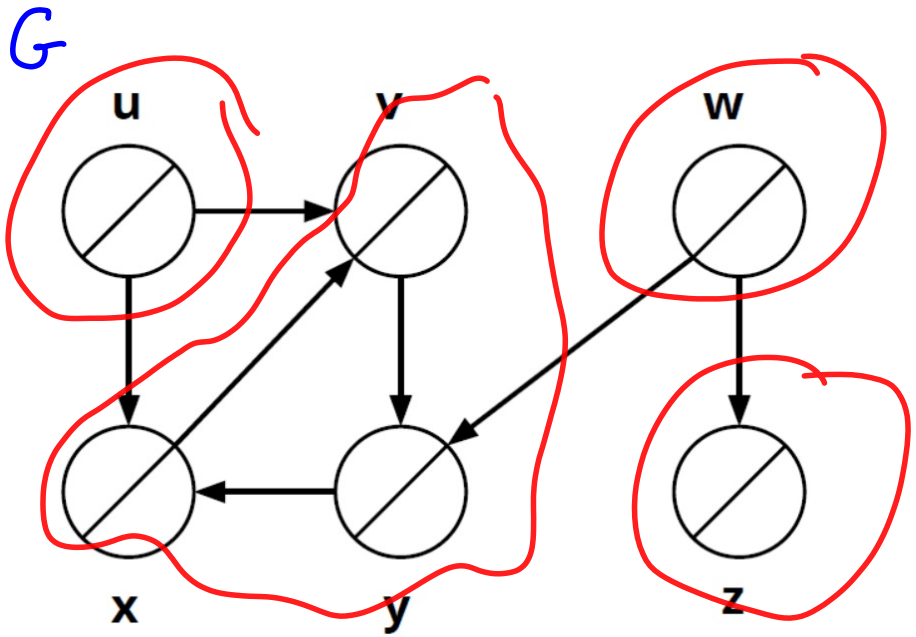
Finding Strongly Connected Components

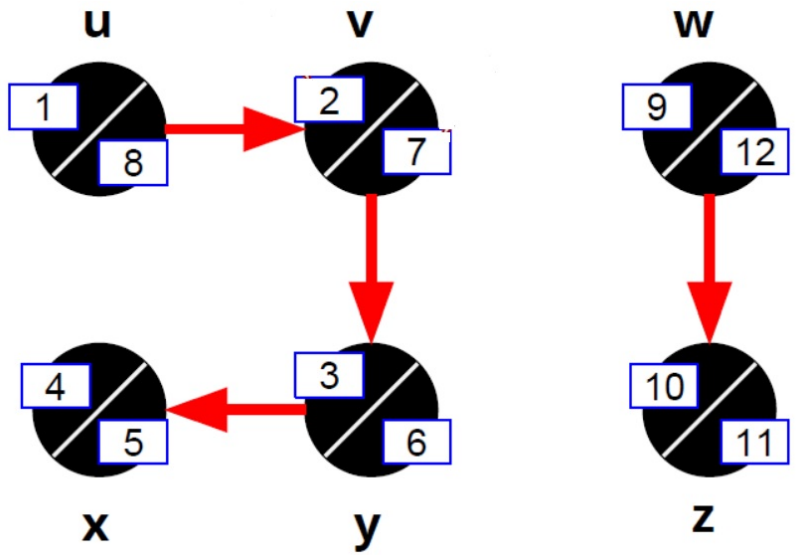
StronglyConnectedComponents(G)

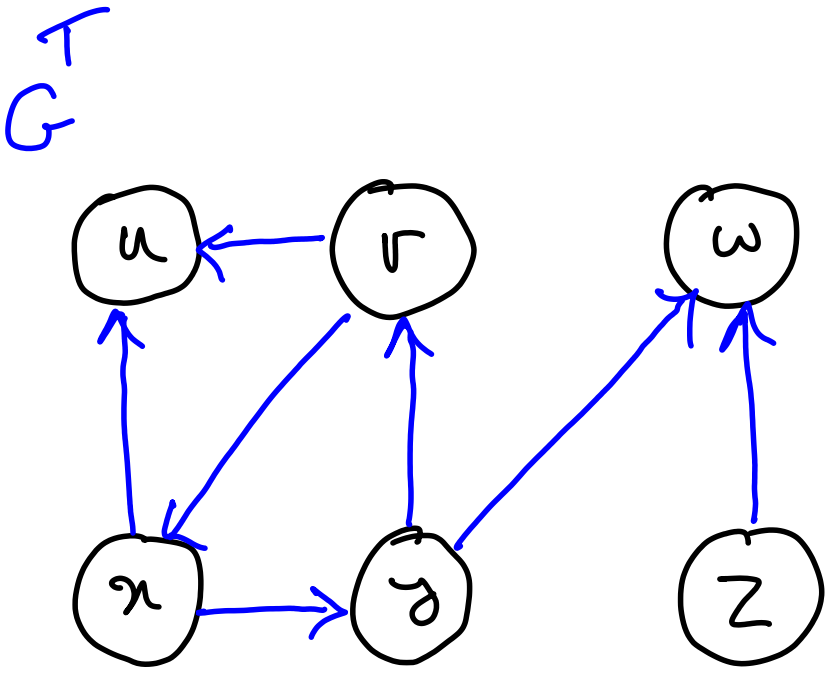
1. Call $DFS(G)$ to compute finishing times $u.f$ for all u.
2. Compute $G^{reverse}$
3. Call $DFS(G^{reverse})$, but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in the first DFS)
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.

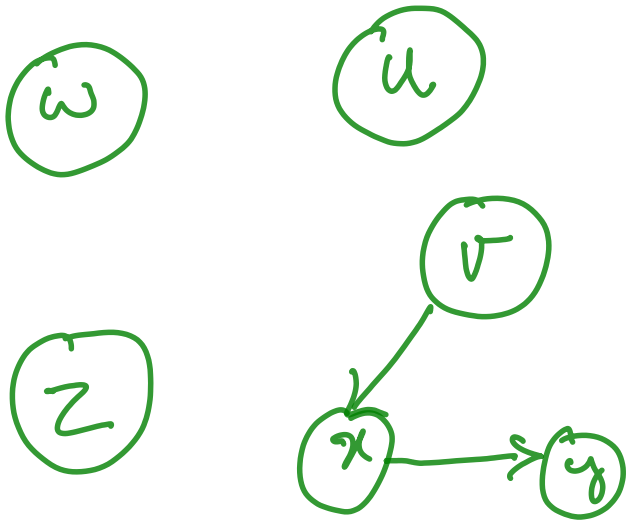



C
x
u
v
y
C'


G
u
v
w
x
y
z


u
v
w
1
8
2
7
9
12
4
5
3
6
10
11
x
y
z


G^T
u
v
w
x
y
z

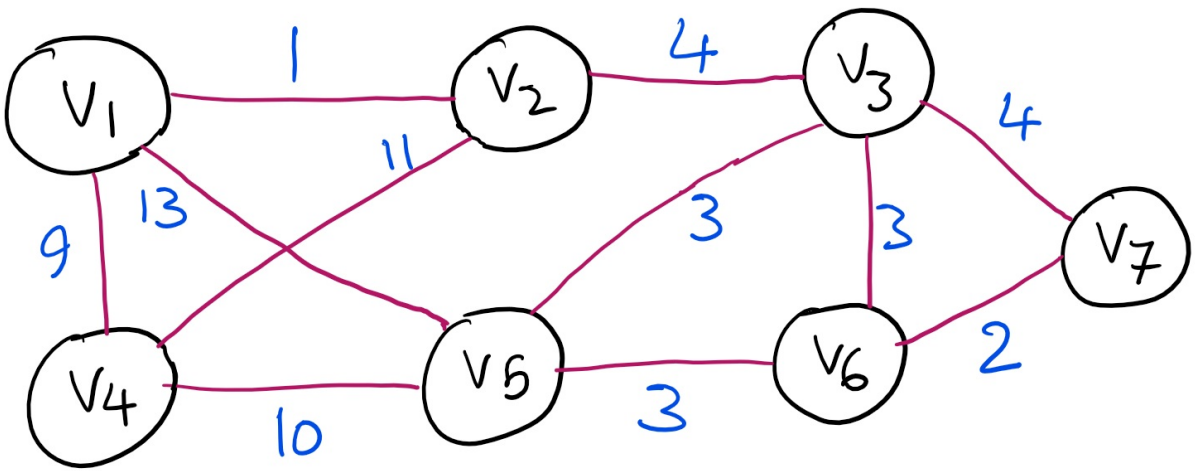

w
u
z
v
x
y

Review: Weighted Graphs

A weighted graph is a graph $G = \langle V, E \rangle$ for which each edge has an associated **weight**, given by a weight function $w : E \to \mathbb{R}$.

