Graphs

This week: Depth First Search and Topological Sort

Announcements

Test 2 this Friday March 18

- 50 minutes
- No aids
- During your Tutorial Time in Assigned Room

Tutorial Section and Time	Family Names starting with	Room
TUT 101 10 am	A - G	RW 117
TUT 101 10 am	H - L	WI 1017
TUT 101 10 am	M - T	SF 2202
TUT 101 10 am	U - Z	BL 205
TUT 201 11 am	A - J	EX 300
TUT 201 11 am	K - R	EX 310
TUT 201 11 am	S - Z	EX 320

Depth First Search

- Basic idea is to search through the graph going as deep as possible before we backtrack to explore the edges of already-discovered vertices
- Similar to BFS, each vertex is coloured as we go
  - White not yet discovered
  - Grey discovered but not fully explored
  - Black fully explored
- Instead of storing distance from source vertex, store timestamps
  - d[v] discovery time
  - f[v] finish time.

Depth First Search

- Natural to write DFS recursively

- Because DFS is commonly used to find connected components
our main algorithm won't take a source vertex but will call
DFS-Visit(G, s) repeatedly on each s in V

adjacency-list order
(outer list order)

Depth First Search

```
DFS(G=(V,E)):
1    for each v in V:
2        colour[v] ← white
3        f[v] ← d[v] ← ∞
4        π[v] ← NIL
5    time ← 0    #global
6    for each v in V:
7        if colour[v] == white:
8            DFS-VISIT(G,v)
```

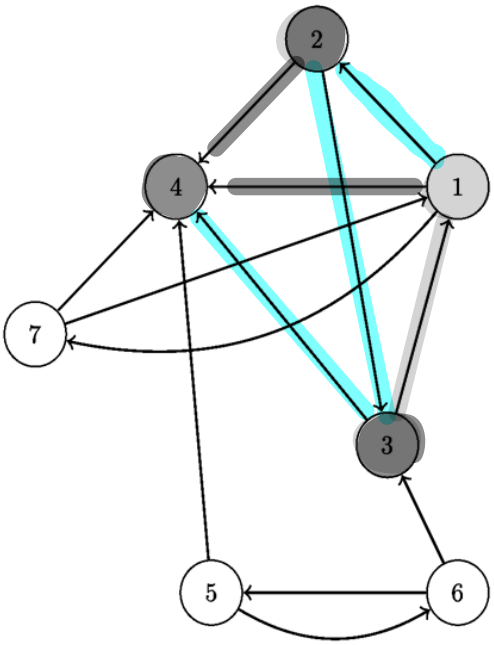
initialise

```
DFS-VISIT(G=(V,E),u):
1    colour[u] ← grey
2    d[u] ← time ← time + 1
3    for each (u,v) in E:
4        if colour[v] == white:
5            π[v] ← u
6            DFS-VISIT(G,v)
7    colour[u] ← black
8    f[u] ← time ← time + 1
```

black edges → when we encounter black vertex
→ • forward edges: when you go to a descendant
• cross edges: from one tree in a forest to another

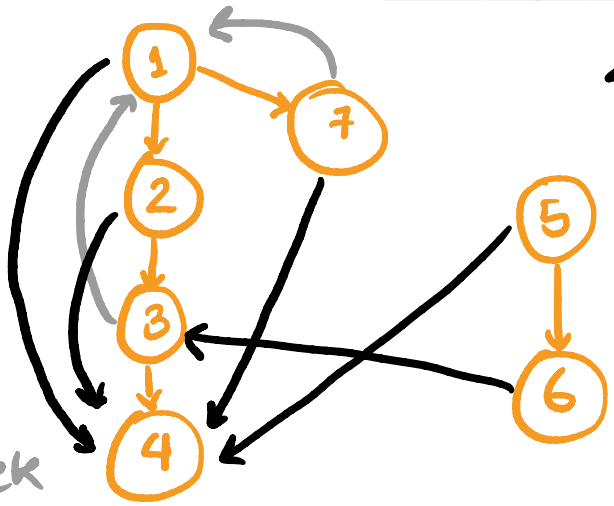
time : 0 1 2 3 4 5 6 7 8 9 10

Chalk Talk



→ 1: [2, 4, 7]
→ 2: [3, 4]
→ 3: [1, 4]
4: []
5: [4, 6]
6: [3, 5]
7: [1, 4]

v	c	π	d	f
1		NIL	1	10
2		1	2	7
3		2	3	6
4		3	4	5
5		NIL	11	14
6		5	12	13
7		1	8	9



2→4, 1→4 } forward edge

7→4, 6→3 } cross

→ back edges cuz they go back to an ancestor (they form an ancestor!)
→ grey edges when we visit a node that's already grey

If you need to find if graph is cyclic, as soon as you explore a grey edge!

DFS: Complexity

DFS-VISIT only called on white vertices and then those vertices immediately they are painted grey
So DFS-VISIT called once on each vertex.

Outside of recursive calls, each execution of DFS-VISIT
Examines the adjacency list for one vertex

So total running time is $O(|V| + |E|)$

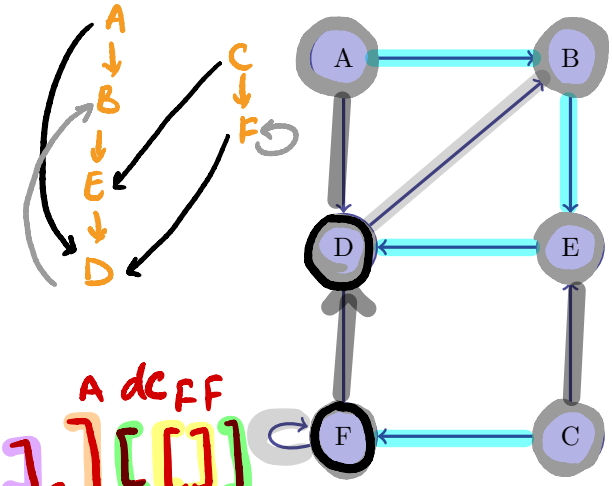
$|E|$: size of adjacency list

→ DFS gets called on every vertex at least once

DFS: forest

- tree (u,v) edges in the DFS-forest
  v is white when edge (u,v) is examined
- back/grey (u,v) v is ancestor of u in DFS-forest
  v is grey when edge (u,v) is examined
- forward (u,v) v is descendant of u in the DFS-forest
  v is black when edge (u,v) is examined
- cross (u,v) all other edges not part of DFS-forest, v is neither ancestor nor descendant
  v is black when edge (u,v) is examined

