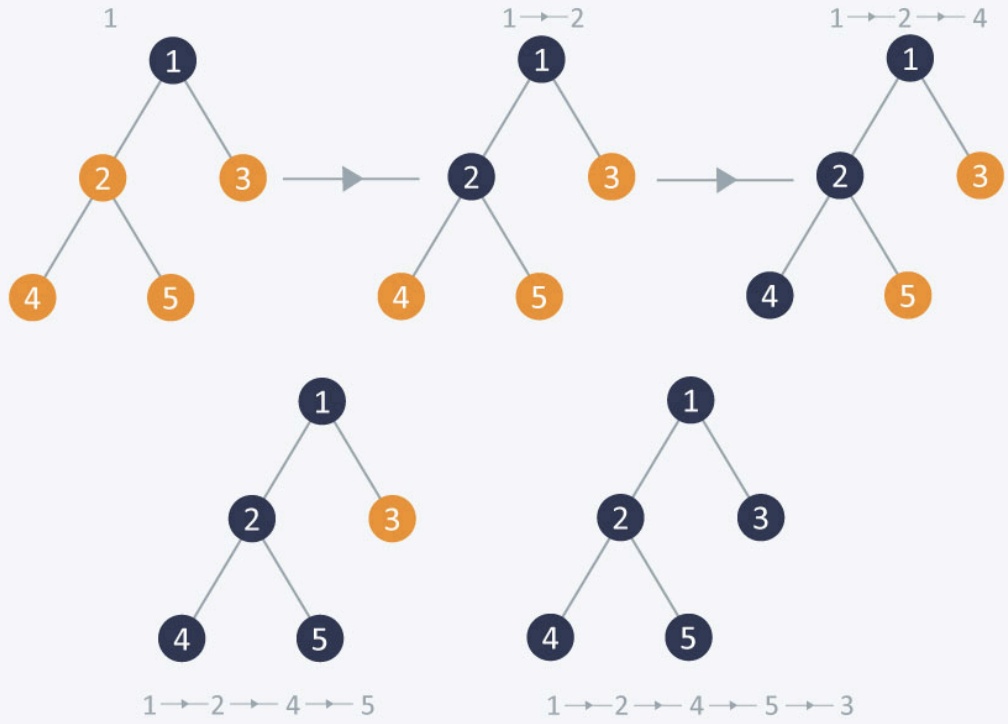
COMP20007 Workshop Week 6

1	**Preparation:** - open `ws6.pptx` from `github.com/anhvir/c207` - (optional) open `wokshop6.pdf` (from `LMS`)
2	**Topic 1:** DFS and BFS: *Group Work:* Problems T1, T2, T4, T3 (in order)
3	**Topic 2:** Dijkstra's and Prim's Algorithms *Group Work:* Problems T5, T6
LAB	**Assignment 1 Q&A**, and - make sure that you can scp, ssh, use dimefox to test/submit - do Assignment 1 or do not-yet-done exercises **Before leaving:** *Please give comments on the format of today's workshop*

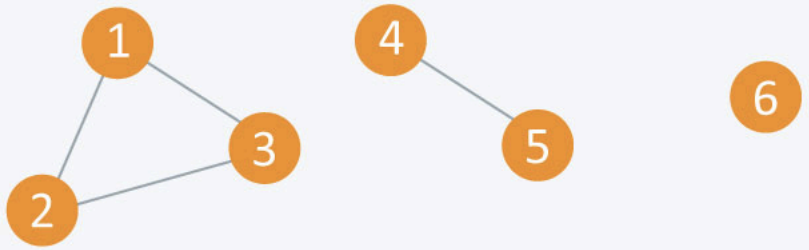
DfsExplore(1): explore a connected component from node 1

```
function DfsExplore(v)
    count ← count + 1
    mark v with count
    for each edge (v, w) do
        if w is marked with 0 then
            DfsExplore(w)
```



DFS: explore a whole (multi-component) graph

```
function DFS(⟨V, E⟩)
    mark each node in V with 0
    count ← 0
    for each v in V do
        if v is marked 0 then
            DfsExplore(v)
```

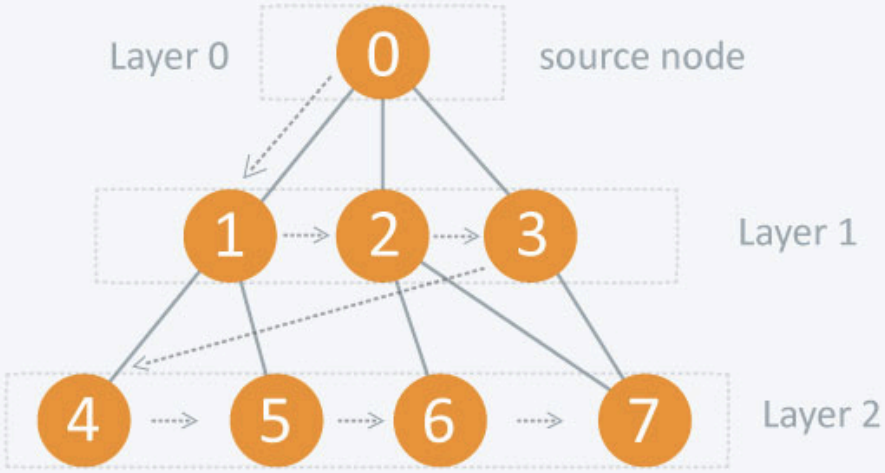



```
function DfsExplore(v)
    count ← count + 1
    mark v with count
    for each edge (v, w) do
        if w is marked with 0 then
            DfsExplore(w)
```

Using DFS how can we discover that:
- this graph has 3 components?
- this graph is cyclic?

Using DfsExplore, how can we list all possible paths from 1 → 2 ?

BfsExplore(O): explore a component from node O, in BFS manner



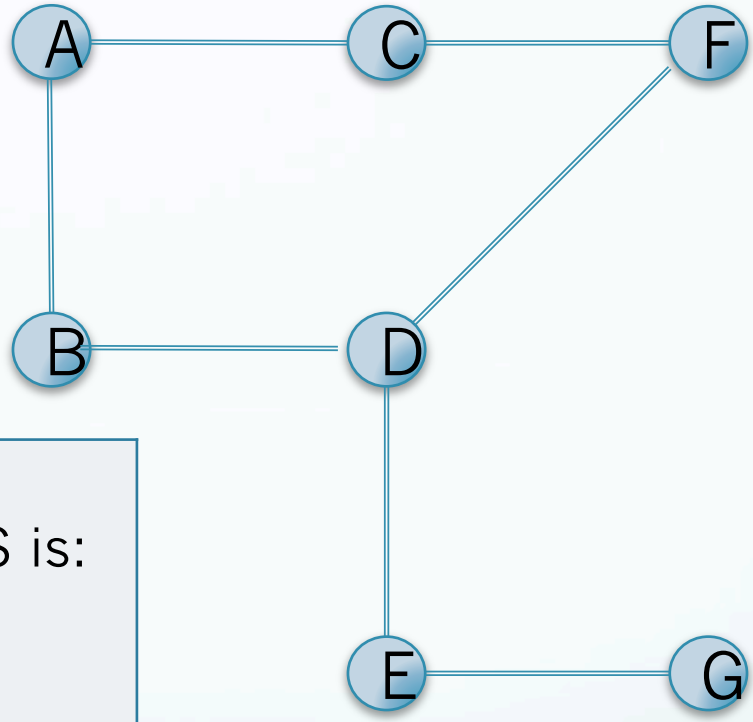

```
function BFS(⟨V, E⟩)
    mark each node in V with 0
    count ← 0, init(queue)
    for each v in V do
        if v is marked 0 then
            BfsExplore(v)
```

function BfsExplore(v)

```
count ← count + 1
mark v with count
inject(queue, v)                                  ▷ queue c
while queue is non-empty do
    u ← eject(queue)
    for each edge (u, w) adjacent to u do
        if w is marked with 0 then
            count ← count + 1
            mark w with count
            inject(queue, w)
```

T1: Depth First Search

- List the order of the nodes visited by the a) DFS and b) BFS algorithms



YOUR ANSWER:

a) The order of the nodes visited by DFS is:

A

b) The order of the nodes visited by BFS is:

A

T3: Finding Cycles

a) Explain how one can also use BFS to see whether an undirected graph is cyclic. b) Which of the two traversals, DFS and BFS, will be able to find cycles faster? (If there is no clear winner, give an example where one is better, and another example where the other is better. − but skip this part if it takes more than 1 minute)

