Indian Institute of Information Technology Allahabad

Data Structures

Depth First Search (DFS)

Dr. Shiv Ram Dubey
Assistant Professor
Department of Information Technology
Indian Institute of Information Technology, Allahabad

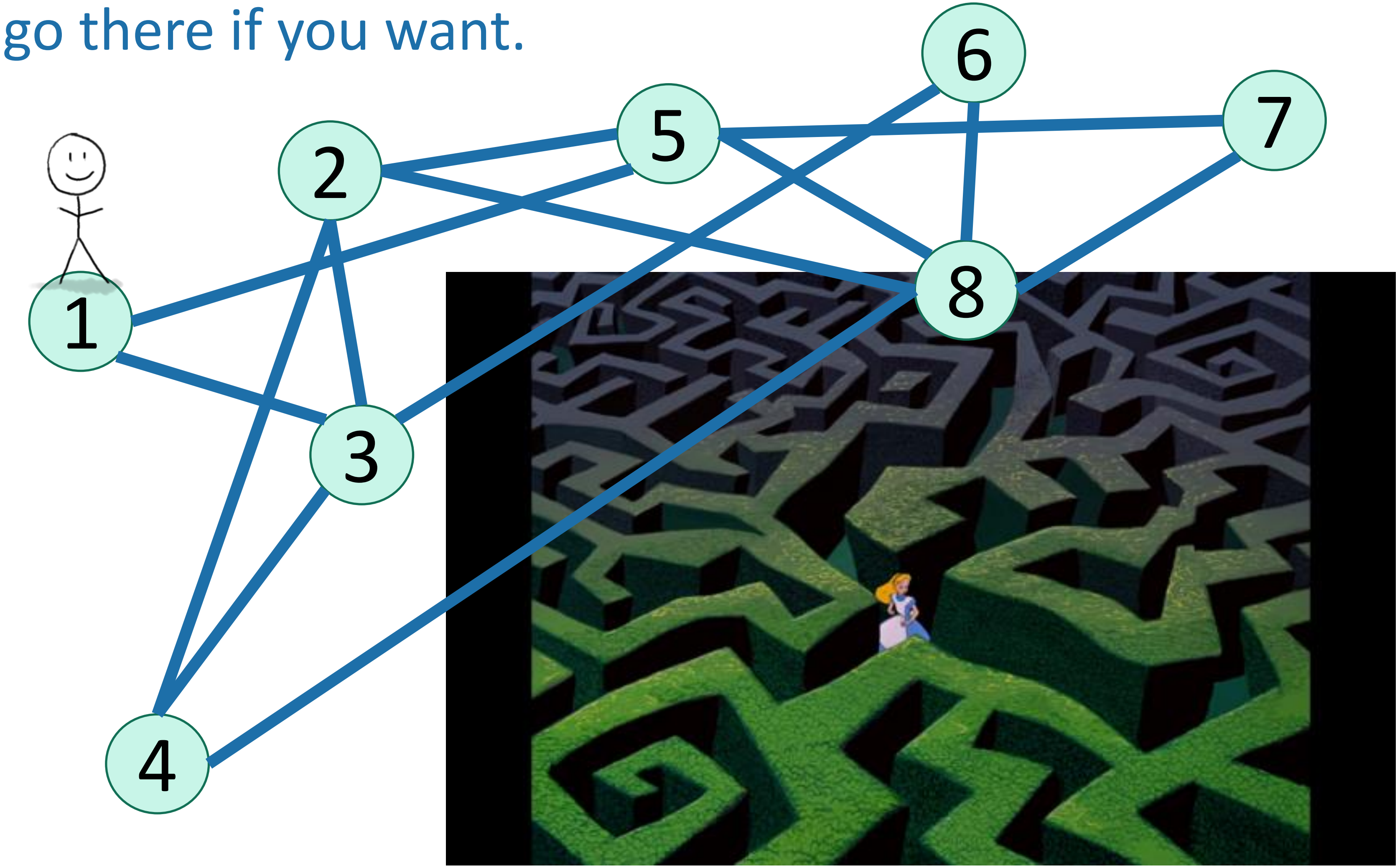
Email: srdubey@iiita.ac.in Web: https://profile.iiita.ac.in/srdubey/

DISCLAIMER

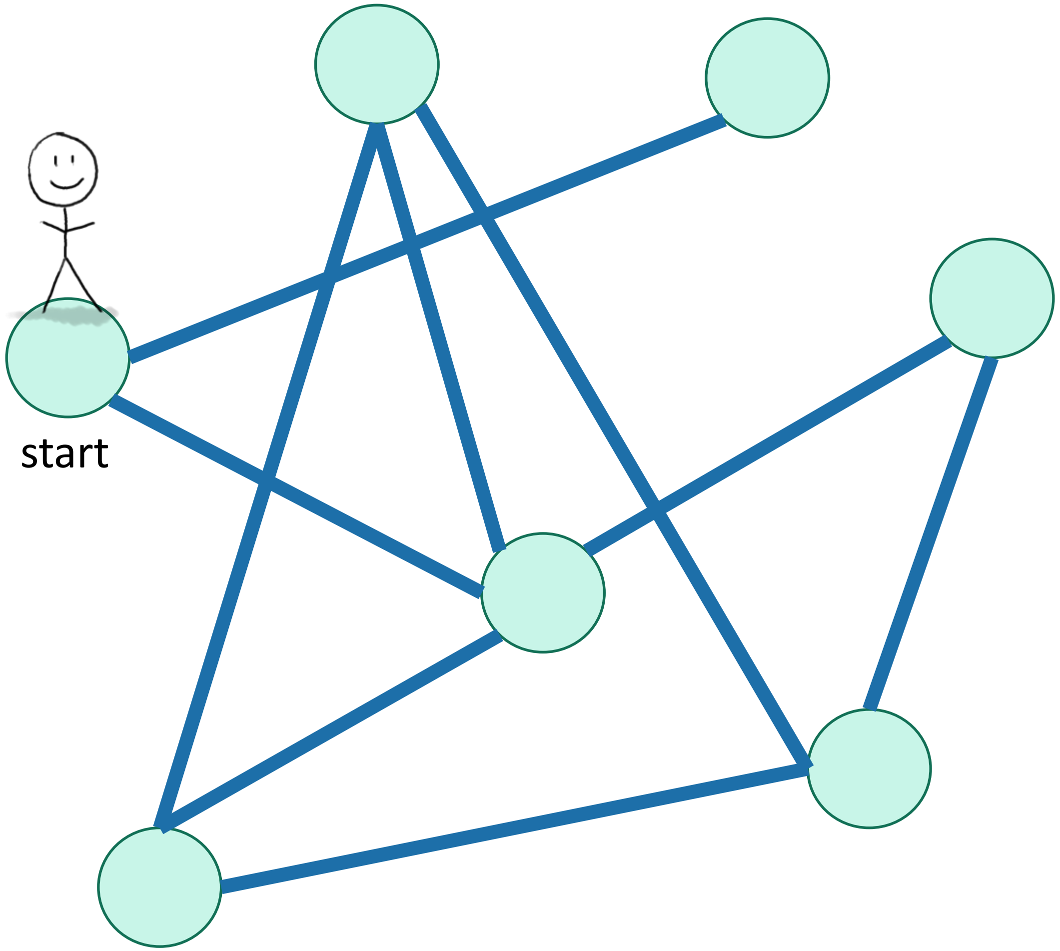
The content (text, image, and graphics) used in this slide are adopted from many sources for academic purposes. Broadly, the sources have been given due credit appropriately. However, there is a chance of missing out some original primary sources. The authors of this material do not claim any copyright of such material.

How do we explore a graph?