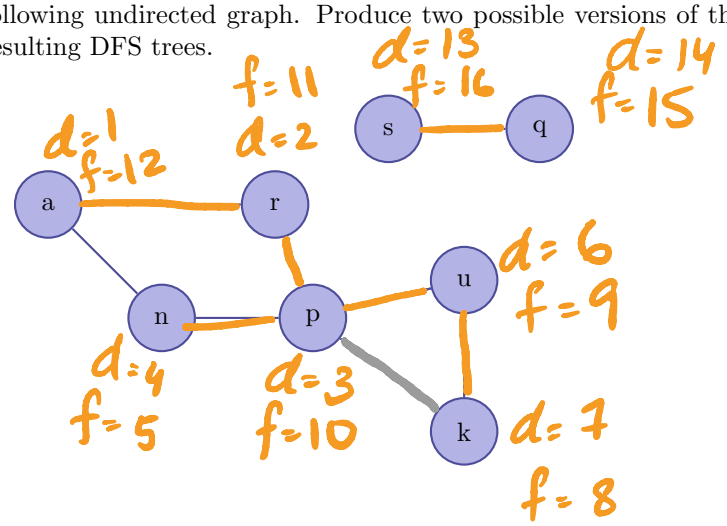
1. Run DFS(G) on the directed graph G shown here. Assume that adjacency list and sub lists are all ordered in increasing alphabetical order. Annotate on each vertex its parent as well as discovery and finish times. And draw the resulting DFS trees.



	d	π	f
A	1	NIL	8
B	2	A	7
C	9	NIL	12
D	4	E	5
E	3	B	6
F	10	C	11

dA dB [[[dE [dD] D] E] B] A dC F F [[ ]] C

2. Now run DFS(G) on the following undirected graph. Produce two possible versions of the adjacenty lists and note the difference in the resulting DFS trees.



a: r, n
r: p, a
n: a, p
p: r, n, u, k
u: p, k
k: u, p
s: q
q: s

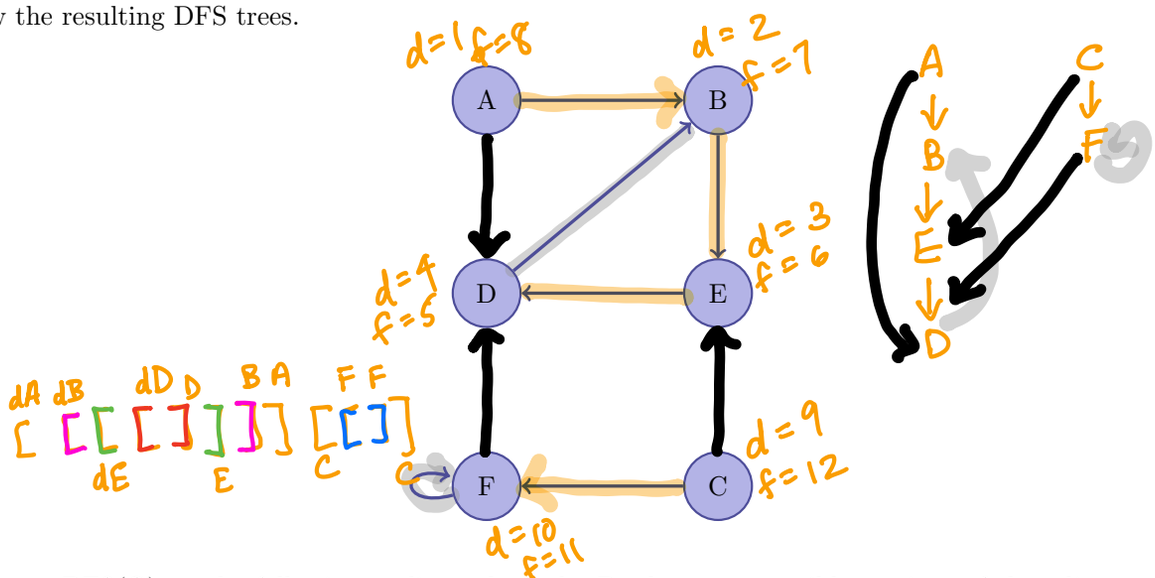
No black edges! (black edges already visited)

That edge can be travelled either way, proof by contradiction

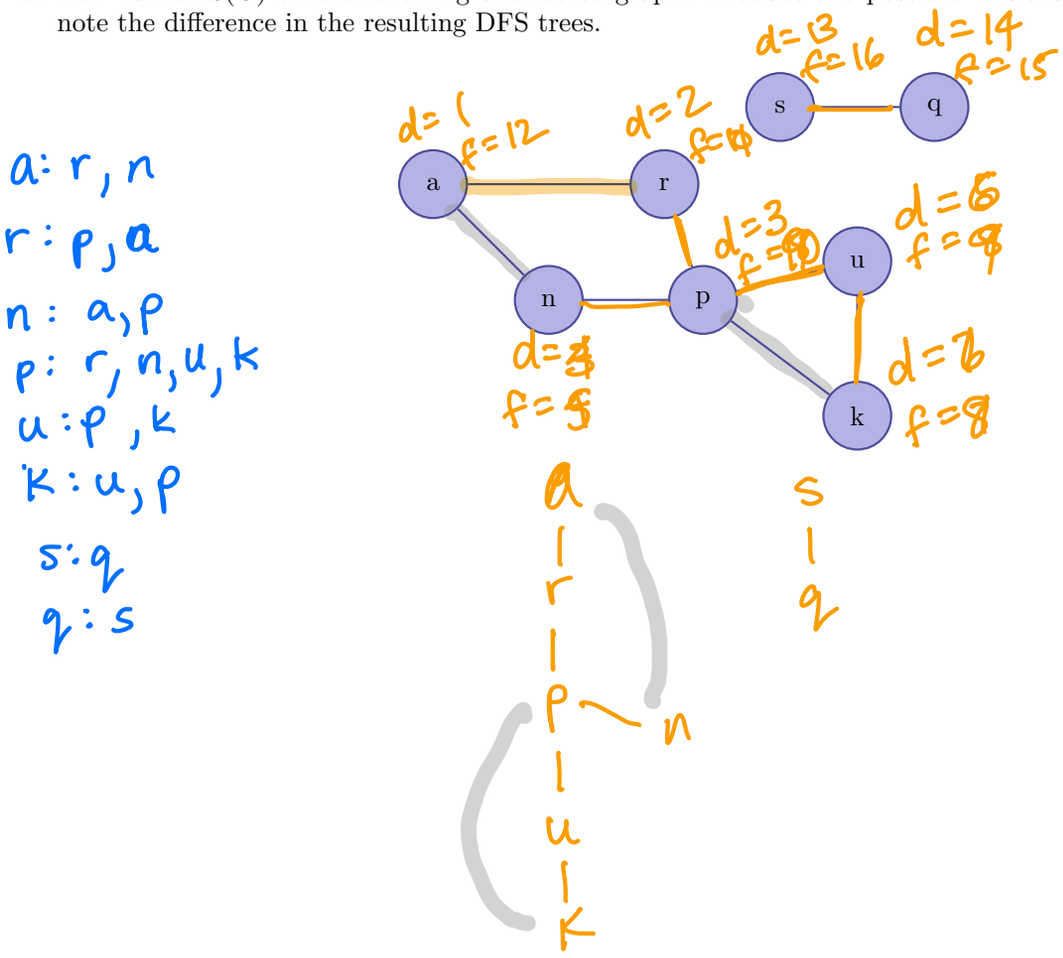
No forward edges or cross edges!