YOUR BRIEF ANSWER:

```
function BFS(⟨V, E⟩)
    mark each node in V with 0
    count ← 0, init(queue)
    for each v in V do
        if v is marked 0 then
            BfsExplore(v)
```

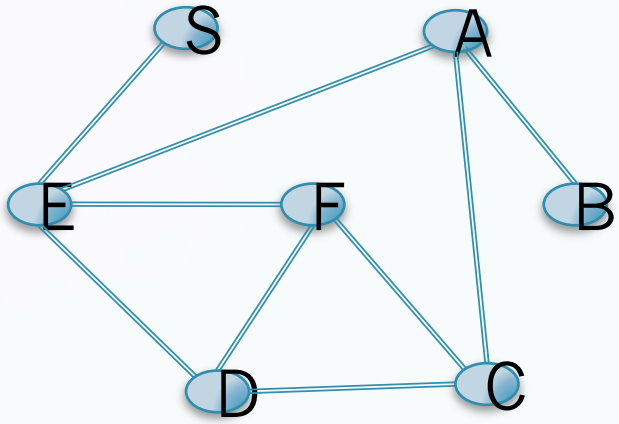
```
function BfsExplore(v)
    count ← count + 1
    mark v with count
    inject(queue, v)                    ▷ queue c
    while queue is non-empty do
        u ← eject(queue)
        for each edge (u, w) adjacent to u do
            if w is marked with 0 then
                count ← count + 1
                mark w with count
                inject(queue, w)
```

T4: 2-Colourability

Design an algorithm to check whether an undirected graph is 2-colourable, that is, whether its nodes can be coloured with just 2 colours in such a way that no edge connects two nodes of the same colour.

To get a feel for the problem, try to 2-colour the following graph (start from **S**).

Do you expect we could extend such an algorithm to check if a graph is 3-Colourable, or in general: k-Colourable?



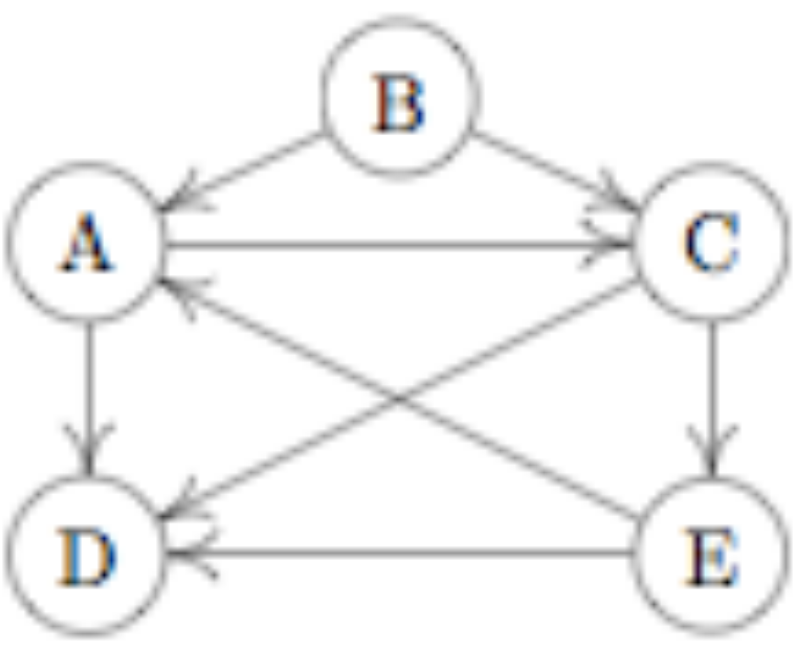
YOUR BRIEF ANSWER:

a) try to 2-colour the above graph, starting from **S**, using 2 colours + and −
hint: what is colour for S? how do we continue?

b) So, how to solve the 2-colourability?

c) Do you expect we could extend such an algorithm to check if a graph is 3-Colourable, or in general: k-Colourable?

T2



A DFS of a di-graph can be represented as a collection of trees. Each edge of the graph can then be classified as a *tree edge*, a *back edge*, a *forward edge*, or a *cross edge*. A tree edge is an edge to a previously un-visited node, a back edge is an edge from a node to an ancestor, a forward edge is an edge to a non-child descendent and a cross edge is an edge to a node in a different sub-tree (i.e., neither a descendent nor an ancestor)

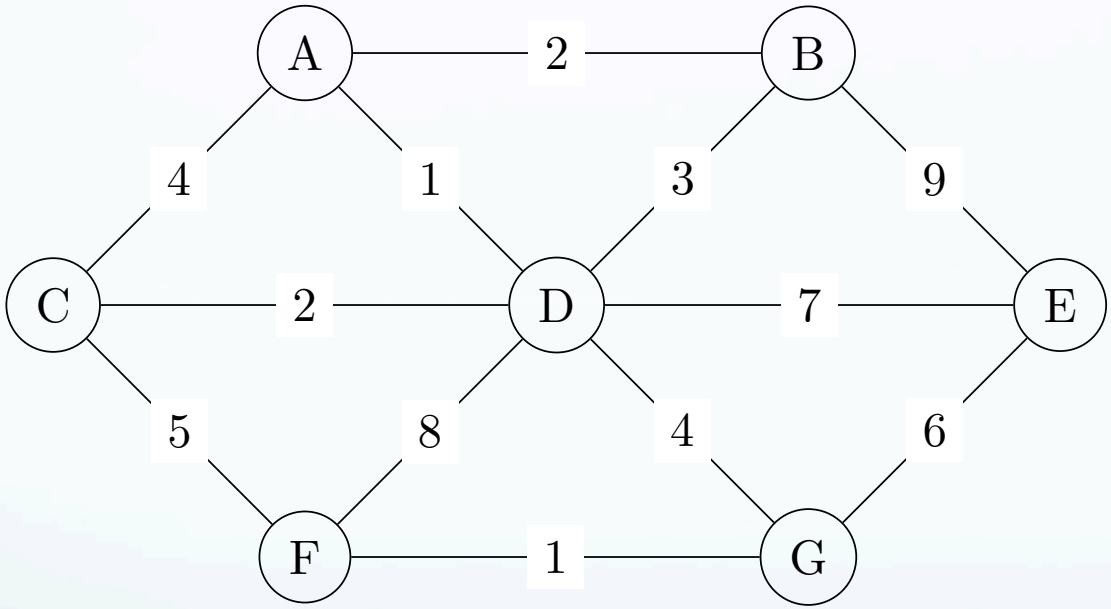
Draw a DFS tree based on the following graph, and classify its edges into these categories.

In an undirected graph, you wont find any forward edges or cross edges. Why is this true? You might like to consider the graph above, with each of its edges replaced by undirected edges.

A gift from Anh: If you are a bit bored or tired, skip this exercise, and instead use pages 23-35 (of this file) to entertain yourselves ;-)

Dijkstra's algorithm

- What's that?



Dijkstra's Algorithm

purpose	SSSP (that involves all nodes of a *connected* graph)	

init	`for all v ∈V :`
	`  dist[v]= ∞`
	`  prev[v]= nil`
	`dist[S]= 0`	

basic loop	for all `u∈V`, at each step choose `u` that `dist[u]` is min amongst the unselected	

action after selecting `u`	`for all unselected v such that (u,v)∈E:`
	`  if dist[v]>dist[u]+w(u,v):`
	`  update dist[v]`
	`  update prev[v]`	

Q: what is `update dist[v]` ? `update prev[v]` ?

why Dijkstra's algorithm is *greedy*?

Dijkstra's and Prim's are similar?