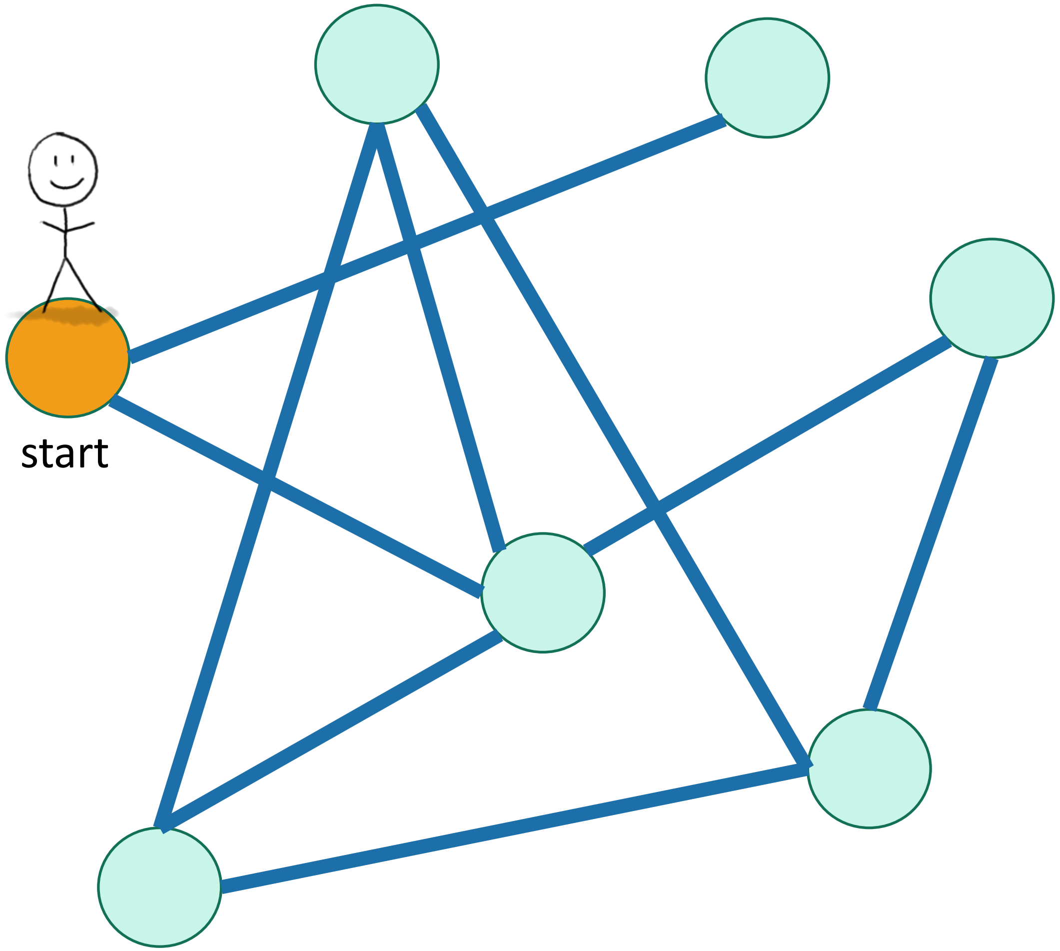
At each node, you can get a list of neighbors, and choose to go there if you want.