Only tree & back edges

1. Run DFS(G) on the directed graph G shown here. Assume that adjacency list and sub lists are all ordered in increasing alphabetical order. Annotate on each vertex its parent as well as discovery and finish times. And draw the resulting DFS trees.



2. Now run DFS(G) on the following undirected graph. Produce two possible versions of the adjacenty lists and note the difference in the resulting DFS trees.



→ No. of back edges on undirected connected graph by DFS = $|E| - [|V|-1]$

$= |E| - |V| + 1$

Eg: 20 edges. 9 vertices, back edges = 12

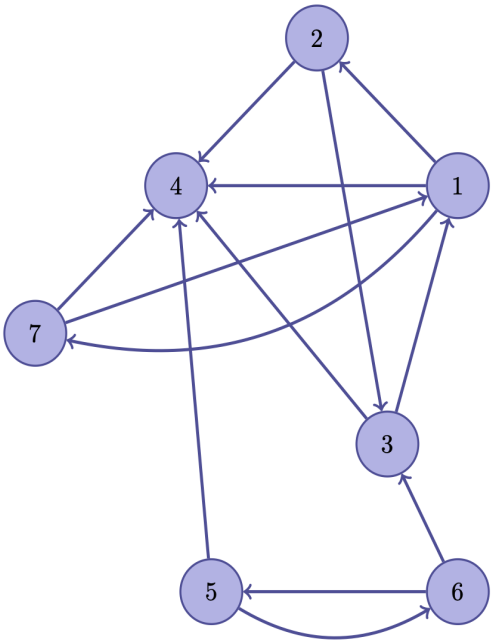
Graphs Continued

This week: Topological Sort, Spanning Trees, Starting Disjoint Sets

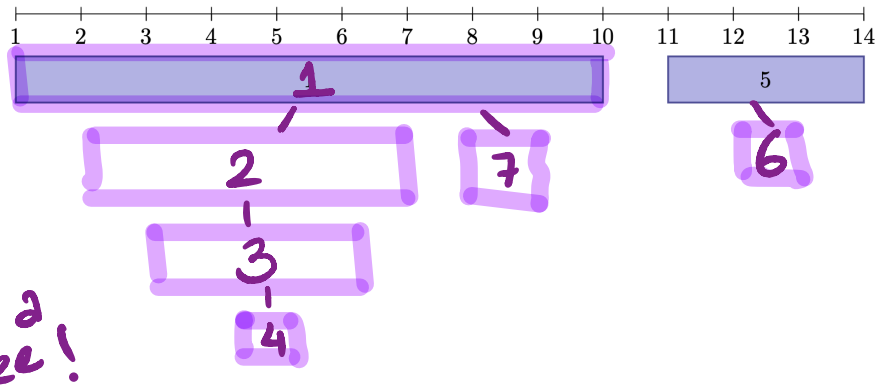
Announcements

- Test 2 grading is on-going
- PS3 due Friday April 1st (4 pm)
- PS3 will have 1-3 more questions

Chalk Talk



v	d	f
1	1	10
2	2	7
3	3	6
4	4	5
5	11	14
6	12	13
7	8	9



It's a tree!

W10 Worksheet Q1, Q4

Start with Q1 then go on to Q4. If you have time, go back to do Q2.

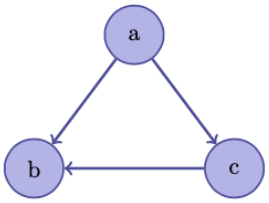
Leave question 3 for optional practice at home.

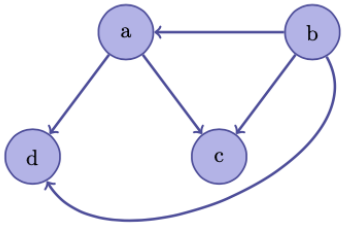
Parenthesis Theorem

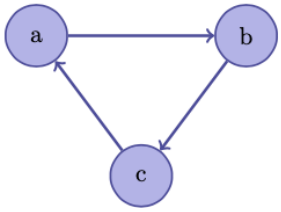
In any DFS of graph G, for any two vertices u and v, exactly one of these 3 conditions holds:

- The intervals [d[u],f[u]] and [d[v],f[v]] are entirely disjoint
- The interval [d[u],f[u]] is contained entirely in interval [d[v],f[v]]
- The interval [d[v],f[v] is contained entirely in interval [d[u],f[u]]

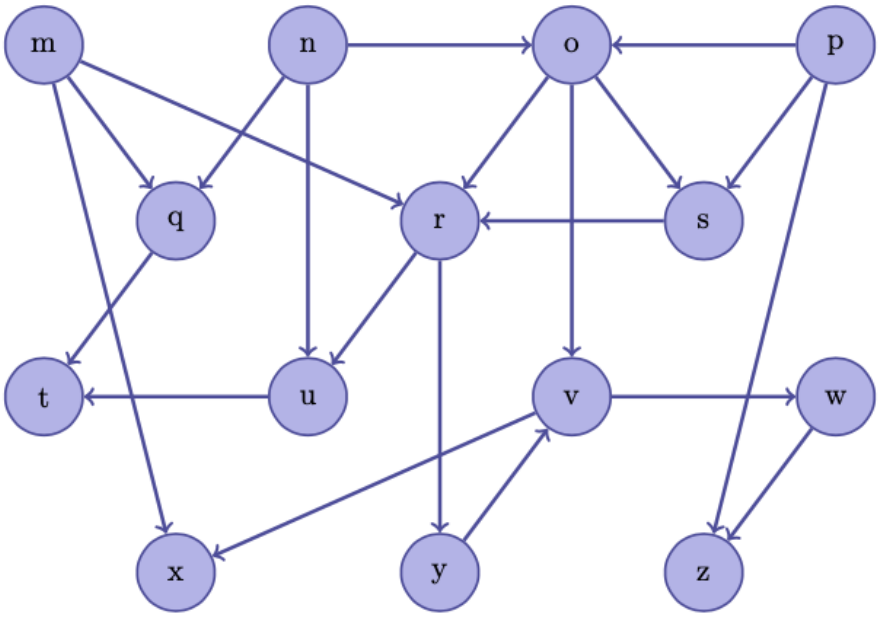
W10 Worksheet Q4 Discussion







W10 Worksheet Q4 Discussion



Topological Sort

- For directed acyclic graphs only
- A linear ordering of all vertices in graph G
- Such that if G contains (u,v) then u comes before v in the order
- Algorithm:

  run DFS on G, collect finish times as each vertex finishes into front of list.