$w((v_2, v_3)) = 4$

$w((v_1, v_4)) = 9$



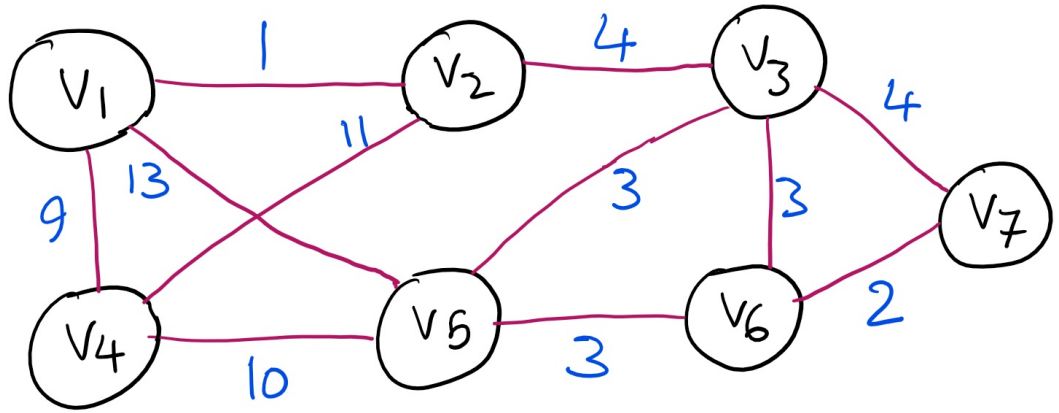
Minimum Spanning Trees (MSTs)

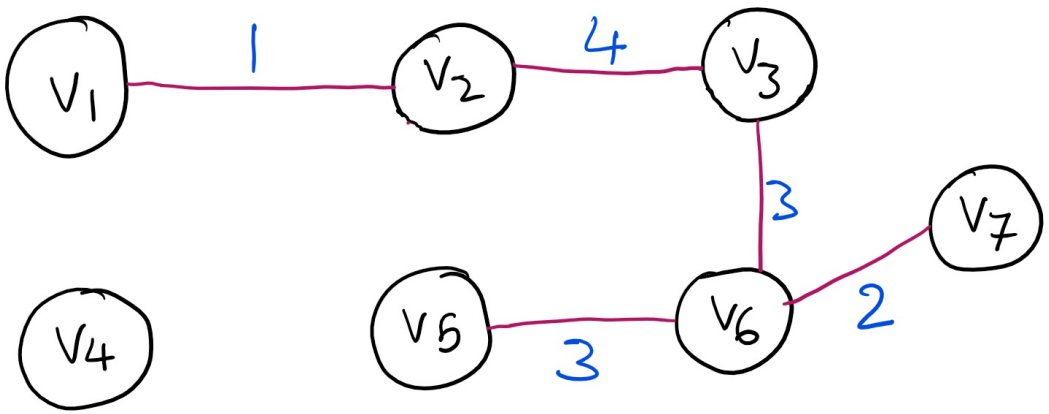
Let $G = \langle V, E \rangle$ be a connected undirected weighted graph.
A **minimum spanning tree** T for G is a sub-graph of G which

- includes all vertices of G; $\longrightarrow$ Spanning
- is acyclic and connected; $\longrightarrow$ Tree
- its total weight is minimized. $\longrightarrow$ Minimum
  (among all other sub-graph of G with the above two properties).

Note: A graph might have more than one MST.