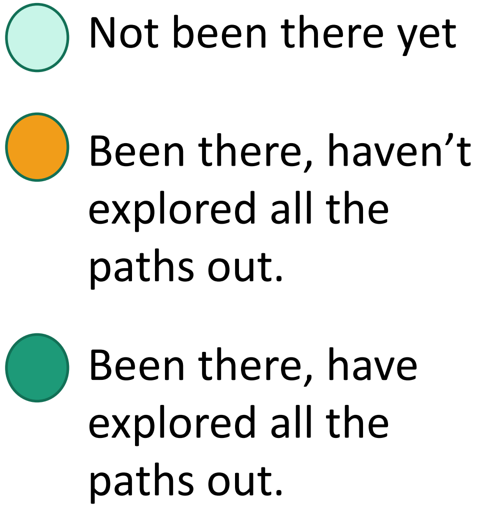


Depth First Search

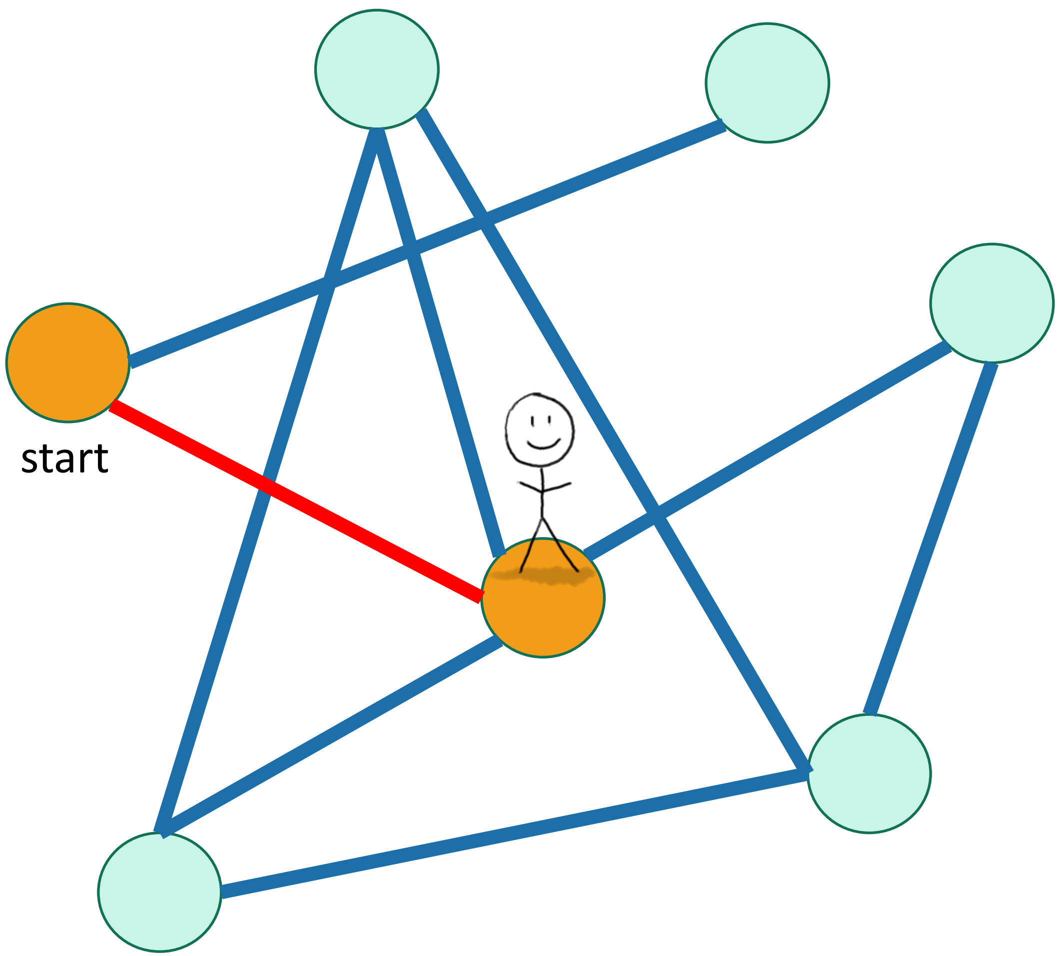


Depth First Search





Depth First Search

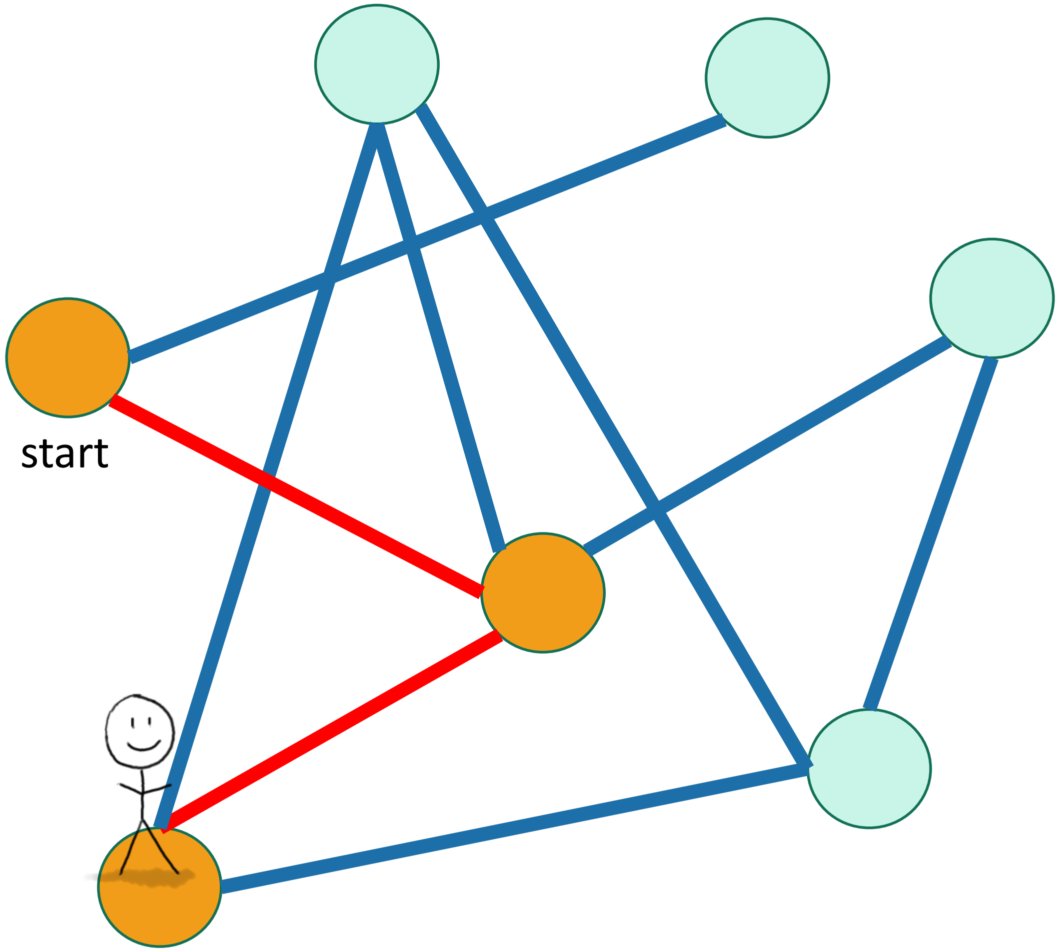


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

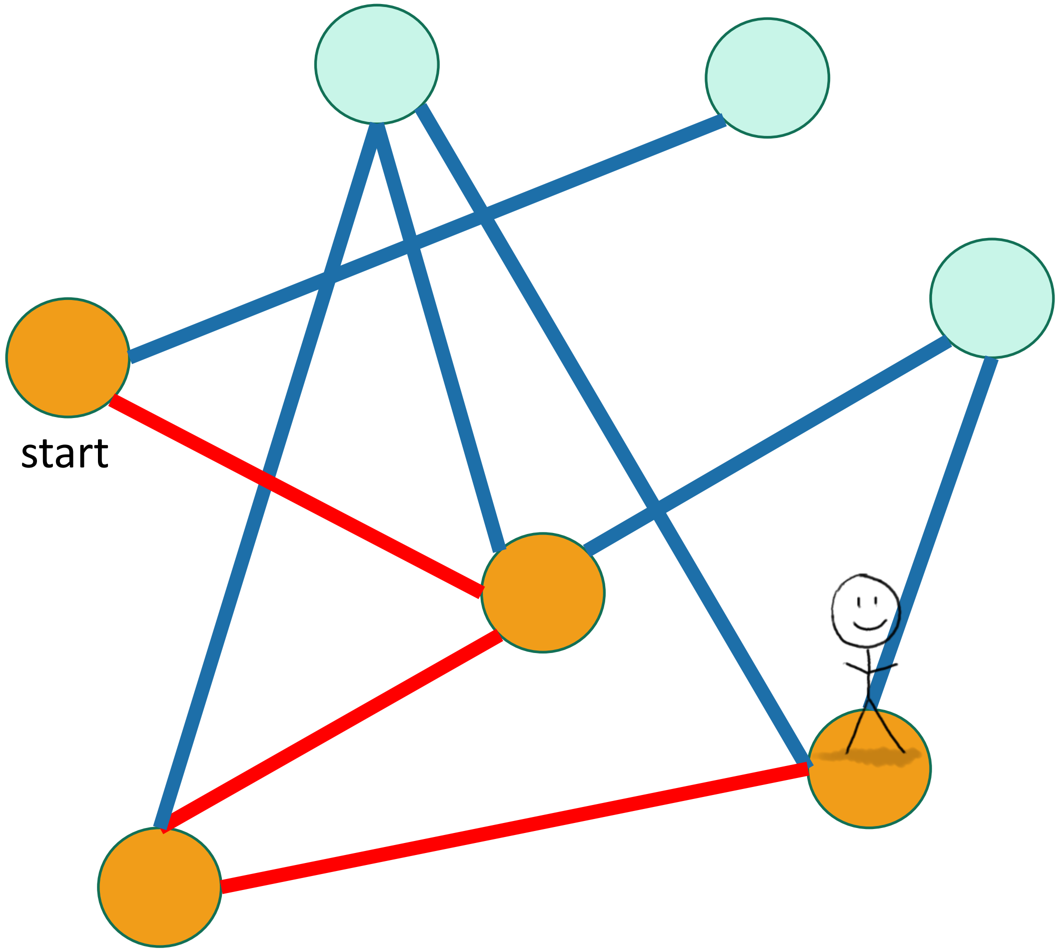


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

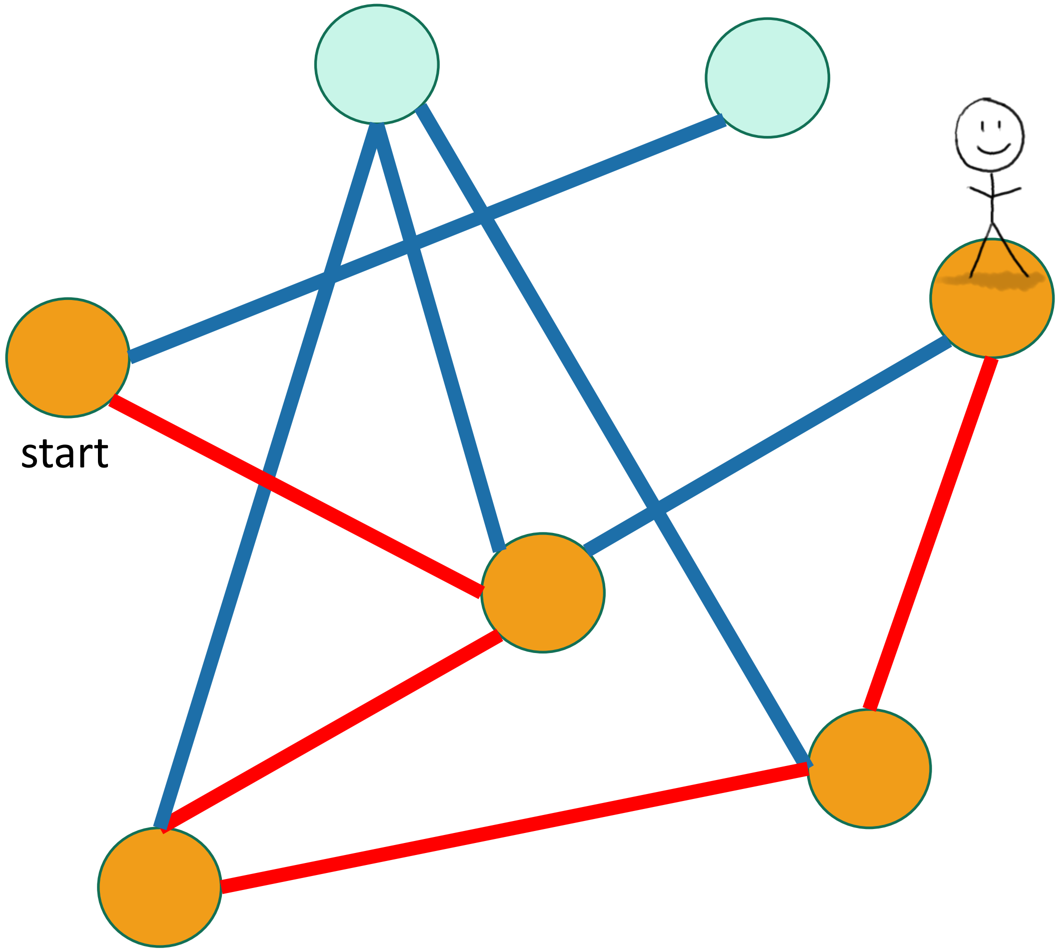