	Dijkstra's	Prim's
purpose	SSSP (that involves all nodes of a *connected* graph)	MST (that involves all nodes of a *connected* graph)

init	`for all v ∈V :`
	`  dist[v]= ∞`
	`  prev[v]= nil`
	`dist[S]= 0`
		same

basic loop	for all `u∈V`, at each step choose `u` that `dist[u]` is min amongst the unselected	same

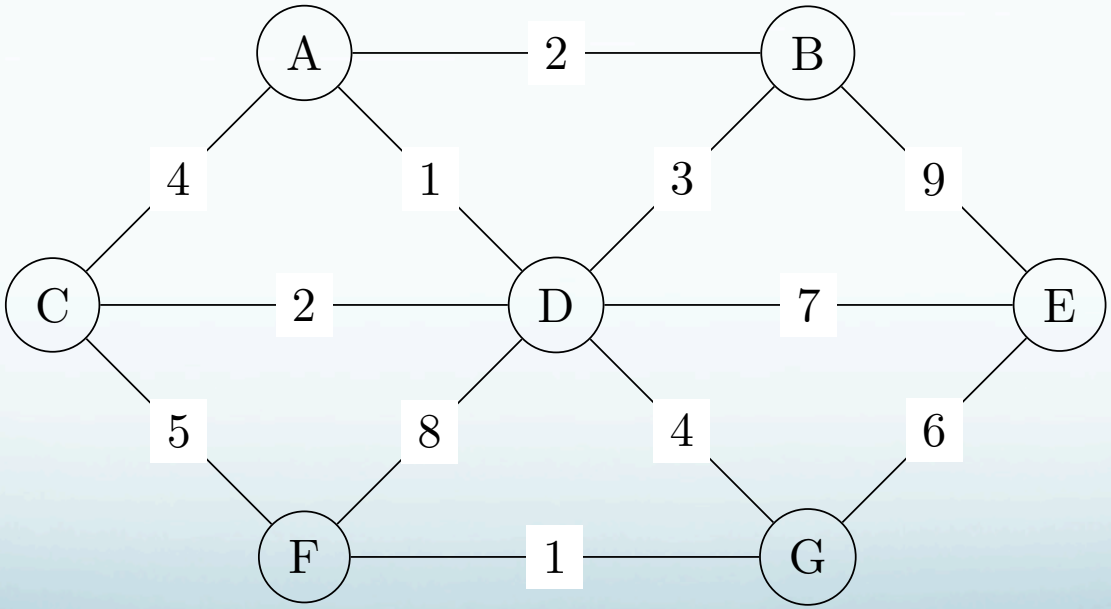
action after selecting `u`	`for all unselected v such that (u,v)∈E:`
	`  if dist[v]>dist[u]+w(u,v):`
	`  update dist[v]`
	`  update prev[v]`
		same loop to decide whether to include edge `(u,v)`:
		`if dist[v]>w(u,v):`
		same

So, the 2 alg are similar. The only difference is the optimizing functions. There are many other situations when the same algorithm applied, but perhaps with a different optimizing function.

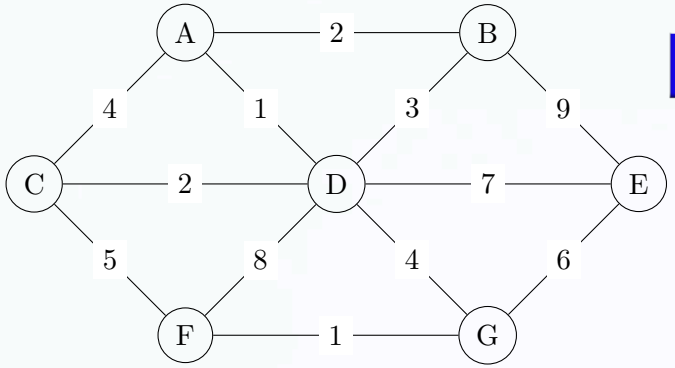
T5: SSSP with Dijkstra's Algorithm (DA)

Dijkstra's algorithm computes the shortest path to each node in a graph from a single starting node (the 'source'). Trace Dijkstra's algorithm on the following graph, with node E as the source

Repeat the algorithm with node A as the source. How long is the shortest path from E to A? How about A to F



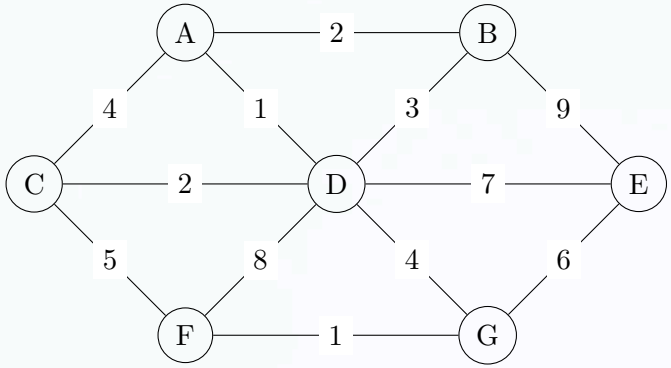
Dijkstra's Algorithm from E



dist[E] prev[E]

step	node visited	A	B	C	D	E	F	G
0		∞/nil	∞/nil	∞/nil	∞/nil	**0**/nil	∞/nil	∞/nil

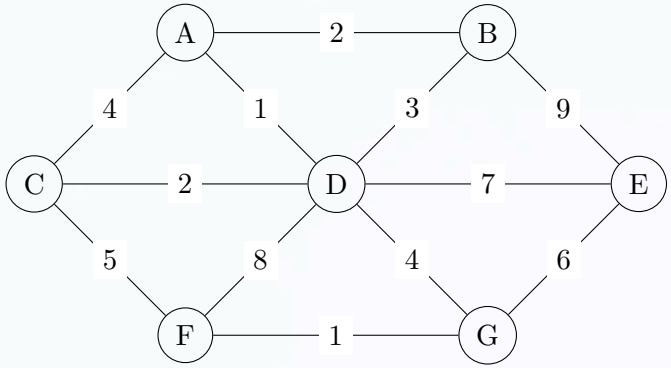
DA is a BFS. DA is used to find shortest path from a **source** to *all other nodes*. A *practical implementation* is to maintain 3 arrays:
dist[v] = shortest-distance-found-so-far from **source** to v
prev[v] = id of the node preceding v in the path-found-so-far
visited[v] = **Y** if v has been explored/visited, **N** if not yet



DA from E

step	node visited	A	B	C	D	E	F	G
0		∞/nil	∞/nil	∞/nil	∞/nil	**0**/nil	∞/nil	∞/nil
1	E		**9,E**		**7,E**			**6,E**

At step 1: choose the node with minimal dist ***amongst the unvisited nodes***. This is E. Then, consider to update dist[v] for v= {B, D, G}. **And, mark E as visited ???**.

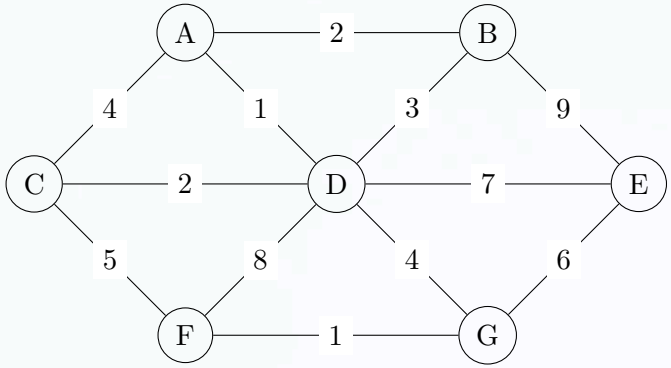


DA from E

How long, *and what is*, the shortest path from E to A?

A ← D ← E

step	node visited	A	B	C	D	E	F	G
0		∞/nil	∞/nil	∞/nil	∞/nil	**0**/nil	∞/nil	∞/nil
1	E	~	9/E	~	7/E		~	**6**/E
2	G(6)		**9/E**		**7/E**		**7,G**	
3	D (7)	**8,D**	**9/E**	9,D			**7,G**	
4	F(7)	**8,D**	**9/E**	9.D				
5	A(8)		**9/E**	9/D				
6	B(9)			9/D				
7	D							



DA from A

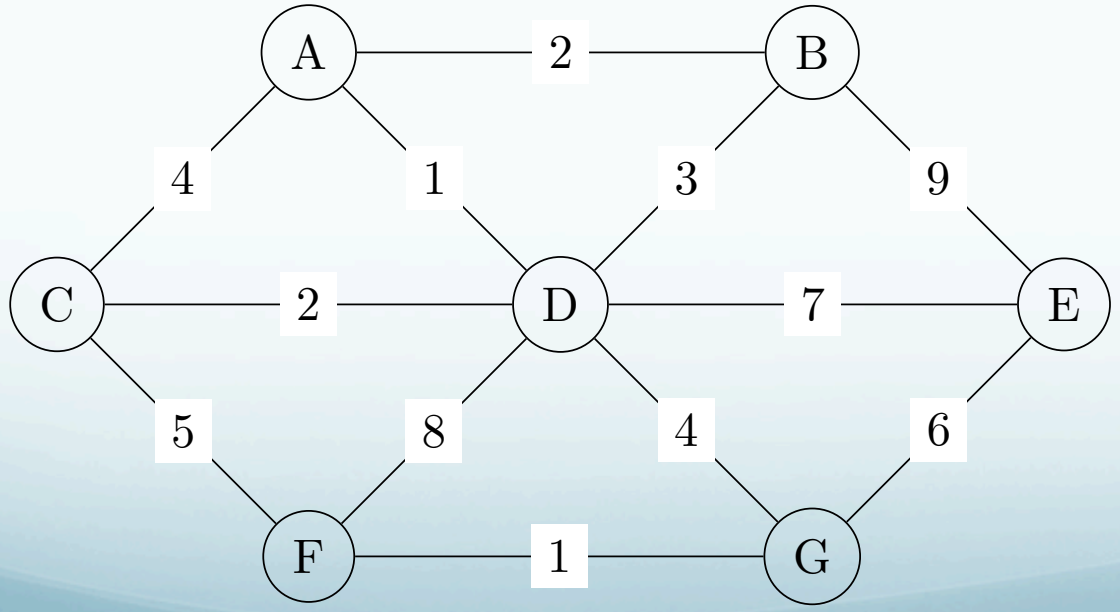
How long, *and what is*, the shortest path from E to A?
How about from A to F?