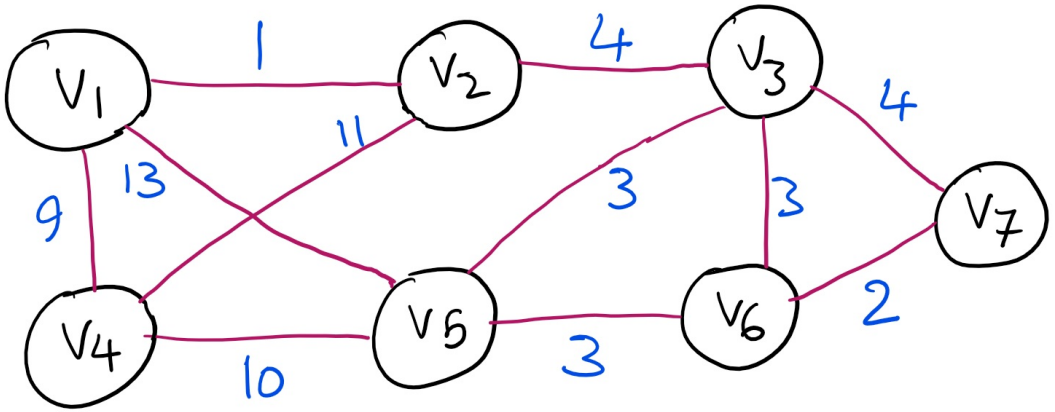
Minimum Spanning Trees (MSTs): Examples

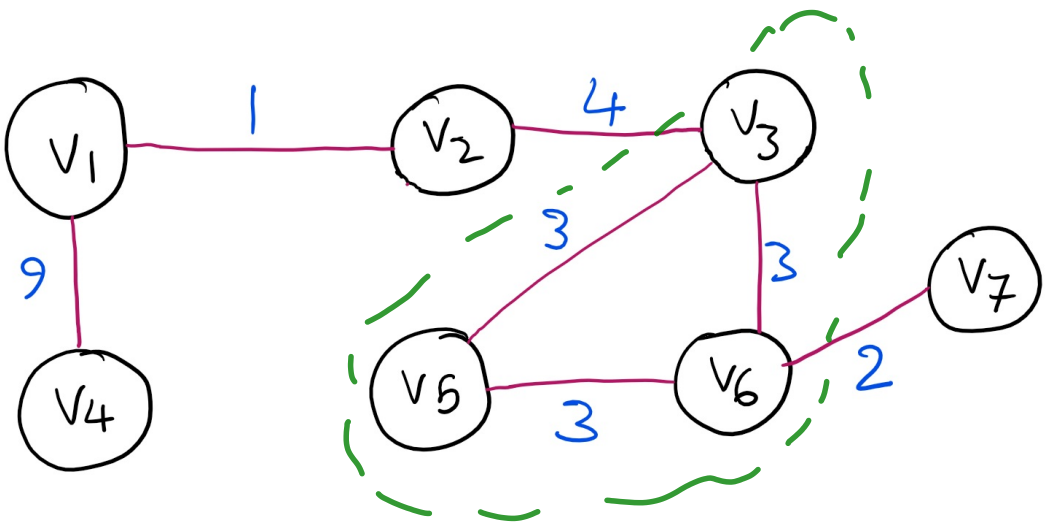




✗ disconnected

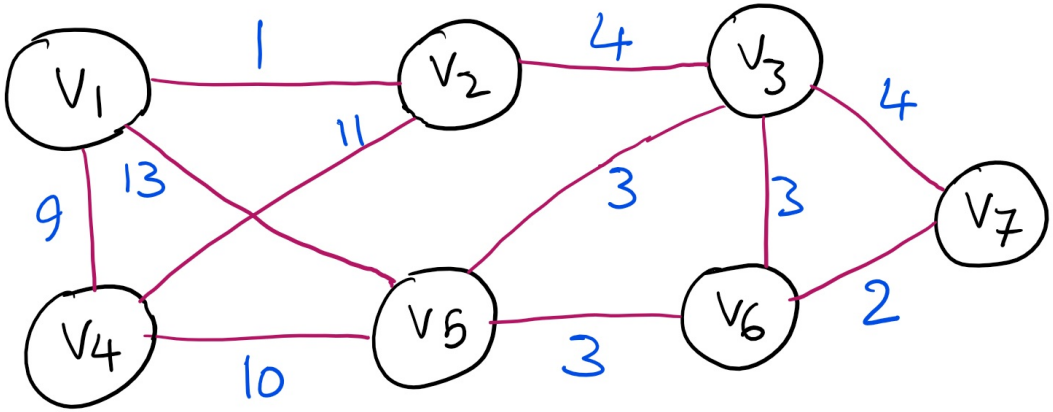
Minimum Spanning Trees (MSTs): Examples

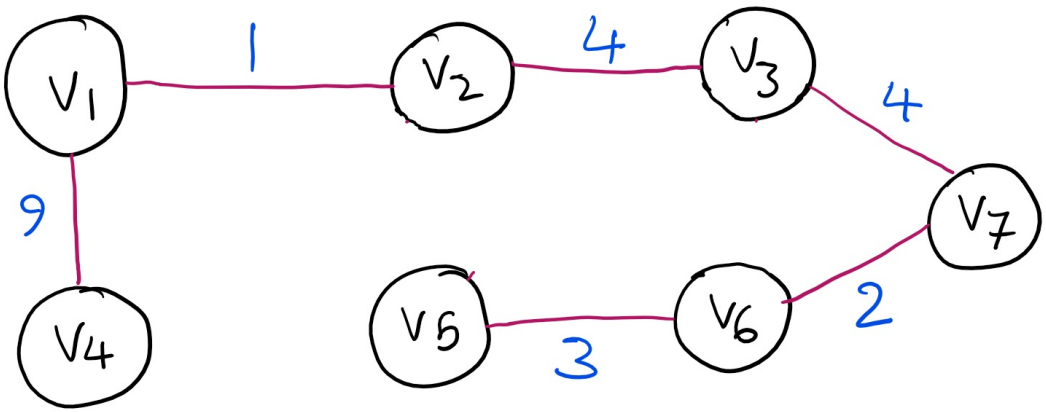




X includes a cycle

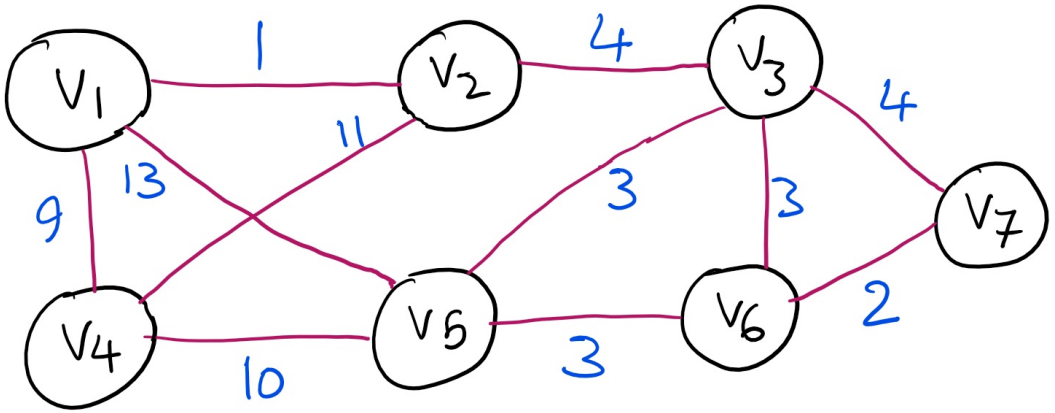
Minimum Spanning Trees (MSTs): Examples

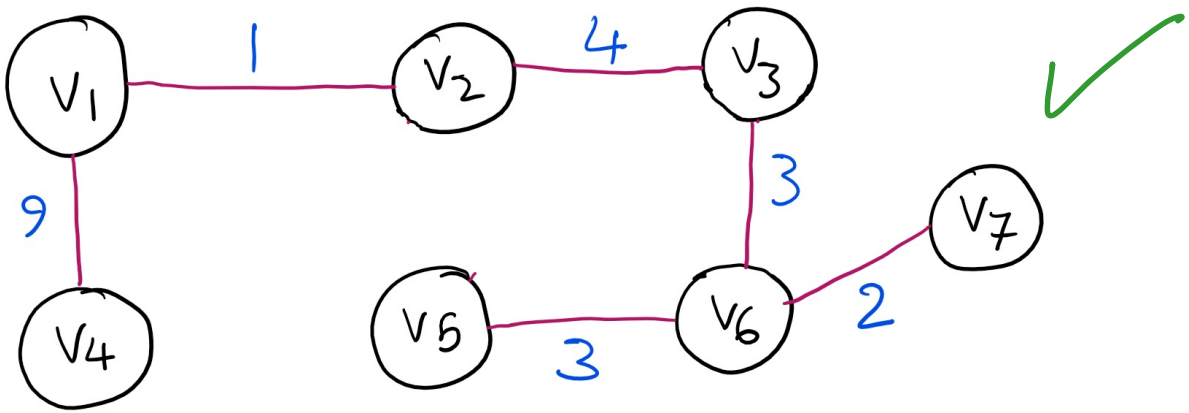




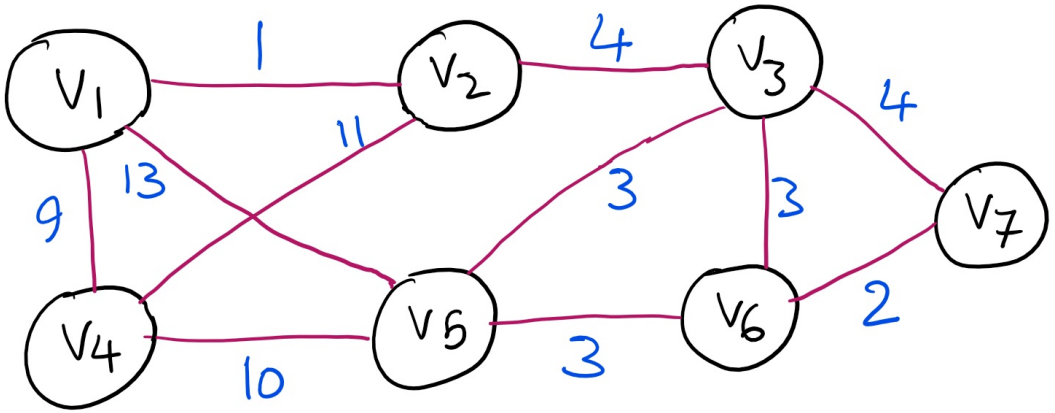
X total weight is not minimum

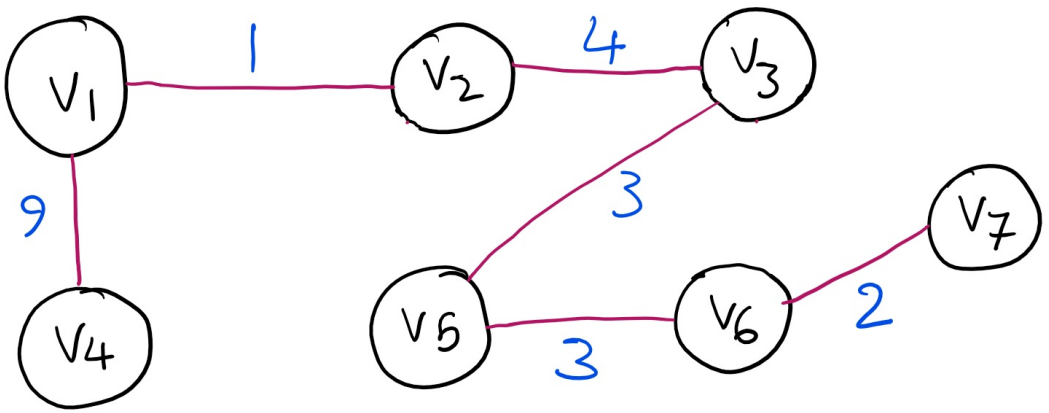
Minimum Spanning Trees (MSTs): Examples





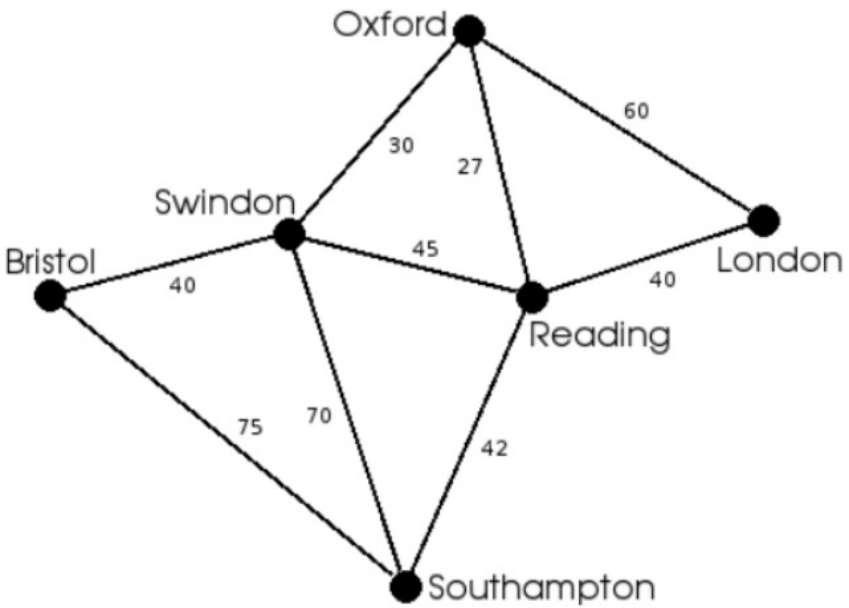
Minimum Spanning Trees (MSTs): Examples





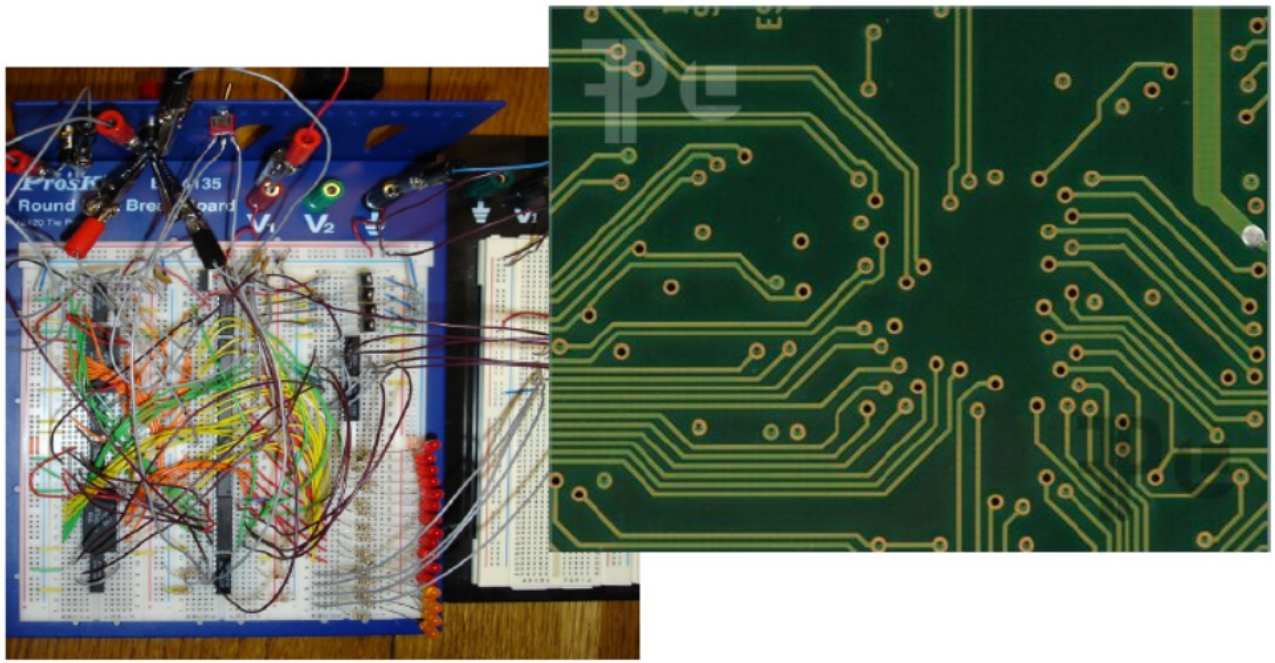
Finding Minimum Costs/Distances

- **Example 1**: Build a road network that connects all towns and with the minimum cost.



Minimum Spanning Trees: Some Applications

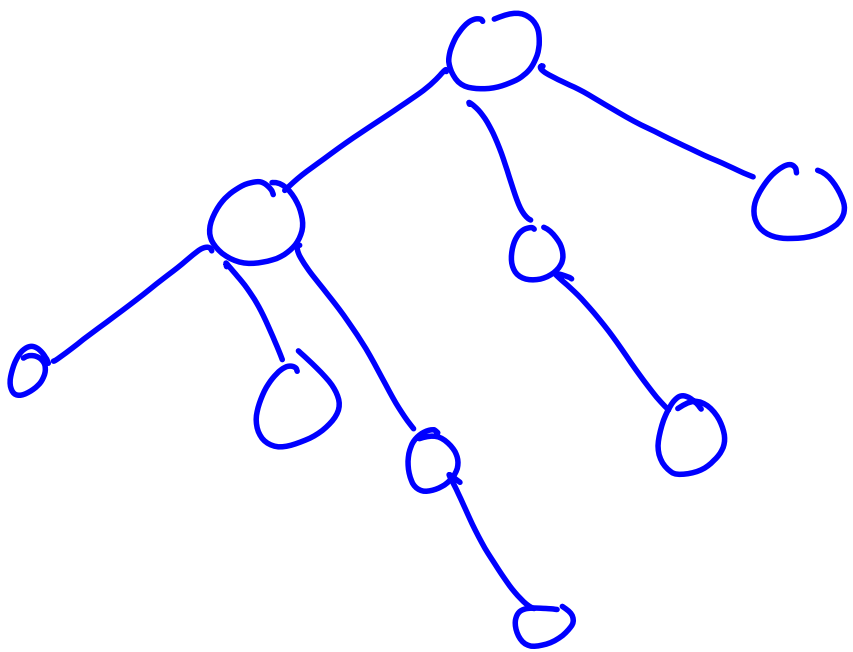
- **Example 2**: Connect all components of a circuit with the least amount of wiring.



Review: Properties of Trees

Let T be a tree with n vertices. Then

- T has **exactly** $n-1$ edges.
- Adding one edge to T will create a cycle
- Removing one edge from T will disconnect the graph



Properties of MSTs

- The MST of a connected undirected graph $G = \langle V, E \rangle$ has $|V|$ vertices.

- The MST of a connected undirected graph $G = \langle V, E \rangle$ has $|V| - 1$ edges.

Finding MSTs of a Graph: Ideas

Idea 1: Let $T.V = G.V$. Start with $T.E = G.E$; Keep **deleting** edges until an MST remains.

$|V| = n$

In the **worst-case**, how many **edges** are deleted?

$\frac{n(n-1)}{2} - (n-1) \in \Theta(n^2)$