Strongly Connected

- In an undirected graph, it is connected if

  there is a path between any 2 nodes

- We didn't previously define "connected" for a directed graph

  Not defined

- Strongly connected in a directed graph means that

  for any pairs of vertices u and v
  - u is reachable from v
  - v is reachable from u

$v \rightsquigarrow u$

$u \rightsquigarrow v$

s is reachable from any other vertex ⇒ make a new graph with edges and check if finish time is twice the # of vertices

DFS-Visit(G, s) ⇒ finish time of s twice the # of vertices

↳ any vertex is reachable from s.

Strongly Connected Directed Graph

- Algorithm to test?

Do DFS-Visit on every vertex (OR) or prove DFS-Visit(G, s) & s is reachable from any other vertex

- In directed graph G=(V, E), a strongly-connected component is the maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have

Minimum Spanning Trees

Two ideas for our algorithm

- Take a graph and remove expensive edges until we have a MST
- Start with vertices and add edges from graph making a forest / tree as we go, until we have MST

Better

spanning tree → connected
→ undirected

weighted for minimum spanning tree

Greedy Minimum Spanning Tree

```
GREEDY-MST(G=(V,E),w:E →R)
1    T ← {} # invariant:
2    while T is not a spanning tree:
3        find e "safe for T"
4        T ← T ∪ {e}
```

w → weight

T is a subset of some min. spanning tree of G

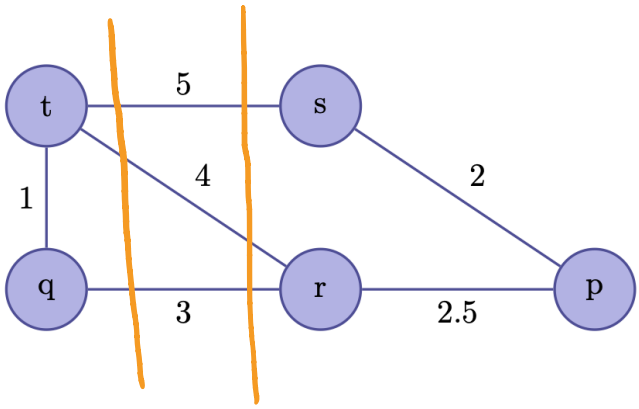
edge is safe iff T ∪ {e} is a subset of some MST of G.

Theorem

If G is a connected, undirected, weighted graph, T is a subset of some MST of G, and e is an edge of minimum weight whose end points are in different connected components of T then e is safe for T

Theorem

If G is a connected, undirected, weighted graph, T is a subset of some MST of G, and e is an edge of minimum weight whose endpoints are in different connected components of T, then e is safe for T



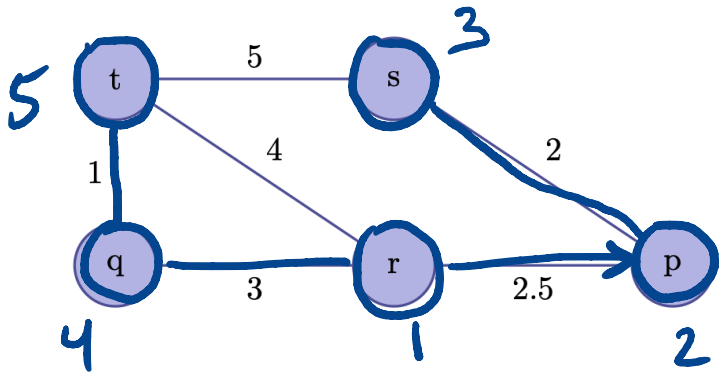
Prim's Algorithm

Idea: pick a root vertex and grow T by connecting an isolated vertex to T.

At each step: connect the vertex not currently in T that has the cheapest edge connection

Prim's Algorithm

start with r



Prim's Algorithm

```
MST-PRIM(G=(V,E),w:E→ R, r):
1    for each v in V:
2        priority[v] ← ∞
3        π[v] ← NIL
4    priority[r] ← 0
5    Q ← V
6    while Q is not empty:
7        u ← ExtractMin (Q)
8        T ← T ∪ {π[u], u} # except when u=r
9        for each v in adj[u]
10           if v in Q and w(u,v) < priority[v]:
10               priority[v] ← w(u,v)
11               decreasePriority(Q, v, new-priority)
12               pi[v] ← u
13
```

O(|V|) for building heap

O(|V| + |E|)