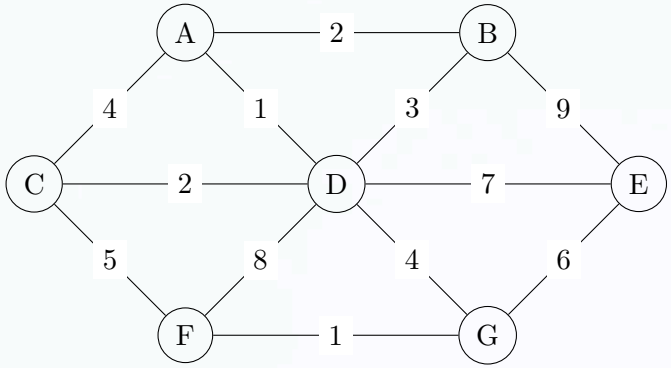
step	node visited	A	B	C	D	E	F	G
0		0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
1								
2								
3								
4								
5								
6								
7								

T6: Minimum Spanning Tree with Prim's Algorithm

Prim's algorithm finds a minimum spanning tree for a weighted graph. Discuss what is meant by the terms 'tree', 'spanning tree', and 'minimum spanning tree'.

Run Prim's algorithm on the graph below, using A as the starting node. What is the resulting minimum spanning tree for this graph? What is the cost of this minimum spanning tree?





Prim's Alg from A

What's the resulting MST?
What's the cost of that MST?

step	node visited	A	B	C	D	E	F	G
0		0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
1	a		2a	4a	1d			
2	d			2d				
3								
4								
5								
6								
7								

Food for our brain

- You're organizing your birthday party and inviting n friends. From these friends you know that there are some pairs of people who hates each other. You want to know if it's possible to divide guests into 2 different tables so that in each table there is no such hating pair? Design an algorithm for that.
- You have un unweighted undirected graph. Design an algorithm to find the shortest part (the part with least number of edges) between 2 vertices. What if the graph is weighted?

assignment 1

- Do it early! Submit early, resubmit if needed!
- Read & participate in discussion forum!
- Make sure that you follow well the specification.
- Check your writing part carefully..
- Test your program carefully: remember to test on dimefox

- Make sure you don't have memory leak:
  - check that every execution of `malloc` matches with an execution of `free`, and
  - [optional] use tools like `valgrind` to test for memory leak.

How to submit the programming part

Suppose that all all of your files are in directory `A1`. To submit:

- When on the parent directory of `A1`, copy the whole `A1` to `dimefox` using `scp` (or `pscp`, but perhaps you should install `scp`):

```
scp –r A1 username@dimefox.eng.unimelb.edu.au:
pscp –r A1 username@dimefox.eng.unimelb.edu.au:
```

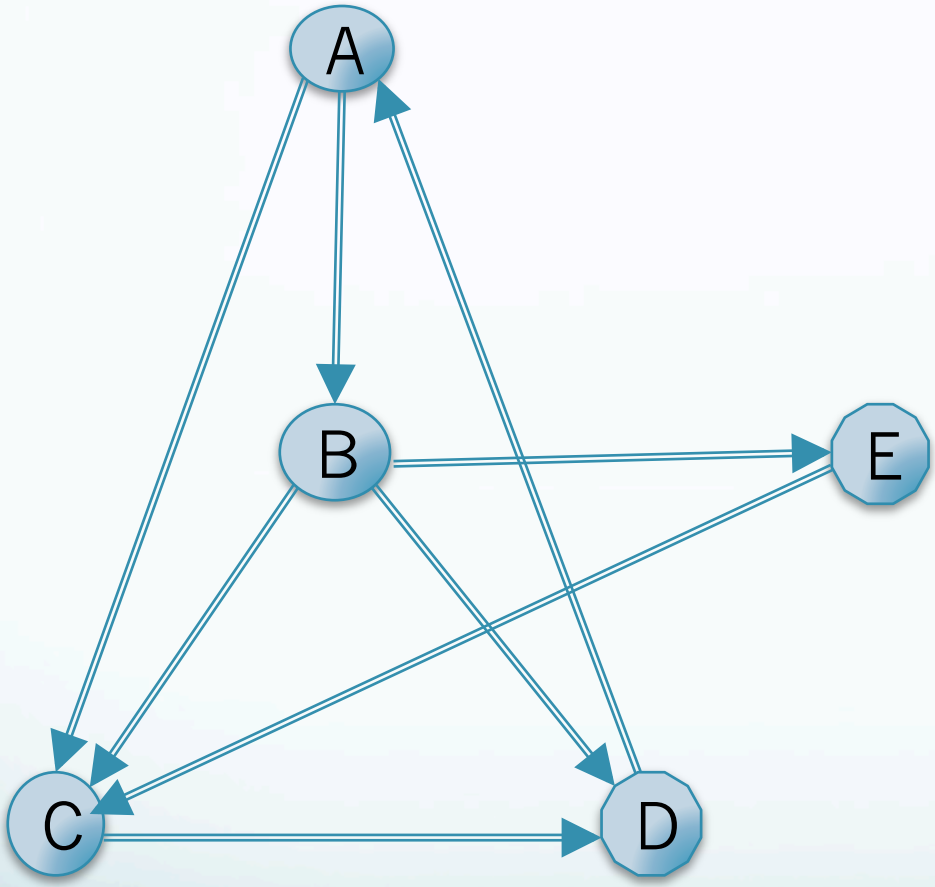
- Then, login into `dimefox` using `ssh`, after that, in `dimefox`, test your program with `make`, and do the test with sample data and sample outputs

- To submit, when on `dimefox` run:

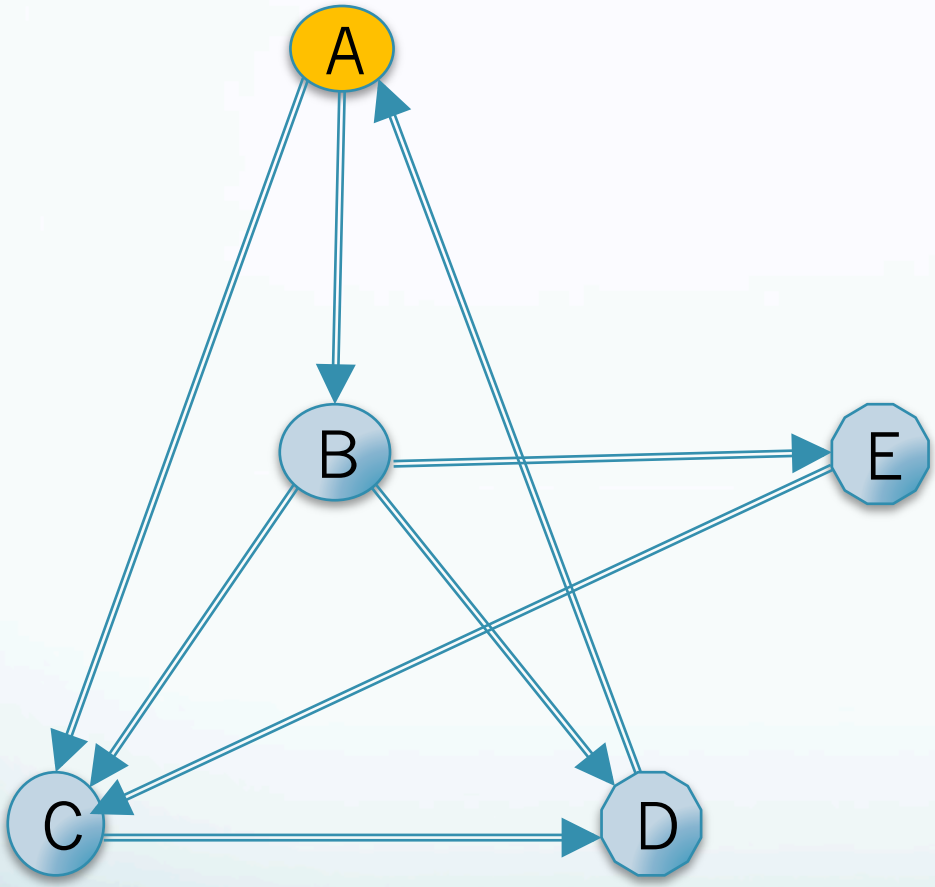
```
cd ~/A1
submit comp20007 a1 Makefile *.c *.h
```