An undirected graph G with n vertices has at most $\frac{n(n-1)}{2}$ edges.

Idea 2: Let $T.V = G.V$. Start with **empty** $T.E$; Keep **adding** edges until an MST is built.

In the **worst-case**, how many **edges** is added? $n-1 \in \Theta(n)$

MSTs: The Generic Growing Algorithm

GenericMST(G):

1. $T.V = G.V$ **# Initializing vertices of the MST**
2. $T.E = \varnothing$ **# Starting with an empty set of edges**
3. **While** T does not form a spanning tree:
4. find an edge $e \in G.E$ that is **safe** for T
5. $T.E = T.E \cup \{e\}$ **# Add the edge e to the MST**
6. **return** T

Challenge: What is a **safe** edge? and how to **find** one?

Growing MSTs: Finding Safe Edges

Intuition: While growing T, if we make sure that T is always a **subgraph** of some MST of G, then eventually T will become an MST.

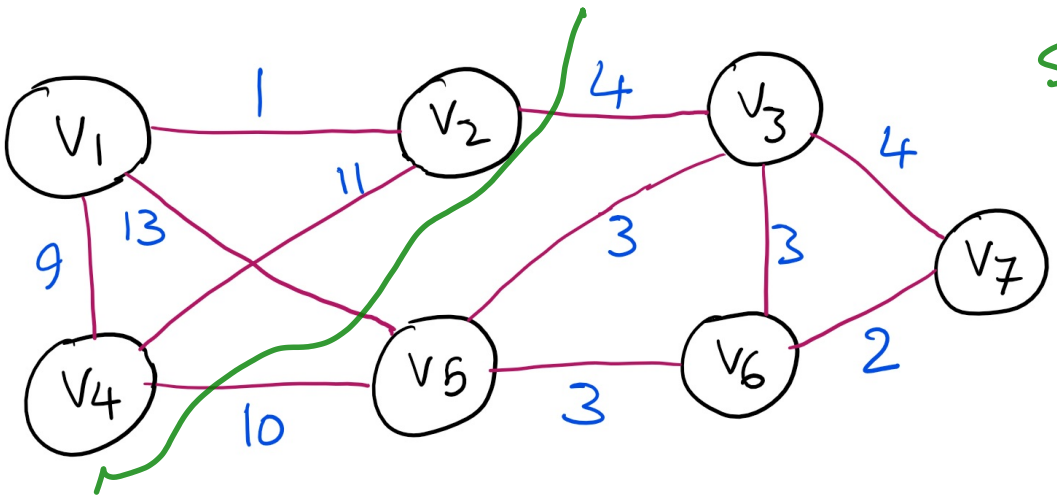
Analogy: If we make sure the pieces we put together is always a subset of the real picture while we grow it, then eventually it will become the real picture! (Picture borrow from Larry Zhange's Slides)

Edge e is **safe** for T iff $T.E \cup \{e\}$ is a subset of edges of some MST of G.

Growing MSTs: Finding Safe Edges

A **cut** $\langle S_1, S_2 \rangle$ of an undirected graph $G = \langle V, E \rangle$ is a partition of V into two **disjoint** subsets S_1 and S_2.
That is, $S_1 \cap S_2 = \varnothing$ and $S_1 \cup S_2 = V$.
We can also define S_2 in terms of V and S_1. That is, $S_2 = (V - S_1)$.
Example: If a graph G has exactly two **connected components**, then the two components form a cut of G.

An edge $(u, v) \in E$ **crosses** the **cut** $\langle S_1, S_2 \rangle$ if one of its endpoints is in S_1 and the other is in S_2.