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

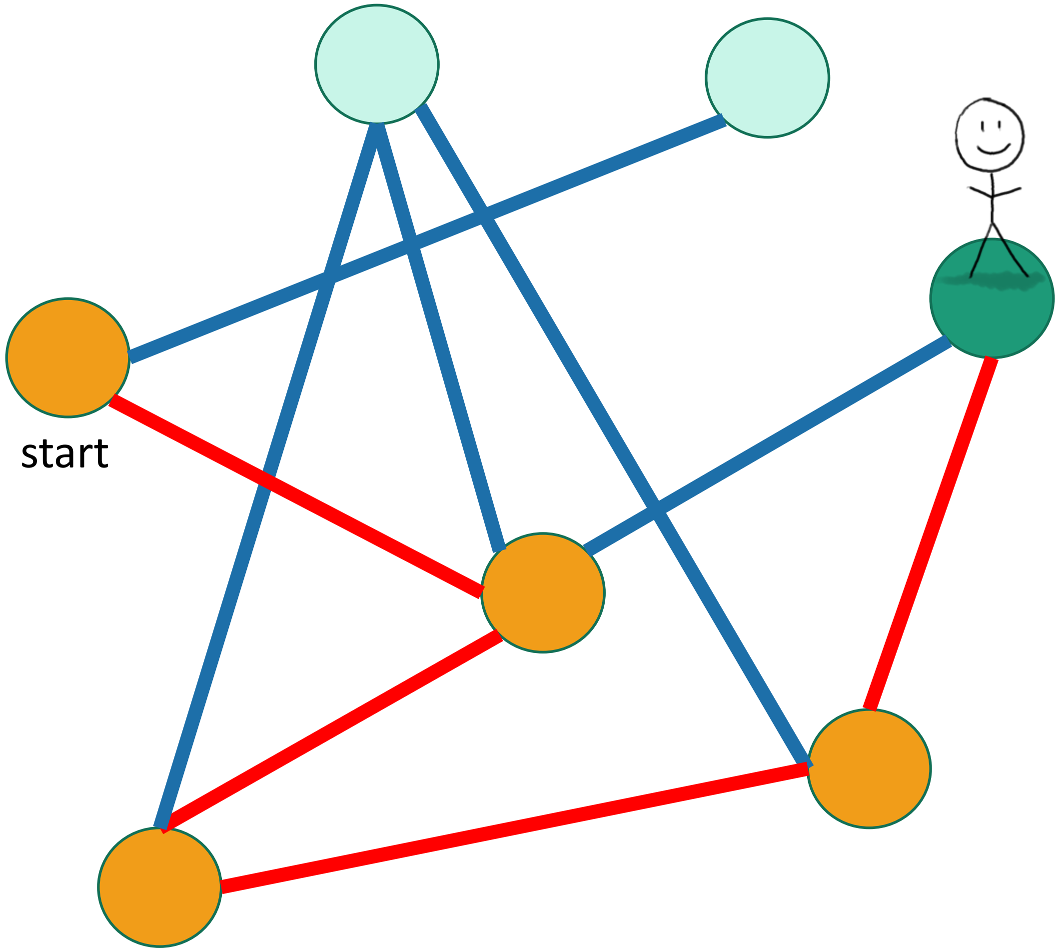


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

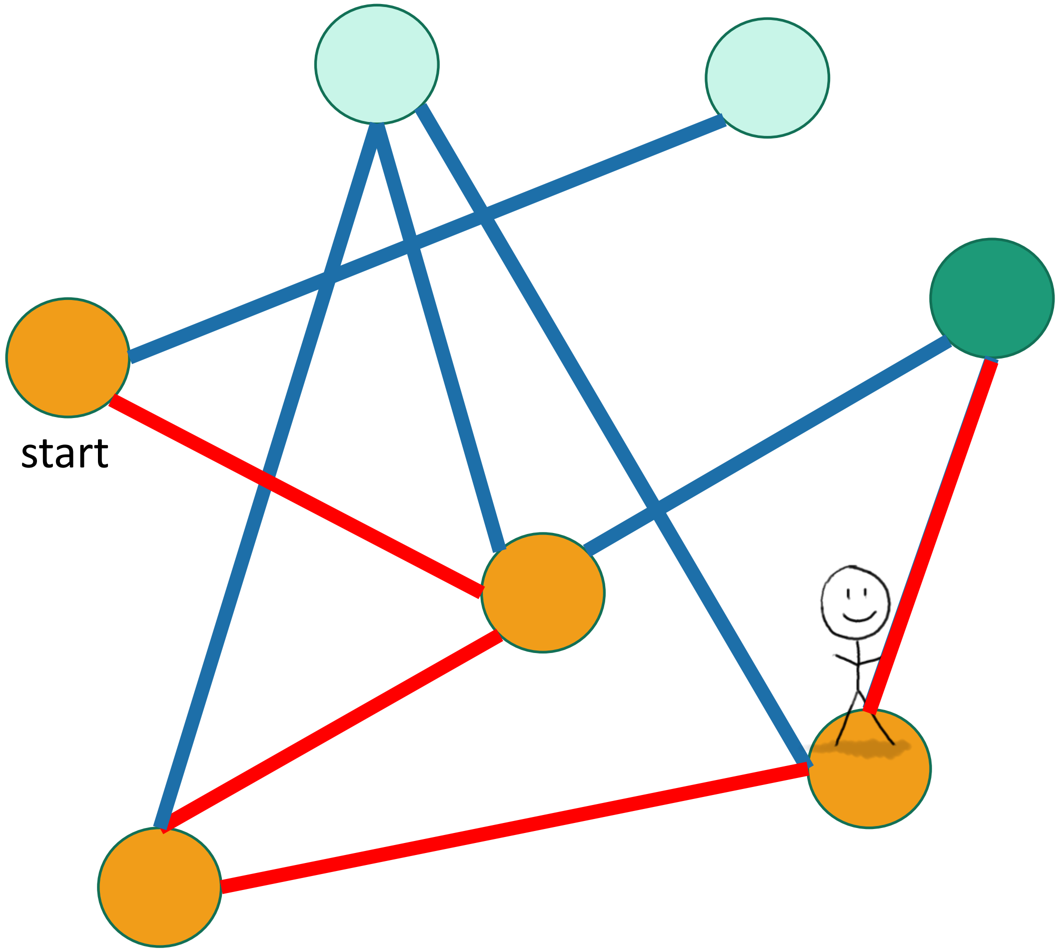


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

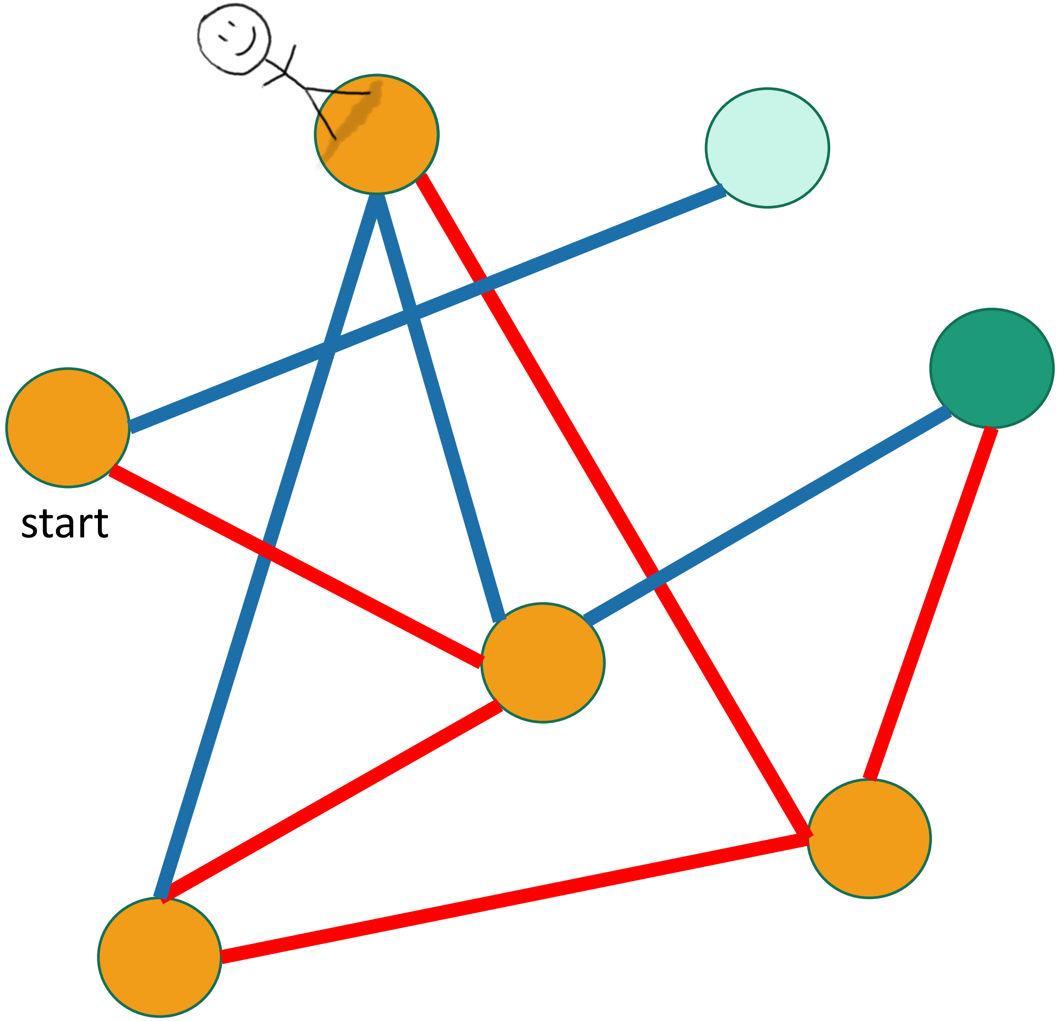


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

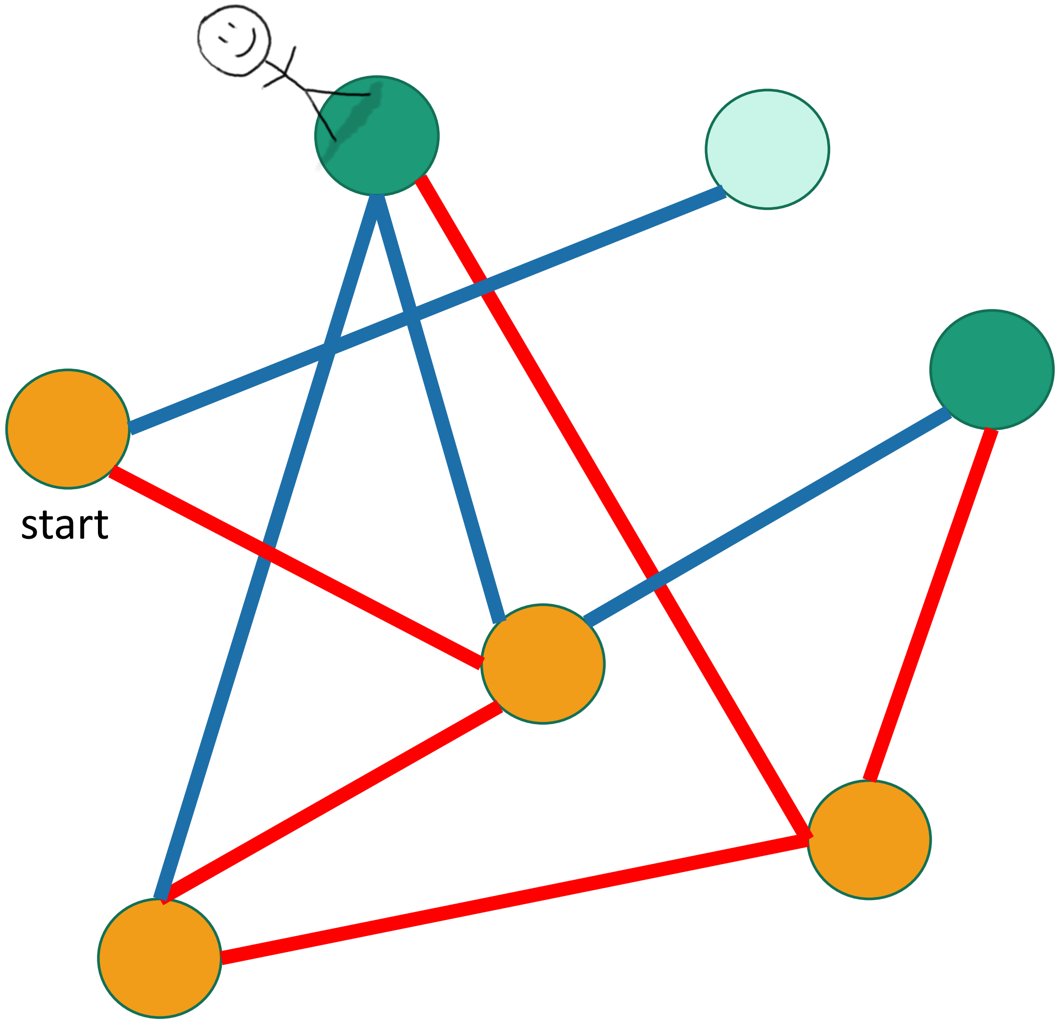