Lab: do assmt1 or your choice

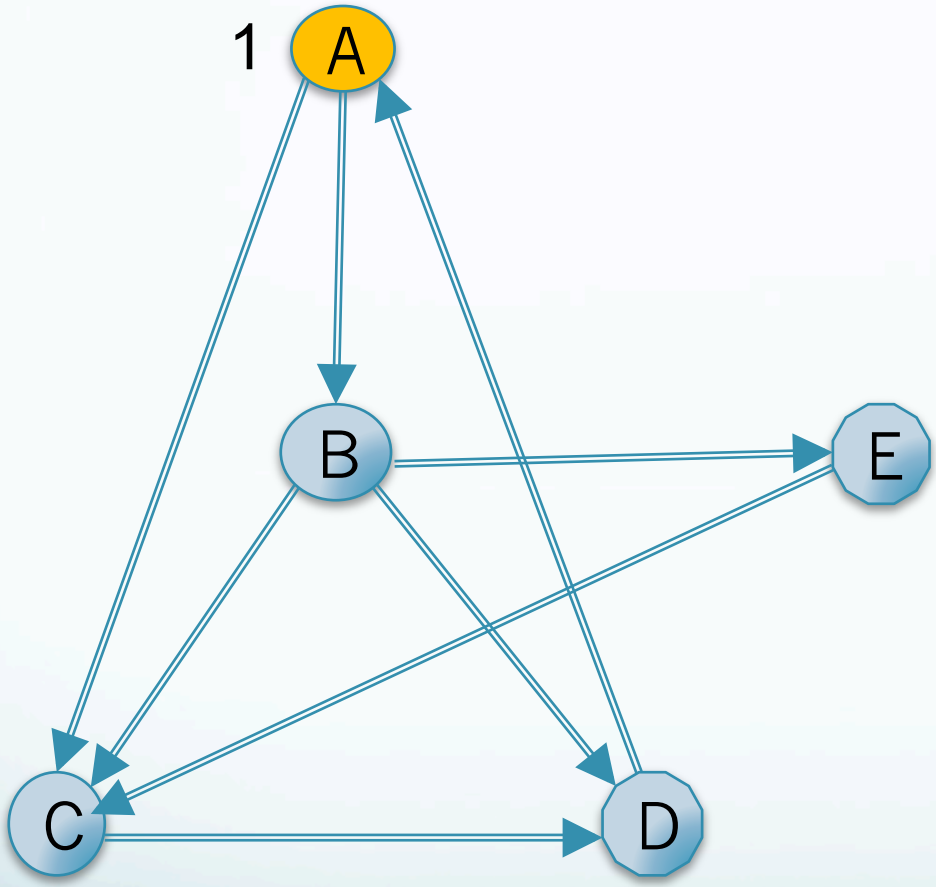
Extra: understanding DFS on di-graphs and the concepts of tree-, back-, forward-, cross-edges