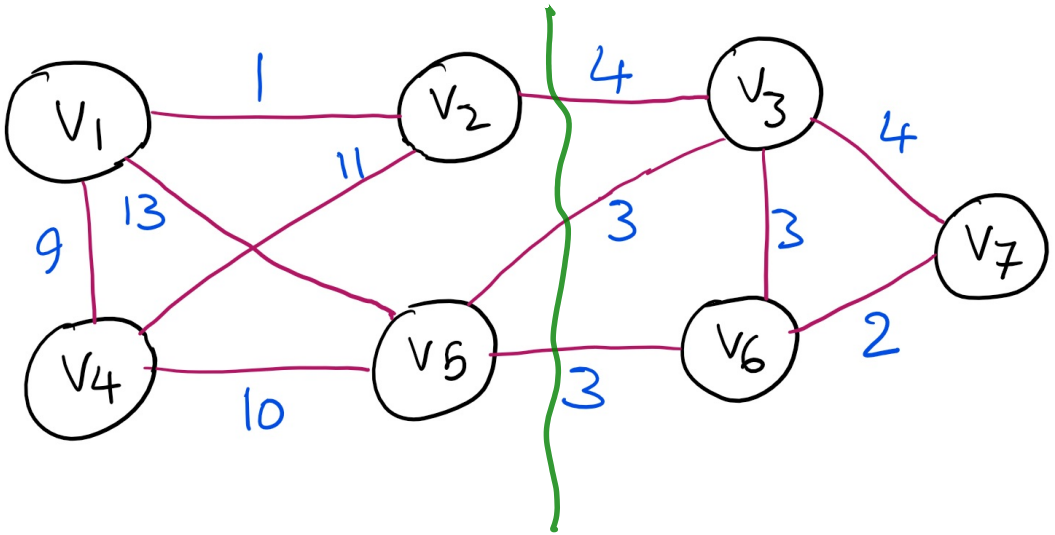


$S_1 = \{v_1, v_2, v_4\}$

$S_2 = \{v_3, v_5, v_6, v_7\}$

Crossing edge:

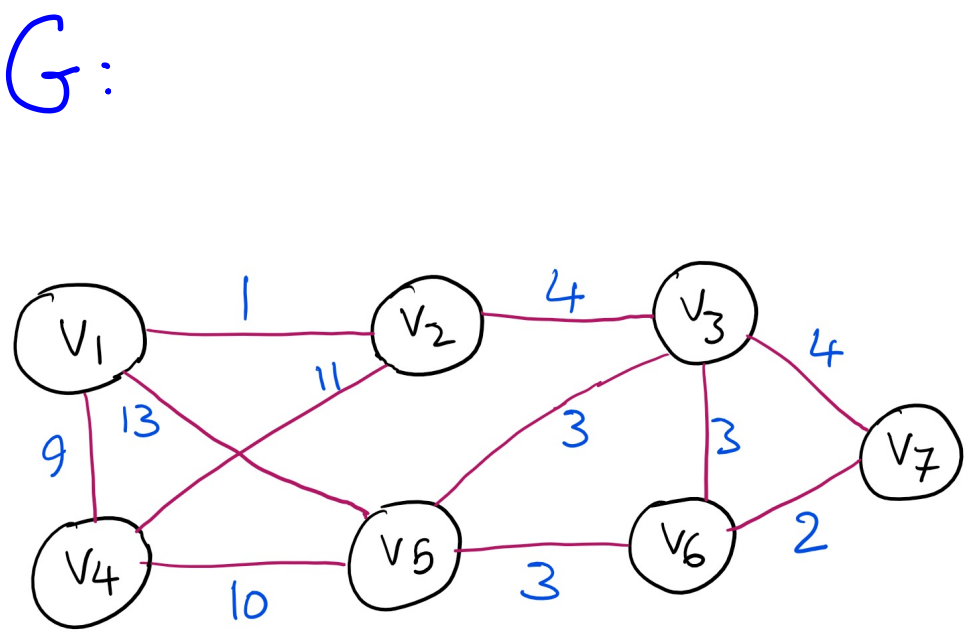
$(v_4, v_5), (v_2, v_3), (v_1, v_5)$

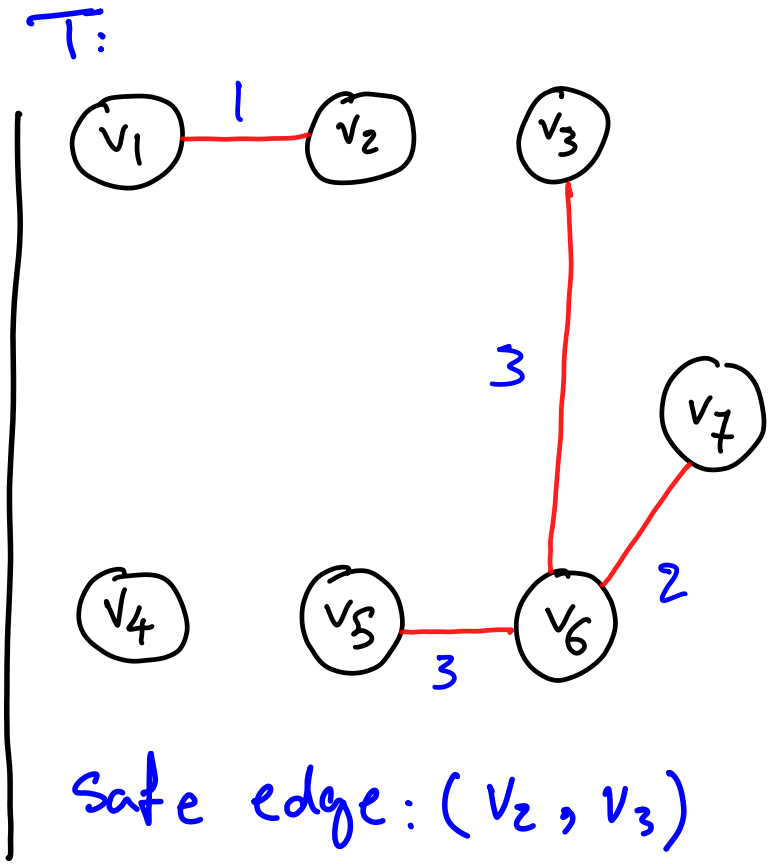

V1
V2
V3
V4
V5
V6
V7
1
4
4
11
13
9
3
3
2
3
10

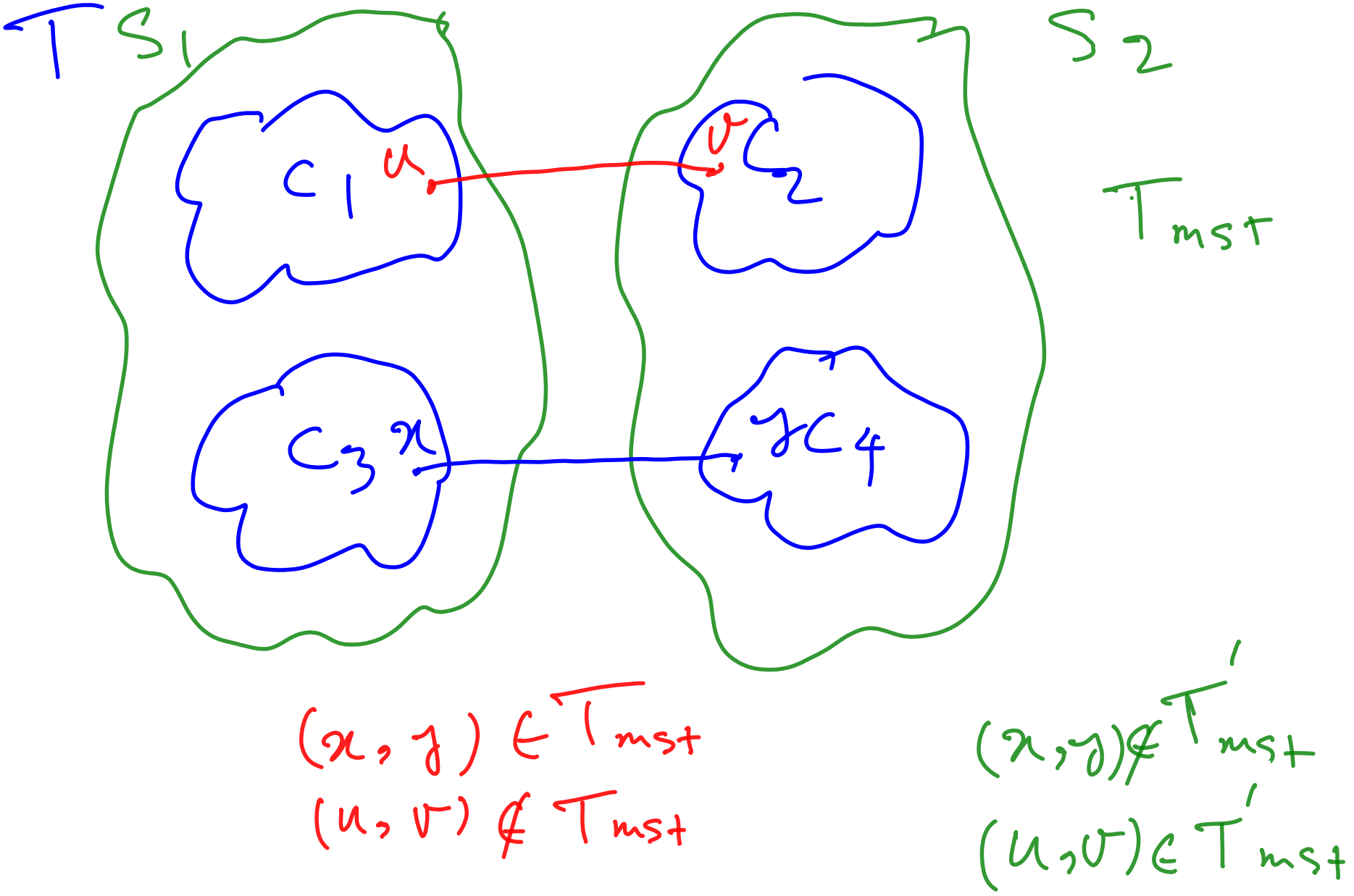
Growing MSTs: Finding Safe Edges

Theorem: Let G be a connected undirected weighted graph, and T be a subgraph of G which is a subset of some MST of G.
Let edge e be the **minimum weighted** edge among all edges that **cross** different **connected components** of T.
Then e is **safe** for T.

Proof: **Theorem 23.1** and **Corollary 23.2** in CLRS.





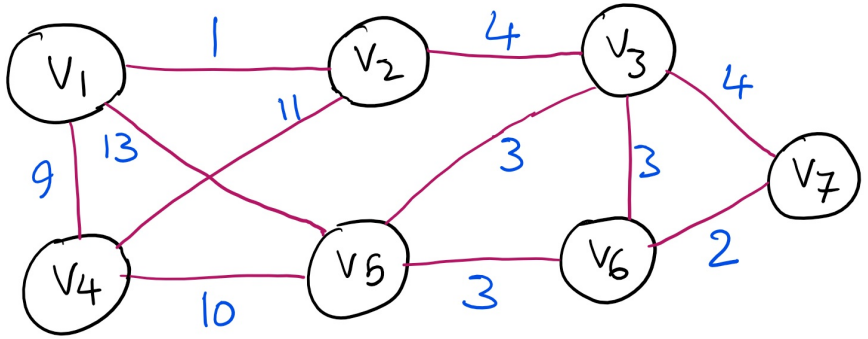

T
S_1
S_2
C_1
u
v
C_2
T_{mst}
C_3
x
y
C_4
$(x, y) \in T_{mst}$
$(u, v) \notin T_{mst}$
$(x, y) \notin T'_{mst}$
$(u, v) \in T'_{mst}$

The Generic MST Growing Algorithm

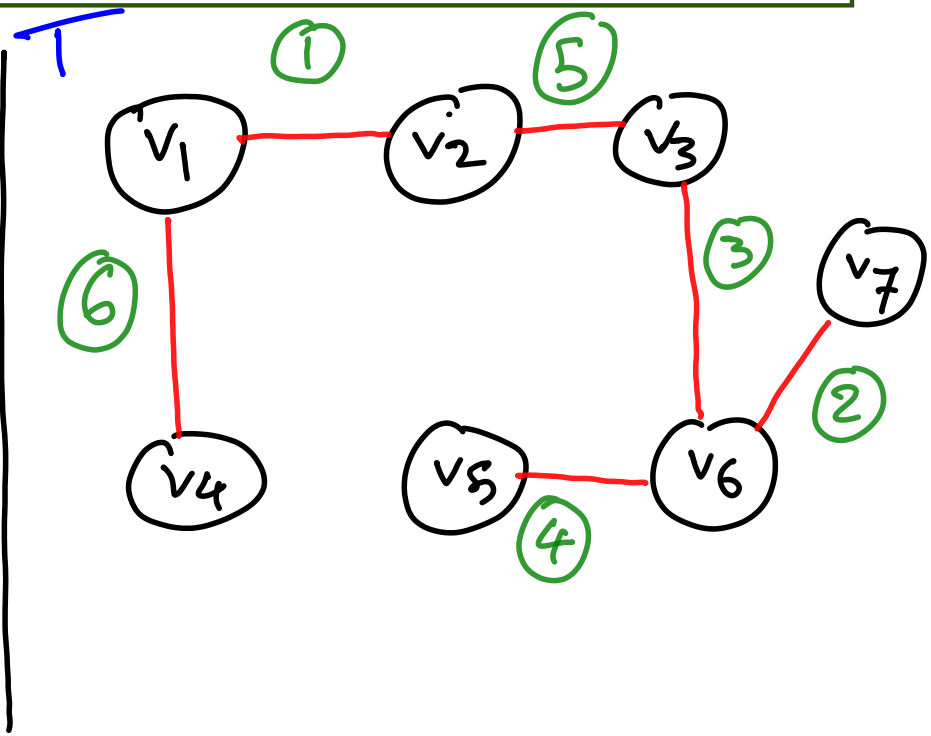
GenericMST(G) :

1. $T.V = G.V$ **# Initializing vertices of the MST**
2. $T.E = \varnothing$ **# Starting with an empty set of edges**
3. **While** T does not form a spanning tree:
4. find $e \in G.E$ with min weight that **crosses** two connected components of T
5. $T.E = T.E \cup \{e\}$ **# Add the edge e to the MST**
6. **return** T

G



T



Growing MSTs: Prim's and Kruskal's Algorithms

For the actual implementation of the growing algorithm:

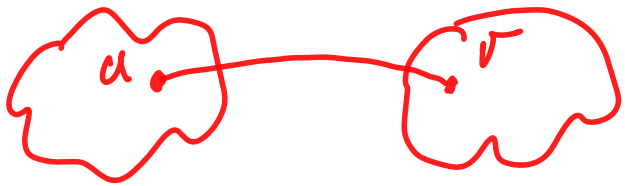
- How to keep track of the **connected components**?
- How to efficiently find the **minimum weighted** edge?

The (main) difference between Kruskal's algorithm and Prim's algorithm is that they use **different data structures** to do these two things.

Kruskal's MST Algorithm

High-Level Idea:
Go through the edges in order of non-decreasing **weight**;
Select and add each edge that **crosses** two **connected components** of T.
That is, if for the edge (u, v), u is in **one** connected component and v is in **another**.

How to keep track of connected components of T efficiently?



ConnectedComponents(G):
1. **for** each vertex $v \in G.V$:
2. MakeSet(v)
3. **for** each edge $(u, v) \in G.E$:
4. **if** FindSet$(u) \neq$ FindSet(v): **# If u and v are in different connected components**
5. Union(u, v)

Kruskal's MST Algorithm

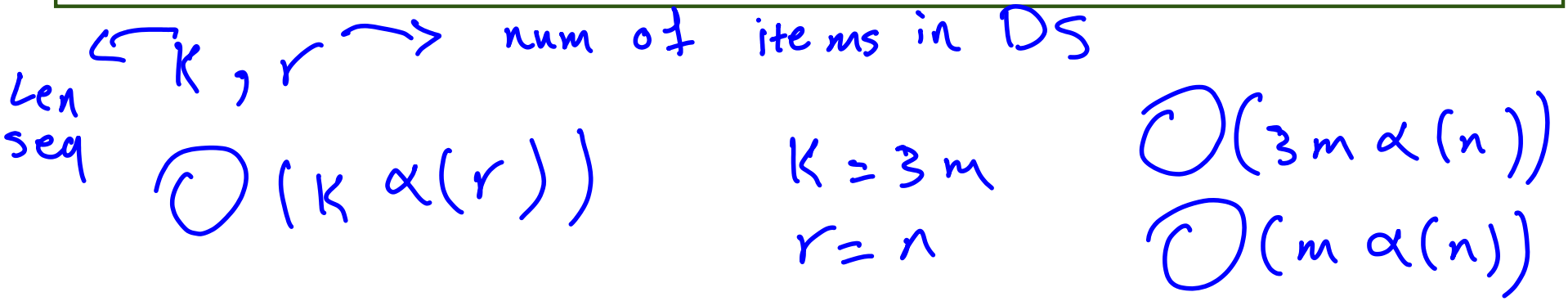
High-Level Idea:
Go through the edges in order of non-decreasing **weight**;
Select and add each edge that **crosses** two **connected components** of T.
That is, if for the edge (u, v), u is in **one** connected component and v is in **another**.