↳ pointer to the element

we know that this is a connected graph, so

$|E| \geq |V|$

so $O(|V| + |E|) = O(|E|)$

with binary min-heap, this $O(|E| \log |V|)$

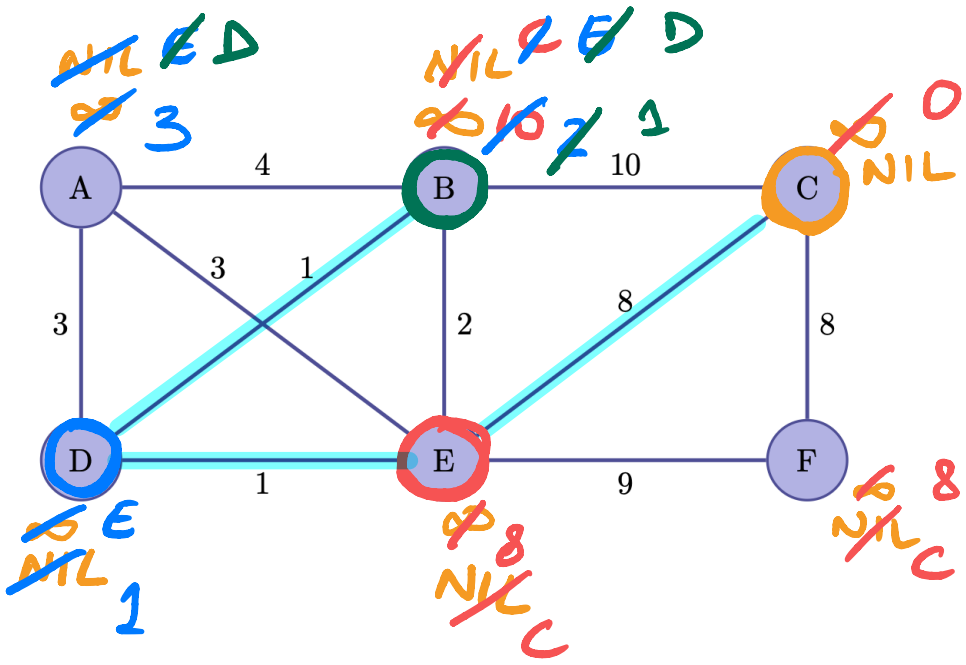
depends on how we implement PQ

$O(|E| \log |V|)$

$\log |V|$ cuz we are bubbling up to resize priority

Chalk talk

Tracing Prim's Algorithm



→ minimum spanning tree is only for an undirected graph

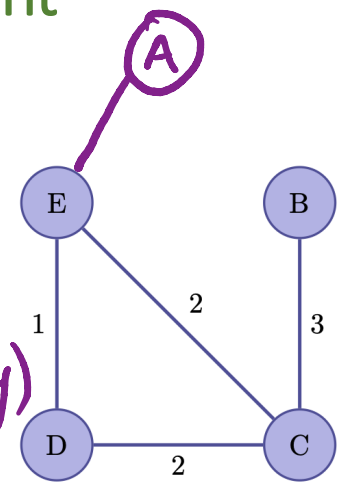
→ MST has |V| - 1 edges

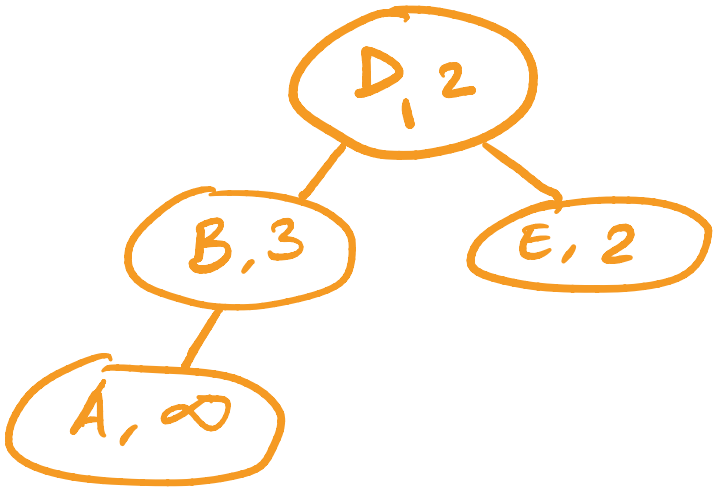
→ graph with MST must be connected

→ Prim's algo adds 1 vertex to the tree, it is building on every iteration.

Making Prim's Algorithm Efficient

- For Queue, use priority queue
- Line 11?
  - ↳ decrease priority (Q, v, priority)





In prim's, we build MST one vertex at a time.

Kruskal's Algorithm

Idea: Don't build a tree until the end. Instead, build a forest and merge trees in our forest until we have only 1 tree.

At each step:

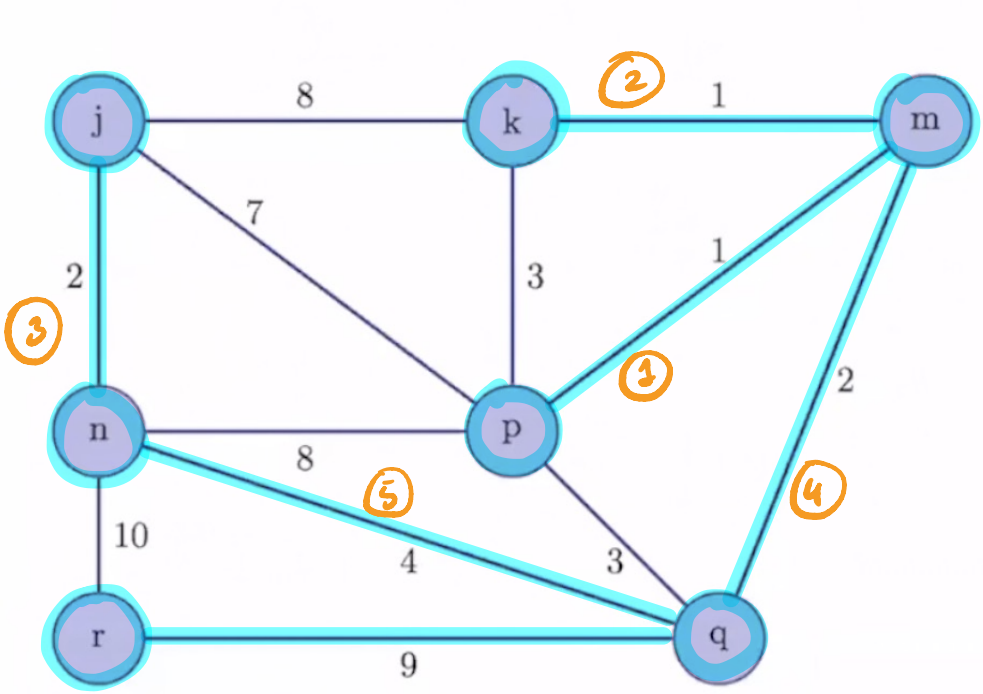
Pick the cheapest edge of graph G

That does not make a cycle

Kruskal's Algorithm

MST-KRUSKAL(G=(V,E),w:E $\rightarrow$R)

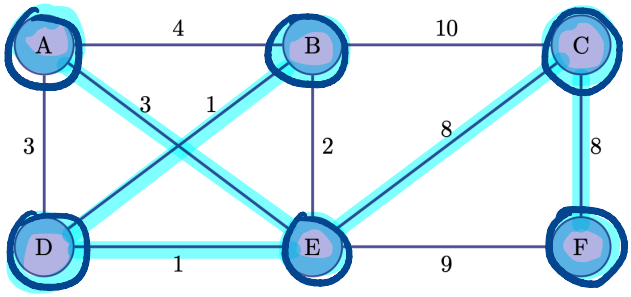
1 T $\leftarrow$ {}
2 sort edges so $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$ → $O(|E| \log |E|)$ to sort all these edges
3 for i $\leftarrow$ 1 to m:
4 let $(u_i, v_i) = (e_i)$
5 if u_i, v_i in different connected components of T:
T $\leftarrow$ T $\cup$ $\{e_i\}$



Worksheet Q7

Report when you finish Q7 in the Google Doc row for your group or in the chat. If you are watching the video, pause until you complete Q7.

http://www.tinyurl.com/breakouts-263



Kruskal's Algorithm: Complexity

What is the hard part of executing this efficiently?

checking if u_i, v_i are in different components of T. (as we go)

if use DFS or BFS ← O(|V|)

loop executes m times |E| times

↳ O(|E||V|)

We need a new data structure!