DFS

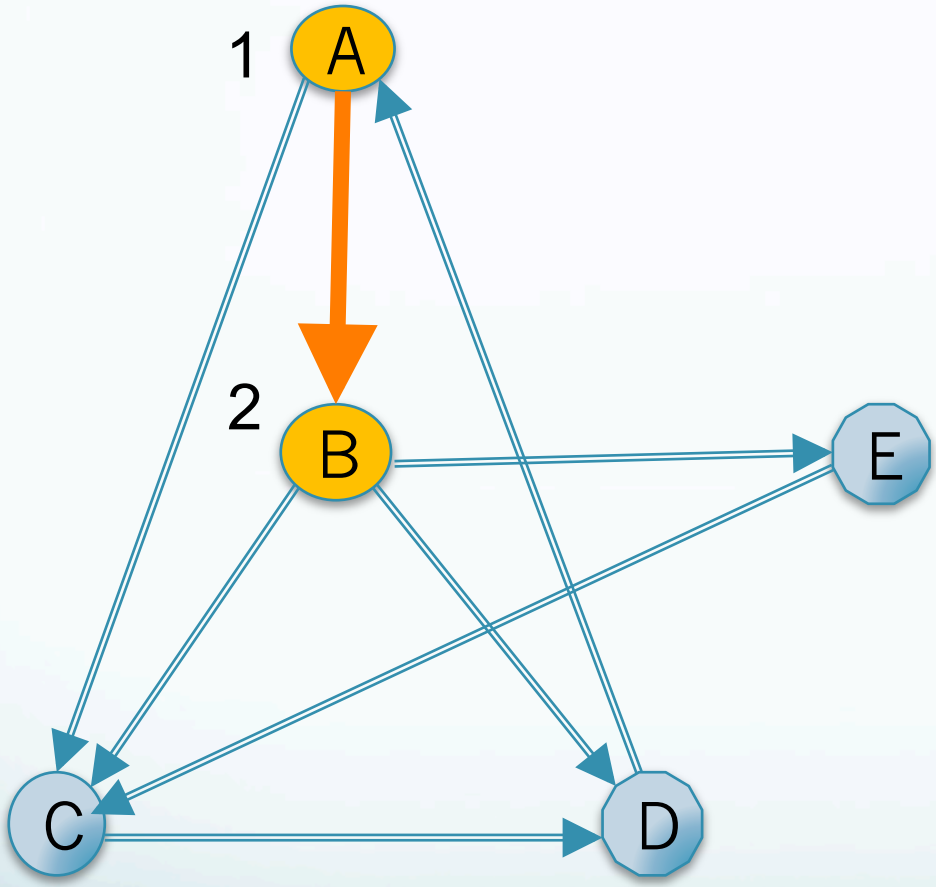


DFS

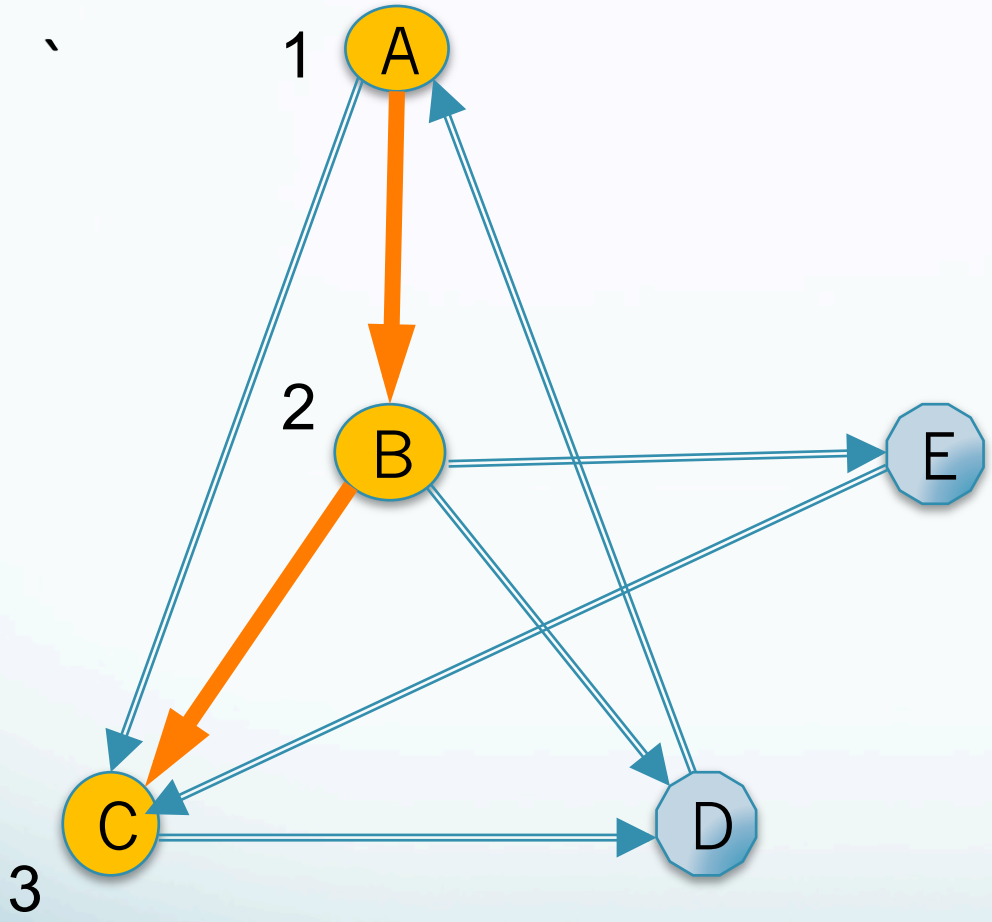


note: black number 1 is the push order

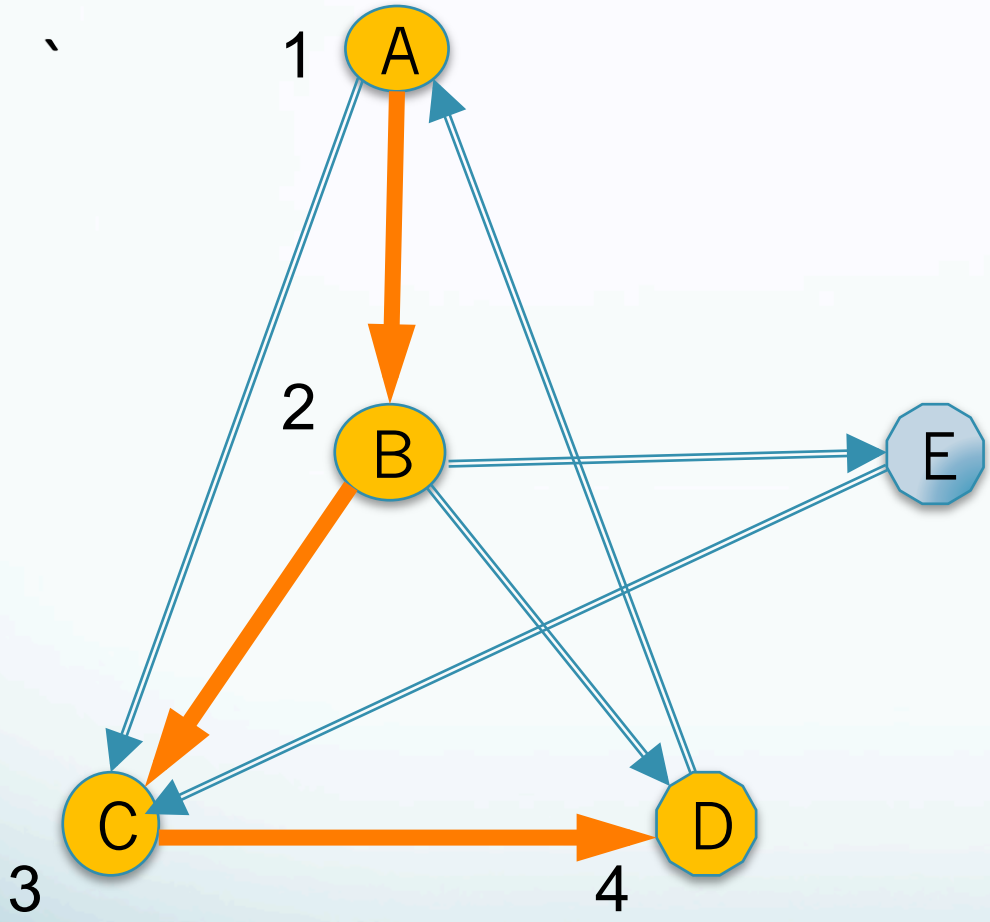
DFS



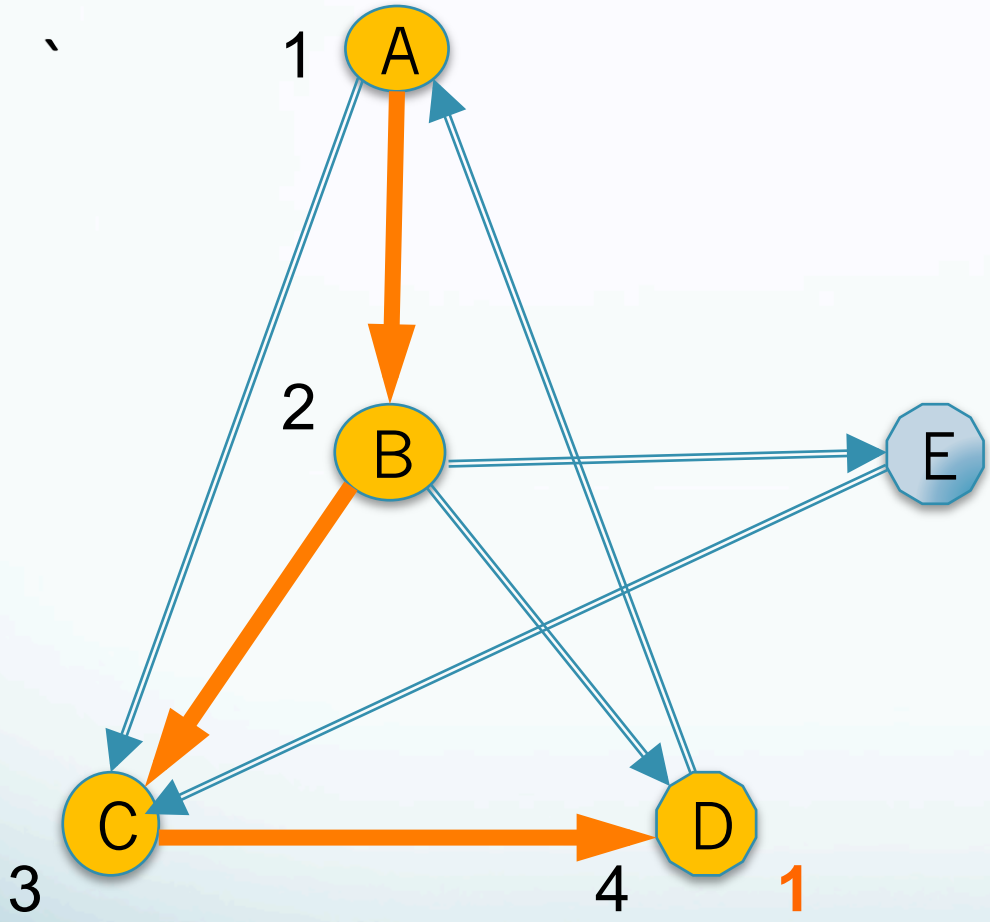
DFS



DFS

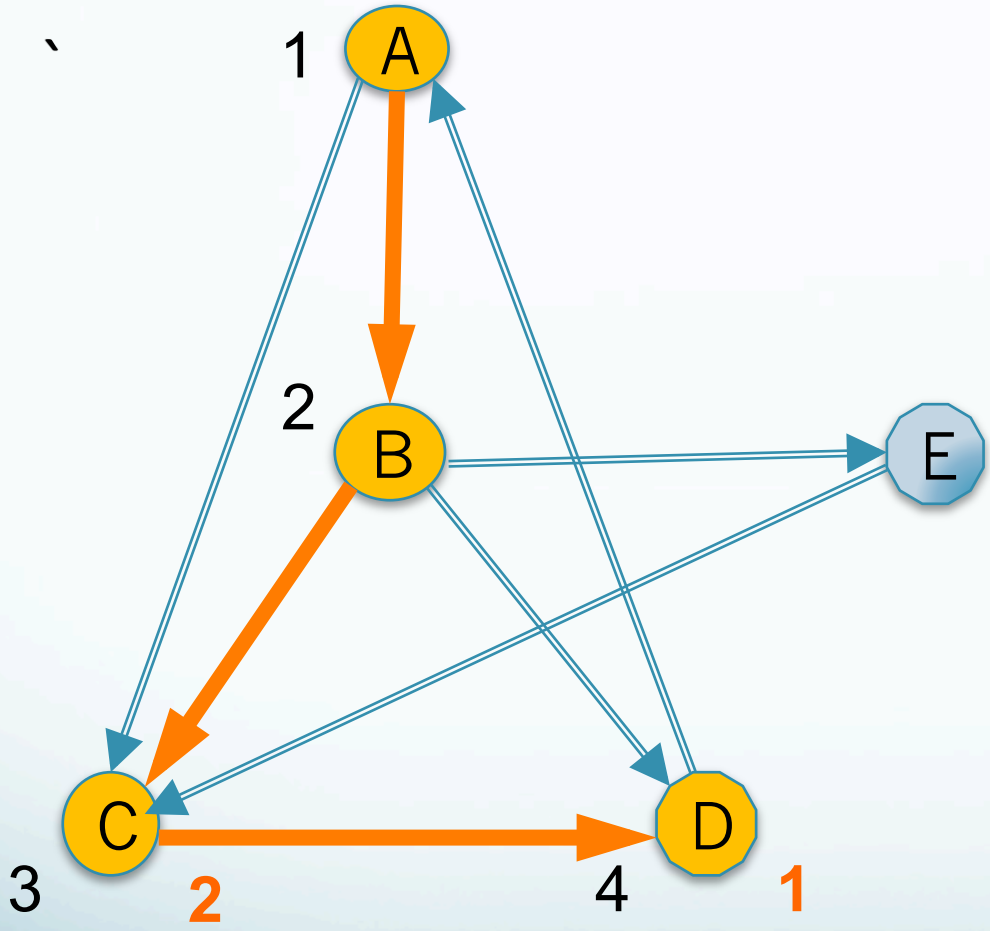


DFS