Implementation Idea:

1. **Sort** the edges according to their weights.
2. Go through the sorted list from **lightest** to **heaviest**:
   - Add (u, v) to T if FindSet$(u) \neq$ FindSet(v)
   - **After** adding (u, v) to T, u and v belong to the **same** connected component of T; Hence Union(u, v)

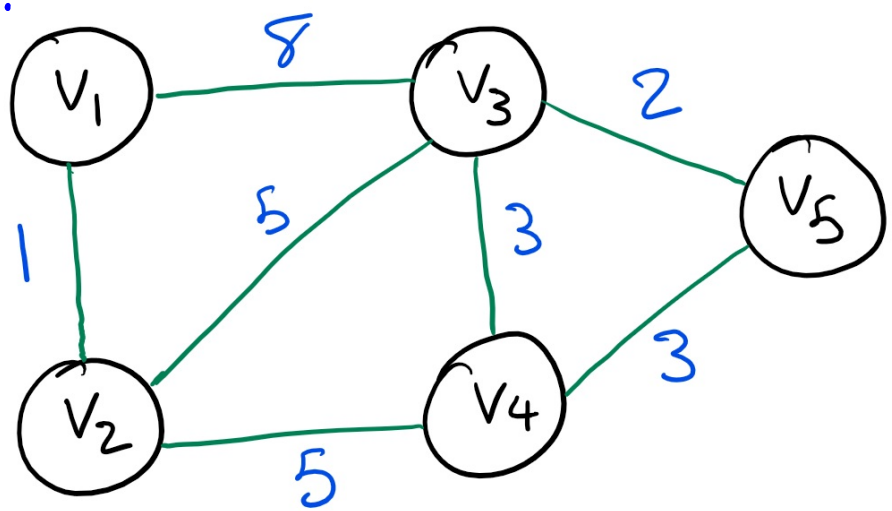
Pre-condition: G is an undirected graph.

KruskalMST(G):
1. $T.V = G.V$
2. $T.E = \varnothing$
3. **for** each vertex $v \in G.V$: **# Create a separate connected component for each vertex**
4. MakeSet(v)
5. Let $e_1, e_2, ..., e_m$ be the edges in $G.E$ sorted by **weight** in non-decreasing order
6. **for** $i = 1$ to m:
7. Let $(u, v) = e_i$
8. **if** FindSet(u) $\neq$ FindSet(v): **# Check u and v are in different components of T**
9. $T.E = T.E \cup \{(v, u)\}$ **# Add the safe edge (v, u) to T**
10. Union(u, v) **# Now u and v are in the same connected component of T**
11. **return** T



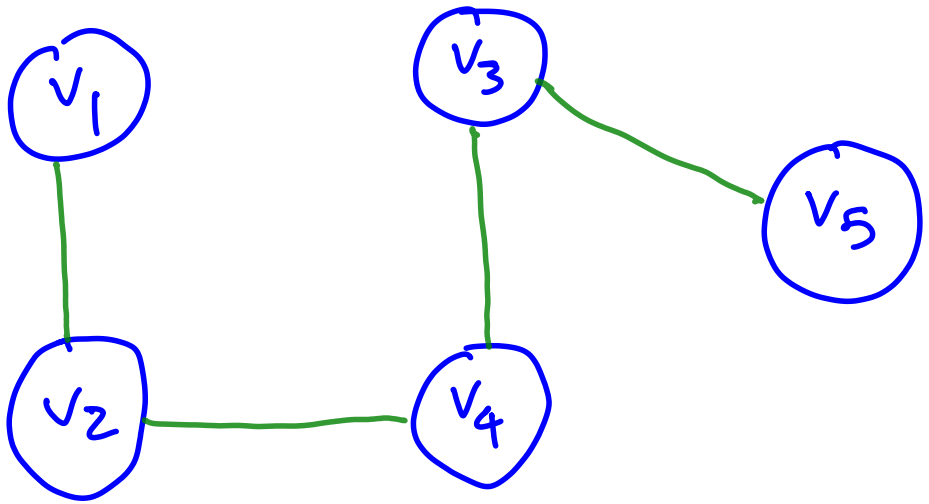
Edges sorted by weight: $(v_1, v_2), (v_3, v_5), (v_3, v_4), (v_4, v_5), (v_2, v_4), (v_2, v_3), (v_1, v_3)$

G:



Iteration	Edge
1	(v_1, v_2) ✓
2	(v_3, v_5) ✓
3	(v_3, v_4) ✓
4	(v_4, v_5)
5	(v_2, v_4) ✓
6	(v_2, v_3)
7	(v_1, v_3)

T:

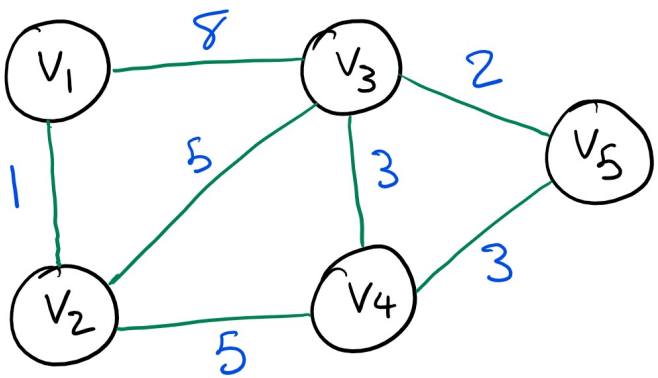


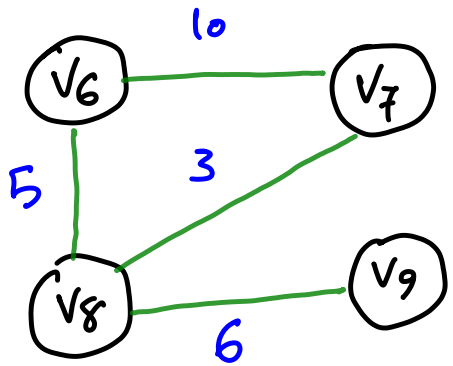
$\{v_1\}$ $\{v_2\}$ $\{v_3\}$ $\{v_4\}$ $\{v_5\}$

$\{v_1, v_2, v_3, v_5, v_4\}$

What if the input graph G is **not connected**?

Kruskal's Algorithm will generate a Minimum Spanning Forest





MST T_1

MST T_2

Kruskal's MST Algorithm: Worst-case Run Time

Assuming that Disjoint-Sets implemented by tree with **union-by-rank** and **path-compression**.

$|V| = n \quad |E| = m$

1. Sorting edges: $O(m \lg m)$
2. ***for*** loop on Lines #3 and #4 (MakeSet): $O(n)$
3. ***for*** loop on Lines #6 to #10 (FindSet and Union): $O(m\alpha(n)) \simeq O(m)$

Total running time: $O(n + m \lg m)$

Recall that $m \leq \frac{n(n-1)}{2}$. $m \in O(n^2) \Rightarrow \lg m \in O(\lg n^2)$

$\lg n^2 = 2 \lg n \in O(\lg n)$ $\Rightarrow$ $\lg m \in O(\lg n)$

So total run time can be **rewrite** as:

$O(n + m \lg n)$

If the input graph G is **connect**, the run time is: $O(m \lg n)$

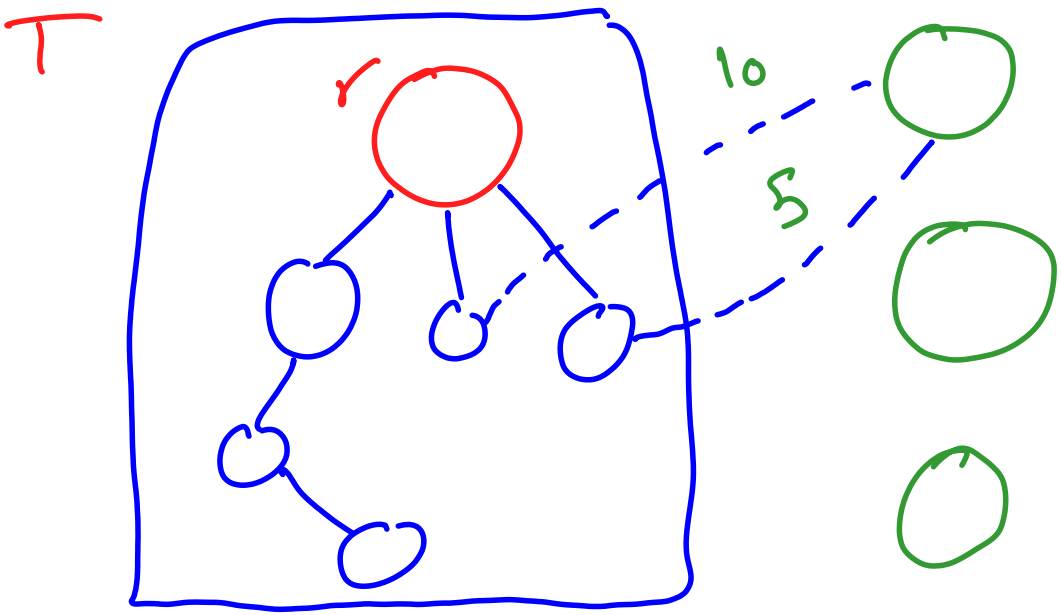
$m \geq n - 1 \Rightarrow n \in O(m)$

Prim's MST Algorithm

Pick an **arbitrary** vertex r of G as the starting vertex of T.
Grow T by adding one edge at a time:

- At every step T consists of one **connected component** that contains r, and **singleton** components that are **not** connected with r (each one is called an **isolated** vertex).