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

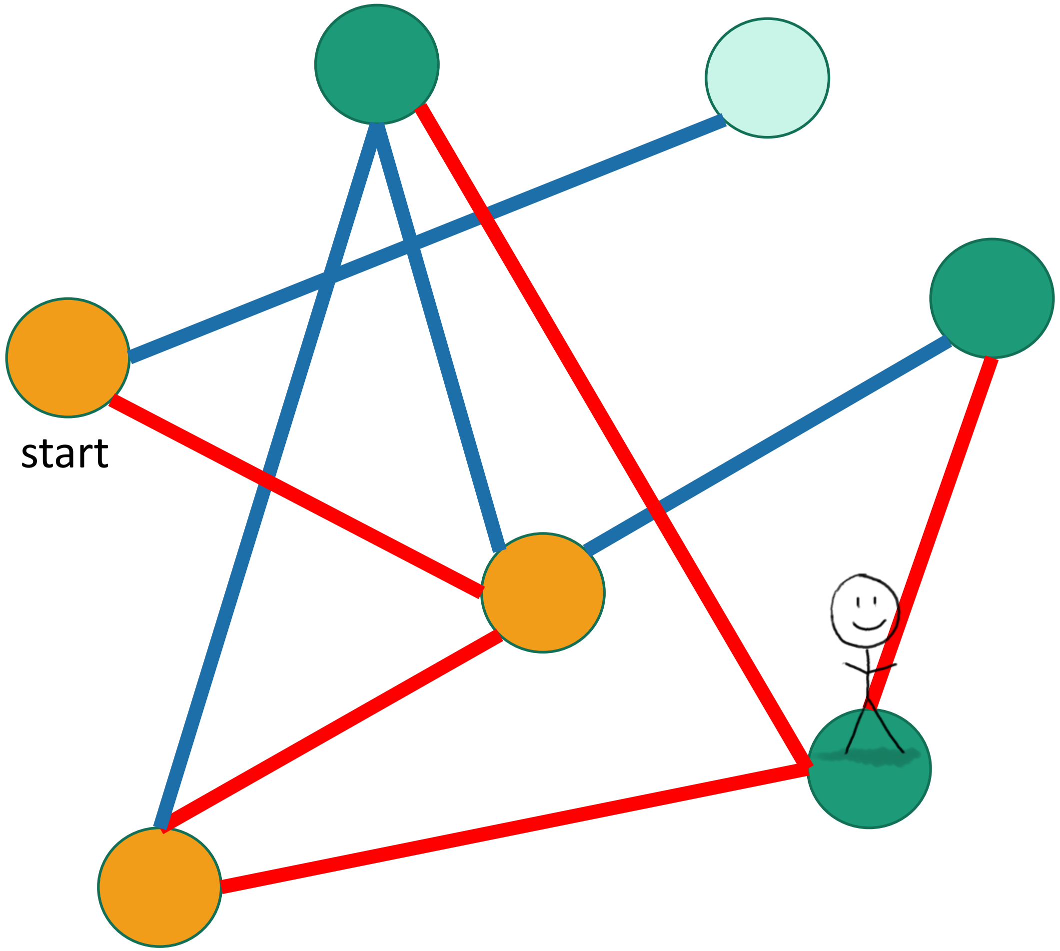


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

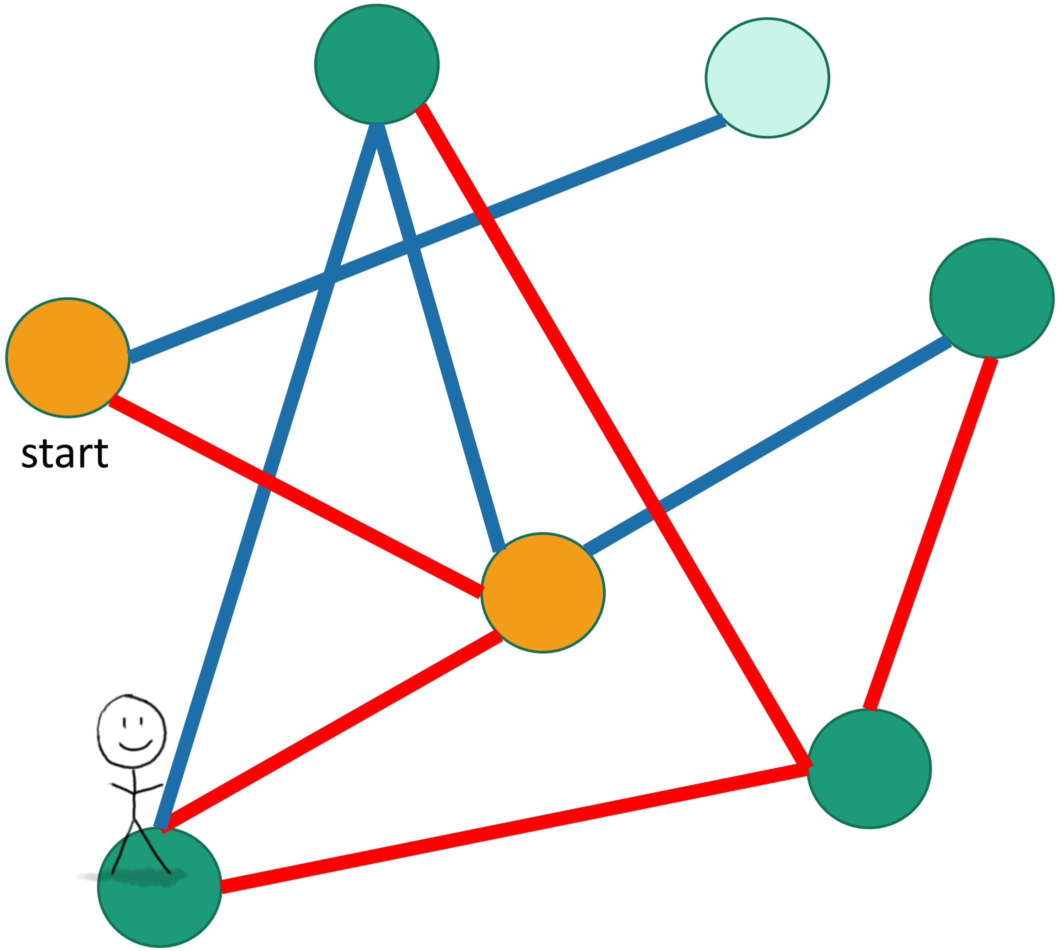


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

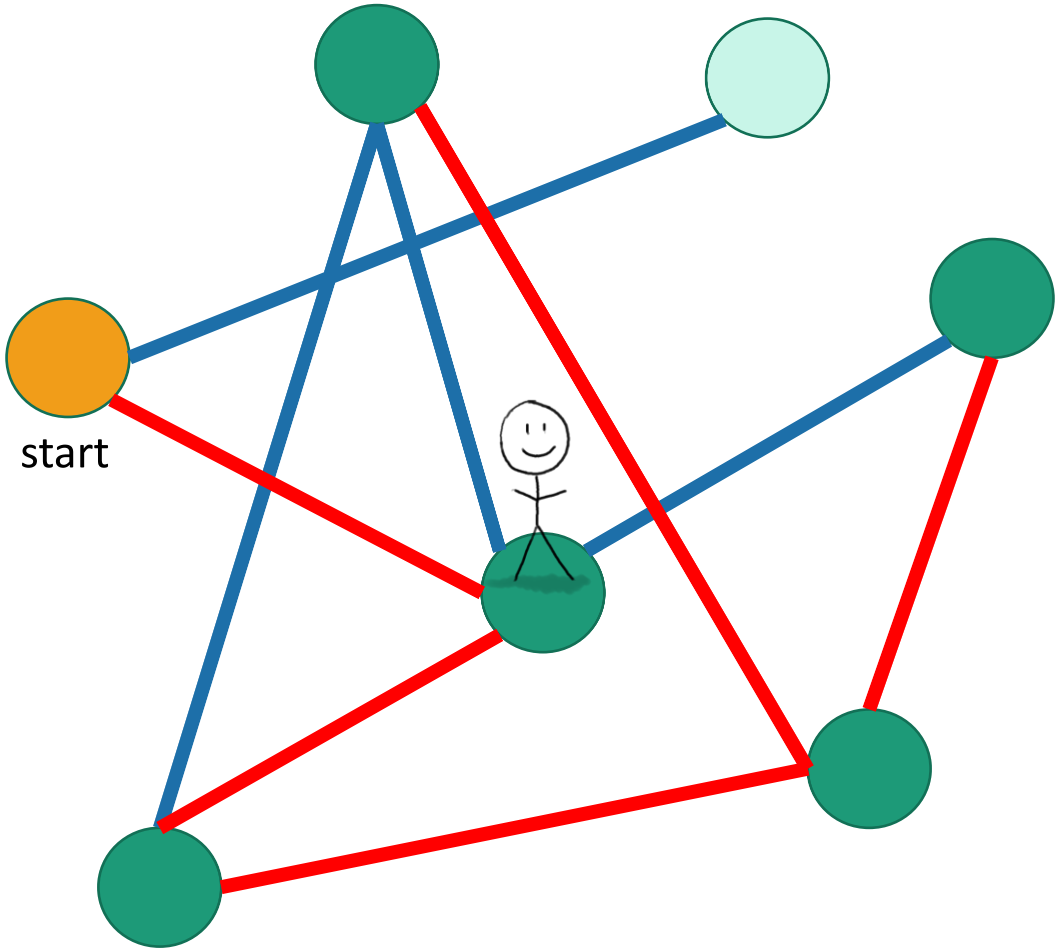


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

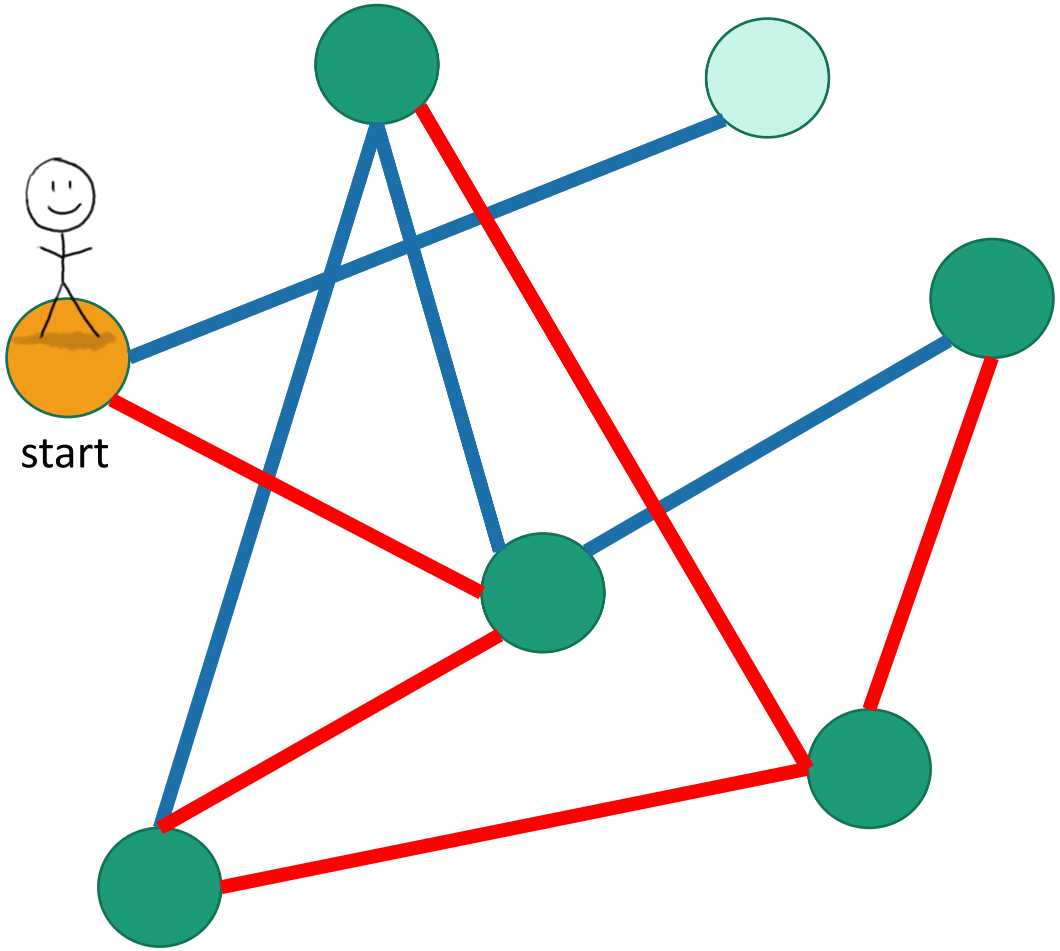


Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search

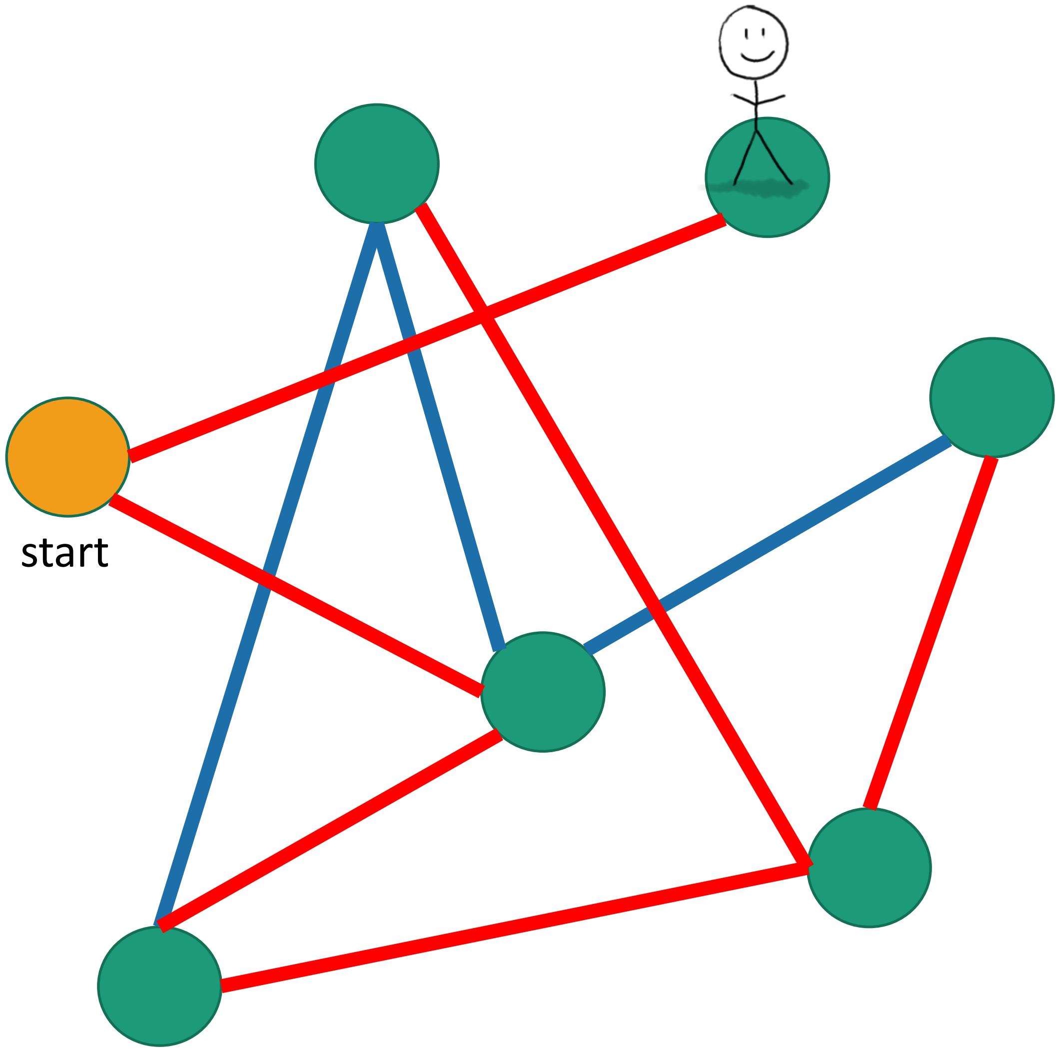


Not been there yet

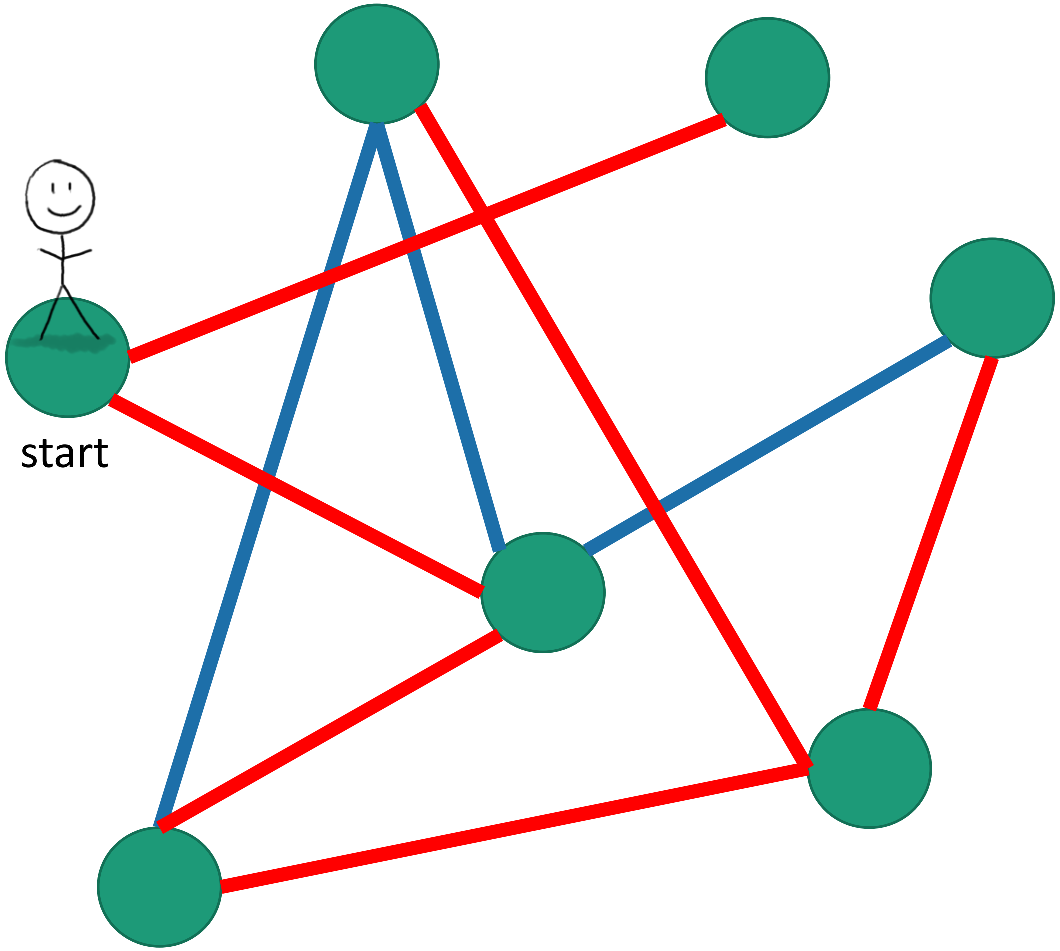
Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search



Depth First Search



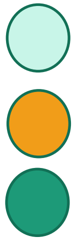
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Depth First Search
Pseudocode

- Each vertex keeps track of whether it is:
  - Unvisited