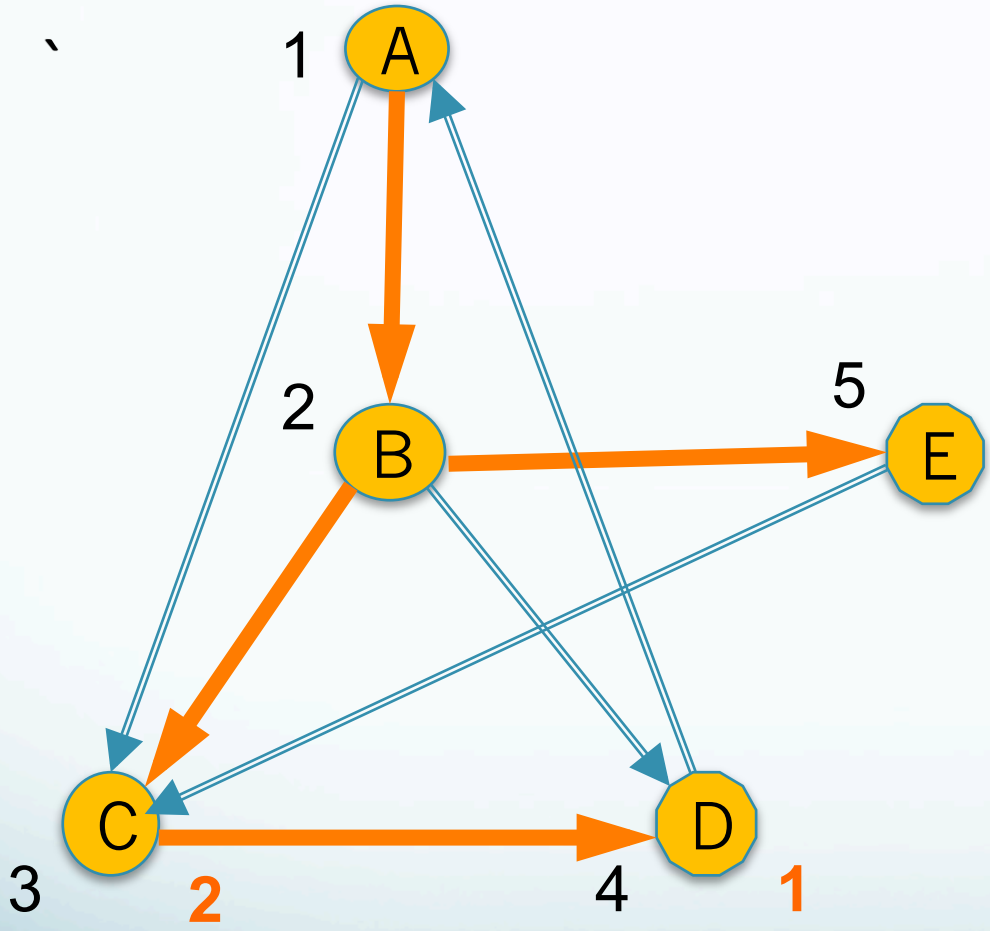


note: orange number **1** represents the pop order

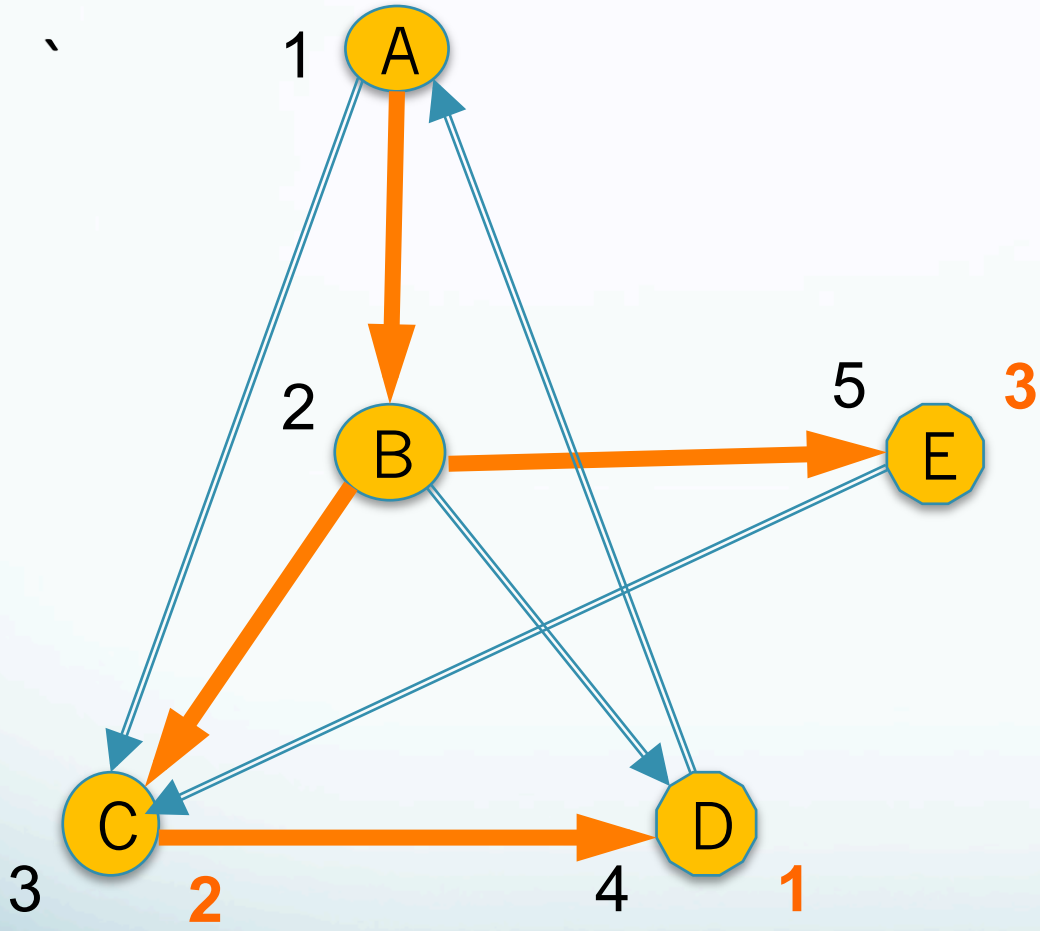
DFS



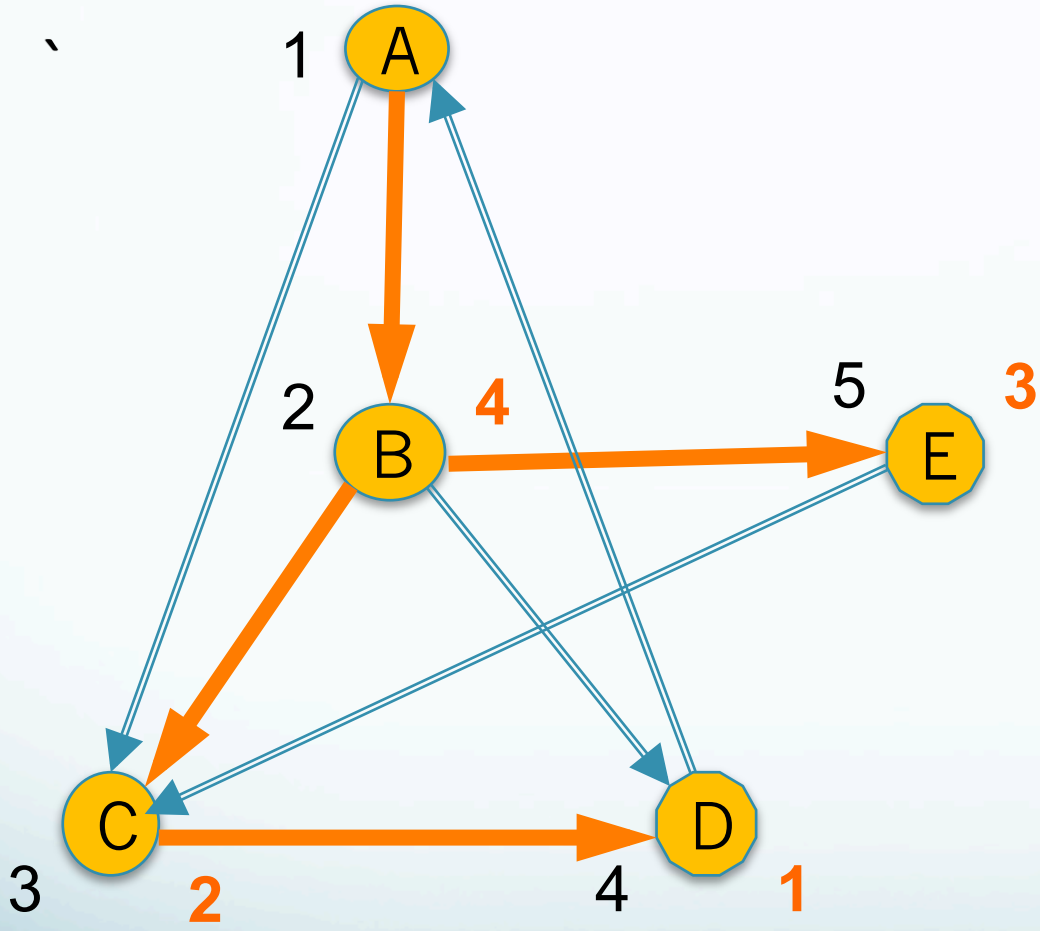
DFS



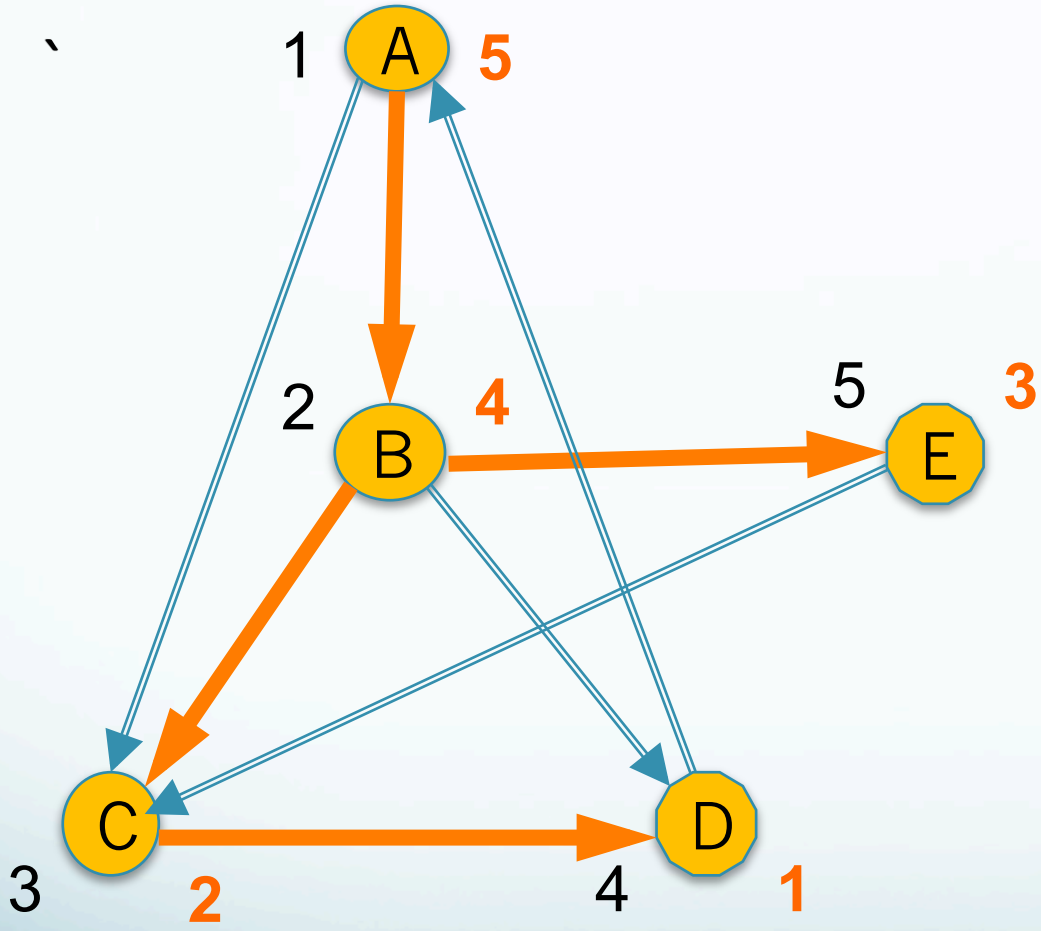
DFS



DFS



DFS

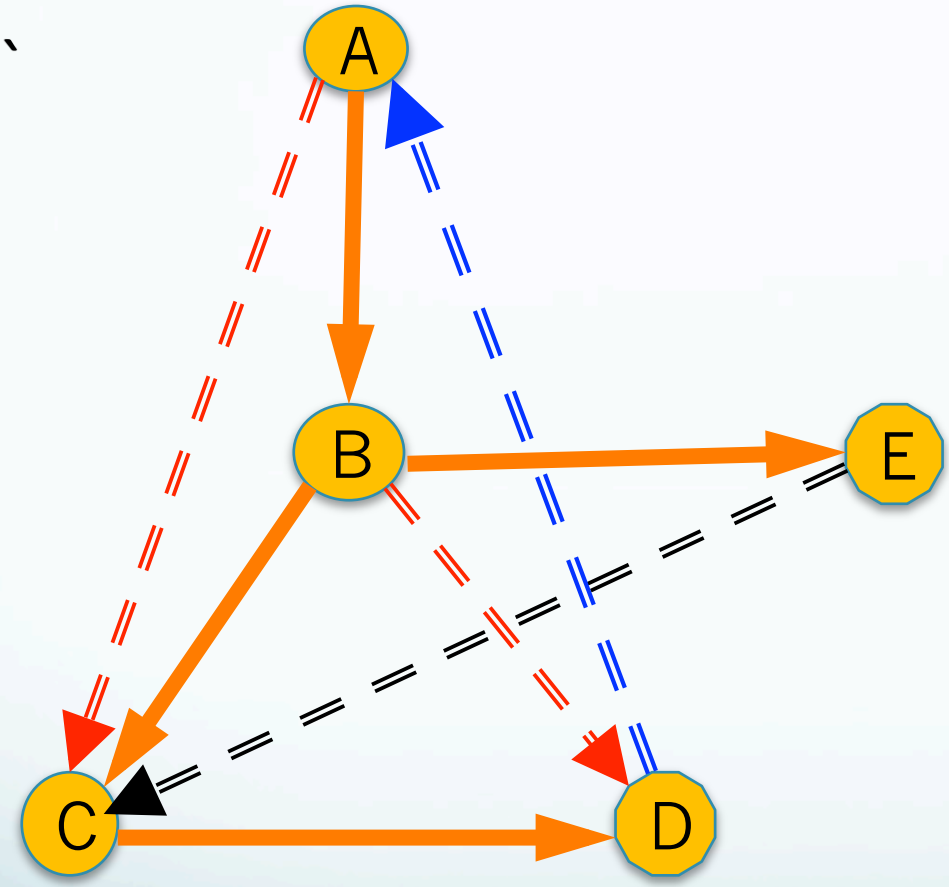


Oranges edges are *tree edges*.

Not used used edges:

- all would be *back-edges* if the graph was un-directed

DFS



Oranges edges are *tree edges*.

Not used edges:

- red are *forward edges*
- blue are *back-edges*
- other dashed are *cross edges*

Why don't we have forward and cross edges in undirected graph?