Disjoint Set ADT

→ no intersection, no common elements in a set.
→ each element in at most 1 set

MAKE-SET(x): Given an element x that doesn't already belong to a set, create a new set containing only x and ... designate x as representative

FIND-SET(x): Given an element x, find the representative for the set that contains x (or NIL if x is not in any set)

UNION(x,y):

Given 2 elements x and y.
S_x is the set containing x
S_y is the set containing y

Make a new set $S_x \cup S_y$, designate a representative and remove S_x and S_y in the ADT

Kruskal's Algorithm

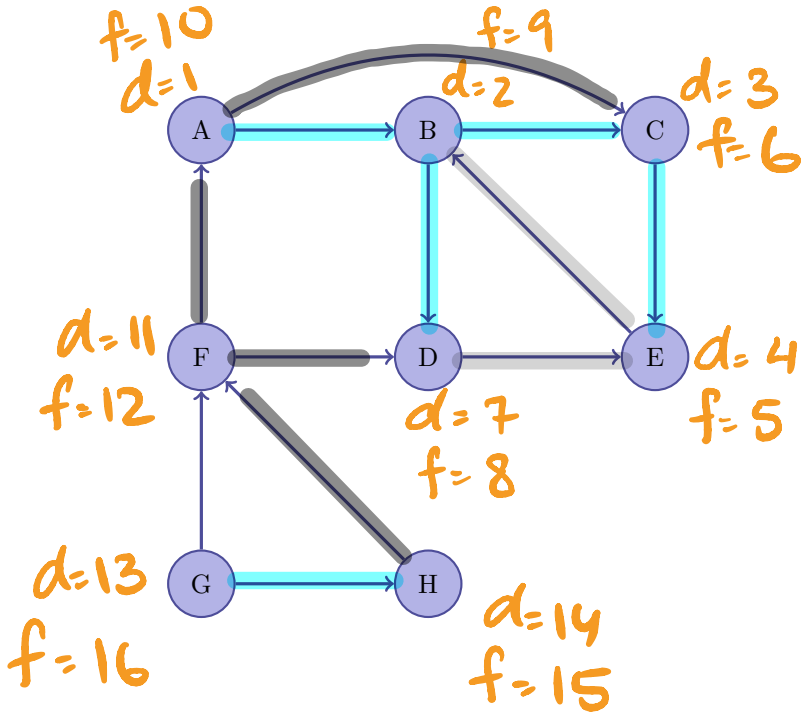
MST-KRUSKAL(G=(V,E),w:E $\rightarrow$R)

1 T $\leftarrow$ {}
2 sort edges so w(e_1) $\leq$ w(e_2) $\leq$ ... $\leq$ w(e_m)

3 for i $\leftarrow$ 1 to m:
4 let $(u_i, v_i) = (e_i)$
5 if u_i, v_i in different connected components of T:

T $\leftarrow$ T $\cup$ $\{e_i\}$

1. On the graph below, run DFS. Use the adjacency list provided to determine the order in which the vertices and edges are considered. Computer the discovery and finish times and draw the resulting DFS forest as well as the timeline and the parenthesis representation on the timeline.




```
A: B, C
B: C, D
C: E
D: E
E: B
F: A, D
G: F, H
H: F, G
```

2. Discuss in your group, what would have been different if the adjacency list had reordered the last three vertices, so that the outer loop finished with G, H, F (in that order.)

3. Optional Practice Homework - redo the same graph with this adjacency list representation.

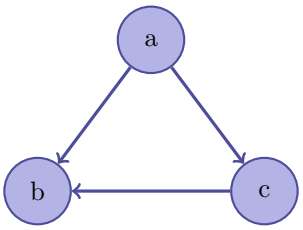
```
H: F, G
G: F, H
F: A, D
E: B
D: E
C: E
B: C, D
A: B, C
```

4. For each of the following graphs, can you order the vertices such that for every edge (u,v) in the graph, u comes before v in your ordering? If you can do it, give the order. If you can't do it, discuss why with your group.