- At each step pick a **crossing** edge (between the component containing r and one of the isolated vertices) with the **minimum weight**.



Prim's MST Algorithm

Pick an **arbitrary** vertex r of G as the starting vertex of T.
Grow T by adding one edge at a time:

- At every step T consists of one **connected component** that contains r, and **singleton** components that are **not** connected with r (each one is called an **isolated** vertex).

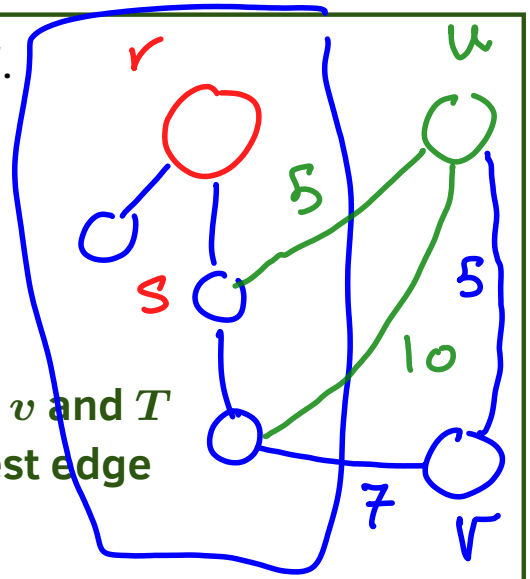
- At each step pick a **crossing** edge (between the component containing r and one of the isolated vertices) with the **minimum weight**.

How to find the minimum crossing edge(s) efficiently?
Store all **isolated** vertices in a **min-heap**;
Keys of the min-heap are **weights** of the **crossing** edges (between isolated vertices and T).

MinHeap Operations:

- BuildMinHeap(A): Given a list of elements A, return a **min-heap** that includes all elements in A. $O(n)$
- ExtractMin(H): Remove and return the item from the heap H with the **lowest** key. $O(\lg n)$
- DecreaseKey(H, i, k): **Decrease** the key value of the element x, stored in position i, to the new value k. (k assumed to be less than the current key of x). $O(\lg n)$

Precondition: G is a **connected** undirected graph. r is a vertex of G.

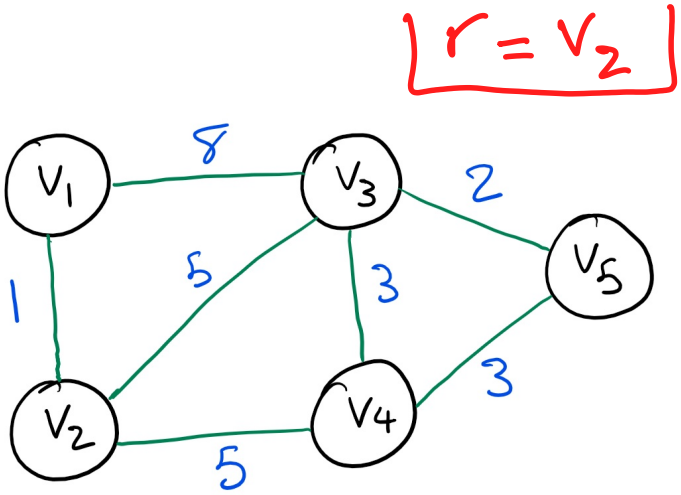


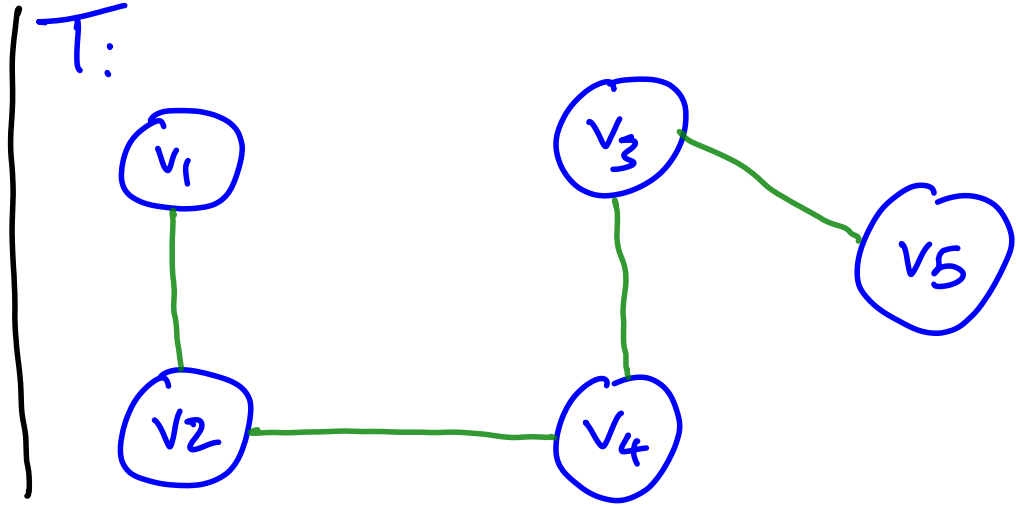
PrimMST(G, r):
1. $T.V = G.V$
2. $T.E = \varnothing$
3. **for** each vertex $v \in G.V$:
4. $\quad v.key = \infty$ **# $v.key$ keeps the shortest distance between v and T**
5. $\quad v.p =$ nil **# $v.p$ keeps who in T is v connected to via lightest edge**
6. $r.key = 0$ **# Set r to be the root of min-heap**
7. $H =$ BiuldMinHeapGraph(G)
8. **While** H not empty:
9. $\quad u =$ ExtractMin(H)
10. $\quad$ **if** $u.p \neq$ nil: **# $u.p$ is nil only when $u = r$**
11. $\quad\quad T.E = T.E \cup \{(u.p, u)\}$ **# Add a safe edge to the MST**
12. $\quad$ **for** each $v \in G.adj[u]$: **# all u's neighbours' distances to T need update**
13. $\quad\quad$ **if** $v \in H$ **and** $w((u, v)) < v.key$:
14. $\quad\quad\quad v.p = u$
15. $\quad\quad\quad$ DecreaseKey$(H, v.pos, w((u, v)))$
16. **return** T

u.p = s
u.key = 5

For each vertex v, access to v's position in H can be achieved in **constant time** by maintaining an **additional field $v.pos$**.

$v.pos$ is -1 when v is **extracted** from H. This also makes test $v \in H$ efficient.





It	$v_1.key/v_1.p$	$v_2.key/v_2.p$	$v_3.key/v_3.p$	$v_4.key/v_4.p$	$v_5.key/v_5.p$	u	H root
0	∞/nil	0/nil	∞/nil	∞/nil	∞/nil	-	v_2
1	1/v_2		5/v_2	5/v_2	∞/nil	v_2	v_1
2			5/v_2	5/v_2	∞/nil	v_1	v_4
3			3/v_4		3/v_4	v_4	v_3
4					2/v_3	v_3	v_5
5						v_5	empty

Prim's MST Algorithm: Worst-case Run Time

Assuming using binary min-heap and adjacency lists.

$n = |V|$ $m = |E|$

1. *for* loop on Lines #3 to #5: $O(n)$
2. Building Min-Heap: $O(n)$
3. Heap Operations: $O(\lg n)$
4. Total run time for all ExtractMin calls: $O(n \lg n)$
5. Total run time for all DecreaseKey calls: $O(m \lg n)$

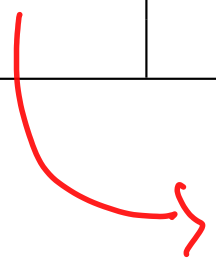
Total running time: $O((n+m)\lg n)$

Since the input graph G is **connected** the run time is: $O(m \lg n)$

$m \geq n-1 \Rightarrow n \in O(m)$

Overview: Prim's and Kruskal's Algorithms

	Input Graph G	**Keep track of Connected Components**	**Find Minimum Weight Edge**
Kruskal's	If G is not connected generates a minimum spanning forest	Disjoint-Sets ADT	Sort all edges according to weight
Prim's	G has to be connected	Keep One Tree plus Isolated Vertices	Use Priority Queue ADT



Generates a Minimum Spanning Forest for disconnected graphs under certain conditions

After Lecture

- After-lecture Readings: Section on "Minimum spanning trees" in the course notes, CLRS Chapter 23

- Problems 23.1-1, 23.2-2 in CLRS.