Topological sort when in reverse order of finishing

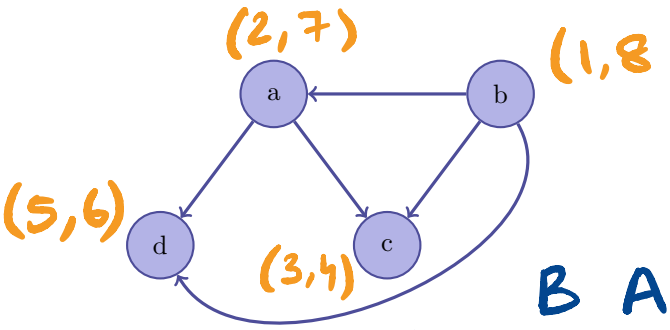
(a) Graph 1

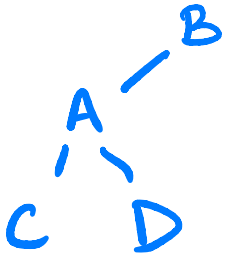
possible orders: 6



A C B ✓

(b) Graph 2

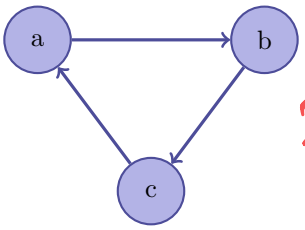




B A C D

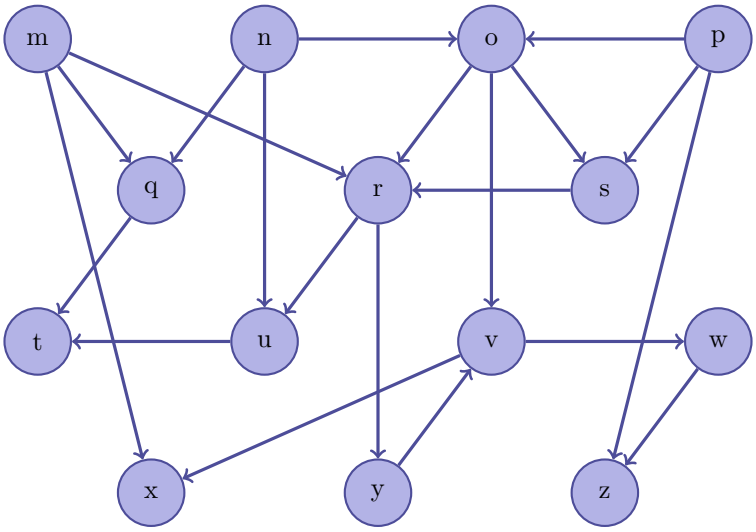
(OR) B A D C

(c) Graph 3



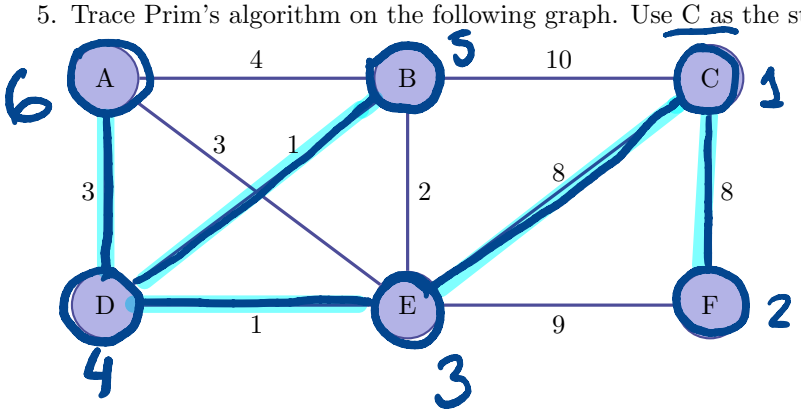
Impossible

(d) Graph 4

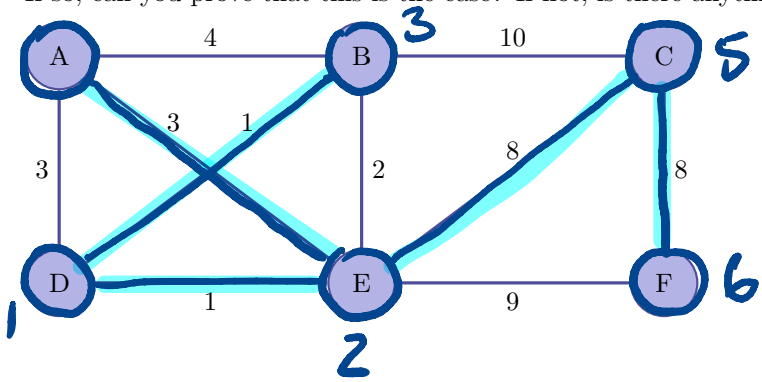


This doesn't have a cycle

5. Trace Prim's algorithm on the following graph. Use C as the start vertex.



6. Trace it again on a new copy below starting with a different vertex. Do you always get exactly the same tree? If so, can you prove that this is the case? If not, is there anything you can say about the trees that you get?

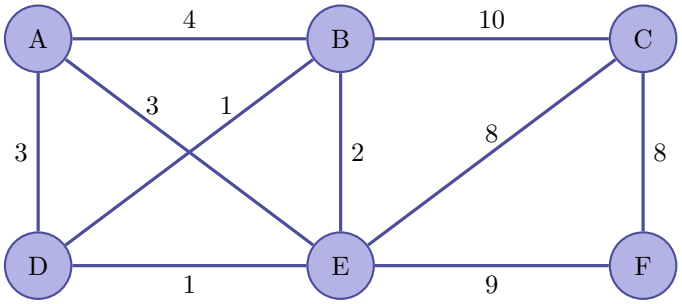


Start D

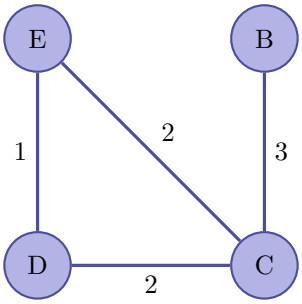
It's diff.

There are several minimum spanning trees for a graph

7. Trace Kruskal's algorithm on the same graph as we used in question 6. Here is another copy so you can write directly on it.

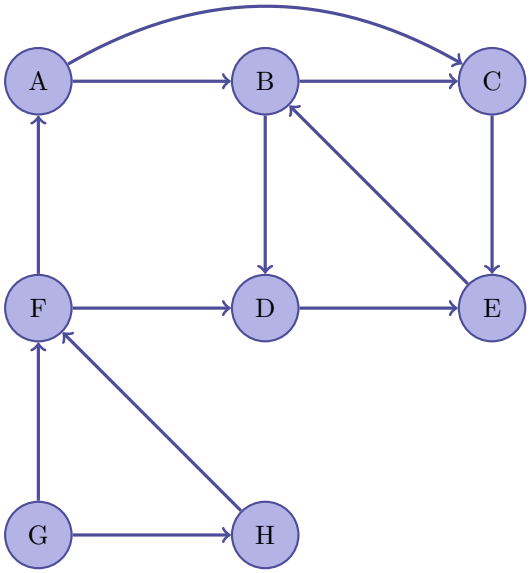


8. Trace Kruskal's algorithm in more detail on the following graph. This time, keep track of the disjoint sets of vertices.



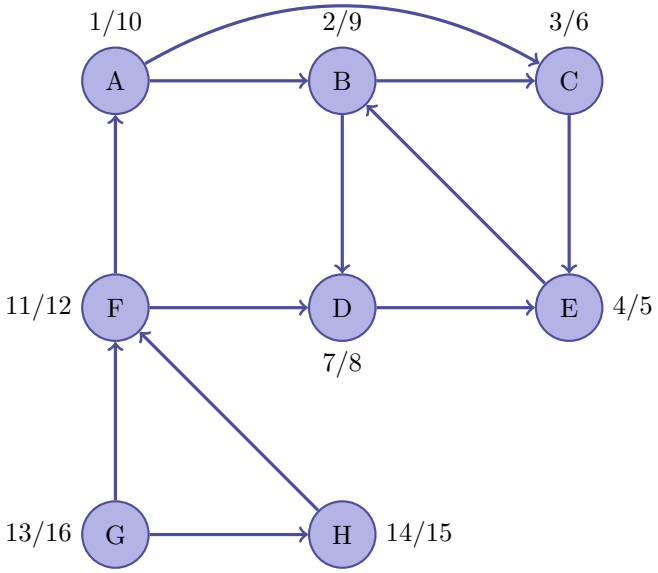
Only the solution to problem 1 is included here. The others are either left as optional practice problems without solutions or were taken up in class. If you are not confident about your solution for any of these problems, you are welcome to discuss on Piazza or bring them to office hours.

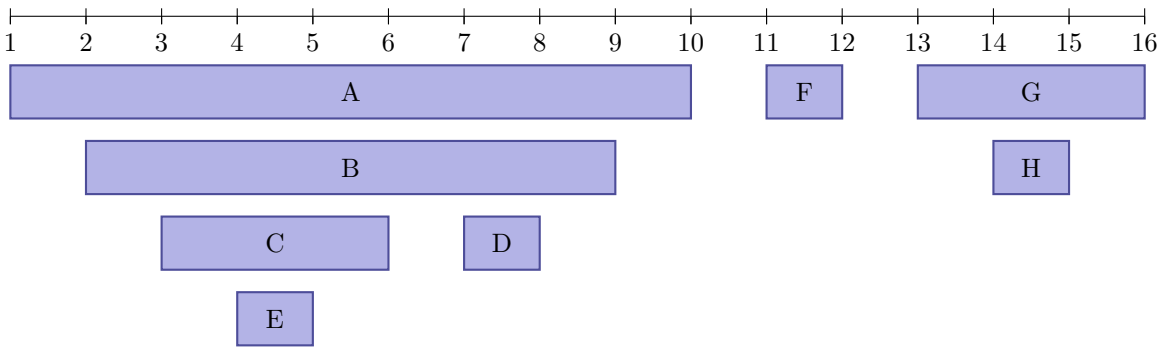
1. On the graph below, run DFS. Use the adjacency list provided to determine the order in which the vertices and edges are considered. Computer the discovery and finish times and draw the resulting DFS forest as well as the timeline and the parenthesis representation on the timeline.




```
A: B, C
B: C, D
C: E
D: E
E: B
F: A, D
G: F, H
H: F, G
```

Solution





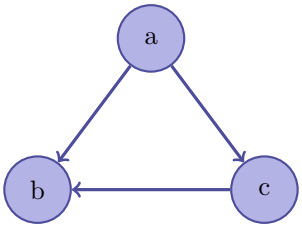
2. Discuss in your group, what would have been different if the adjacency list had reordered the last three vertices, so that the outer loop finished with G, H, F (in that order.)

3. Optional Practice Homework - redo the same graph with this adjacency list representation.

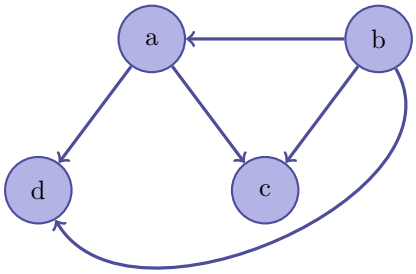
```
H: F, G
G: F, H
F: A, D
E: B
D: E
C: E
B: C, D
A: B, C
```

4. For each of the following graphs, can you order the vertices such that for every edge (u,v) in the graph, u comes before v in your ordering? If you can do it, give the order. If you can't do it, discuss why with your group.

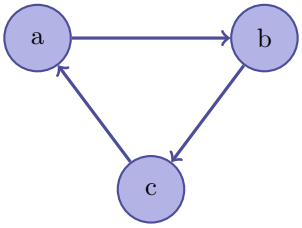
   (a) Graph 1



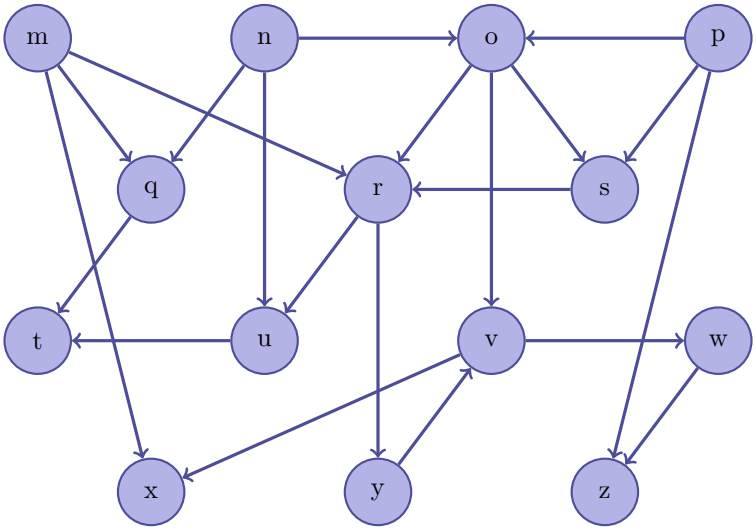
   (b) Graph 2



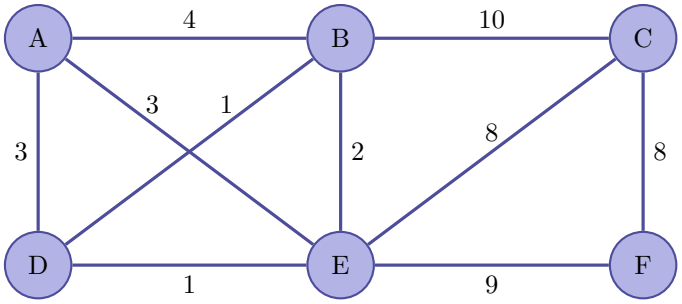
   (c) Graph 3



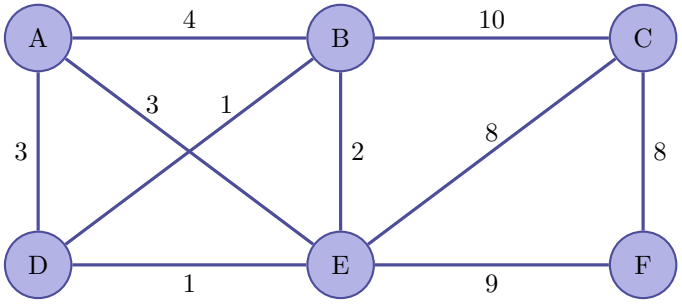
(d) Graph 4



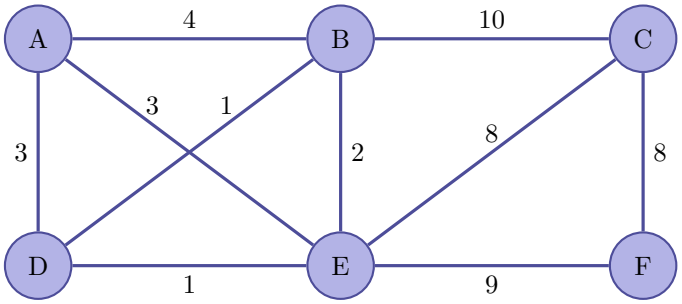
5. Trace Prim's algorithm on the following graph. Use C as the start vertex.



6. Trace it again on a new copy below starting with a different vertex. Do you always get exactly the same tree? If so, can you prove that this is the case? If not, is there anything you can say about the trees that you get?



7. Trace Kruskal's algorithm on the same graph as we used in question 6. Here is another copy so you can write directly on it.



8. Trace Kruskal's algorithm in more detail on the following graph. This time, keep track of the disjoint sets of vertices.