  - In progress
  - All done



Depth First Search

Pseudocode

- Each vertex keeps track of whether it is:
  - Unvisited
  - In progress
  - All done

- Each vertex will also keep track of:
  - The time we **first enter it**.
  - The time we finish with it and mark it **all done**.

Depth First Search

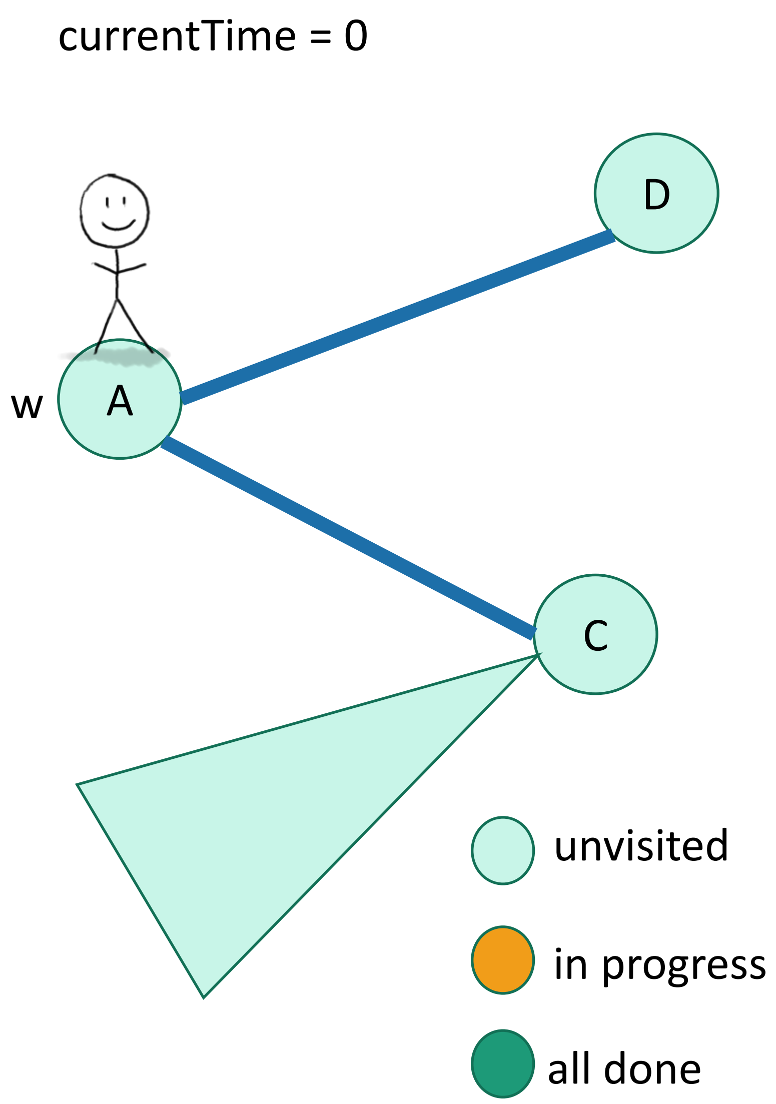
Pseudocode

- Each vertex keeps track of whether it is:
  - Unvisited
  - In progress
  - All done

- Each vertex will also keep track of:
  - The time we **first enter it**.
  - The time we finish with it and mark it **all done**.

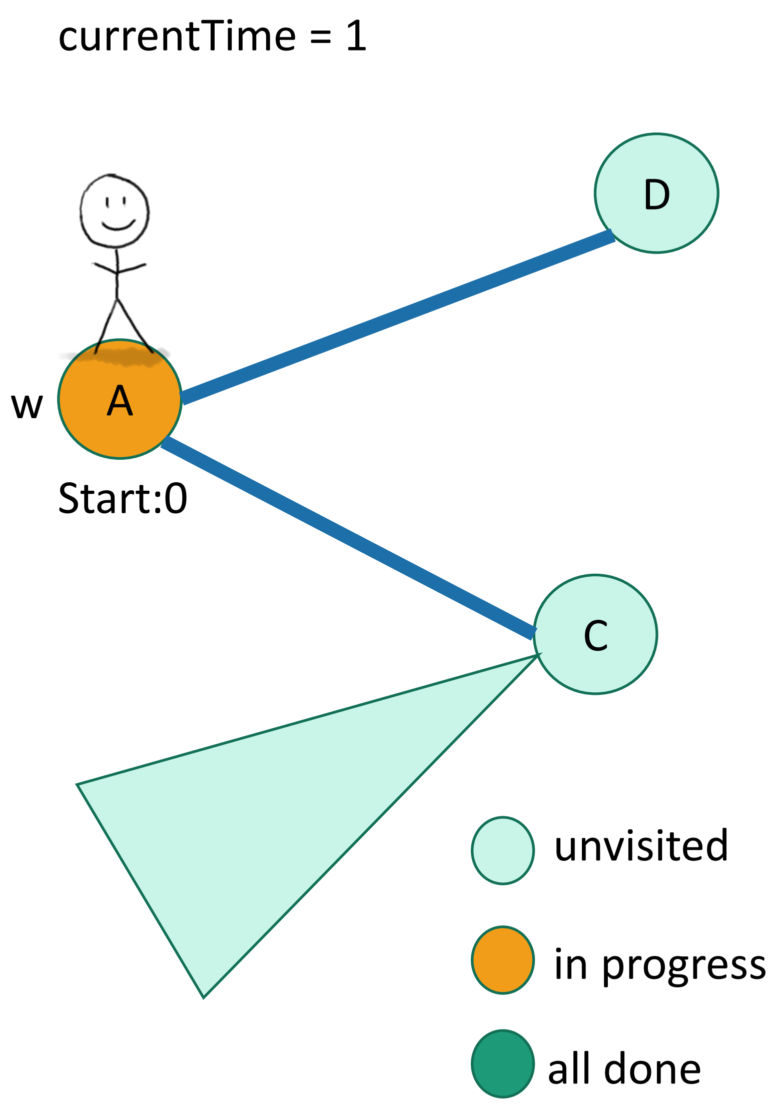
You might have seen other ways to implement DFS than what we are about to go through. This way has more bookkeeping – the bookkeeping will be useful later!

Depth First Search



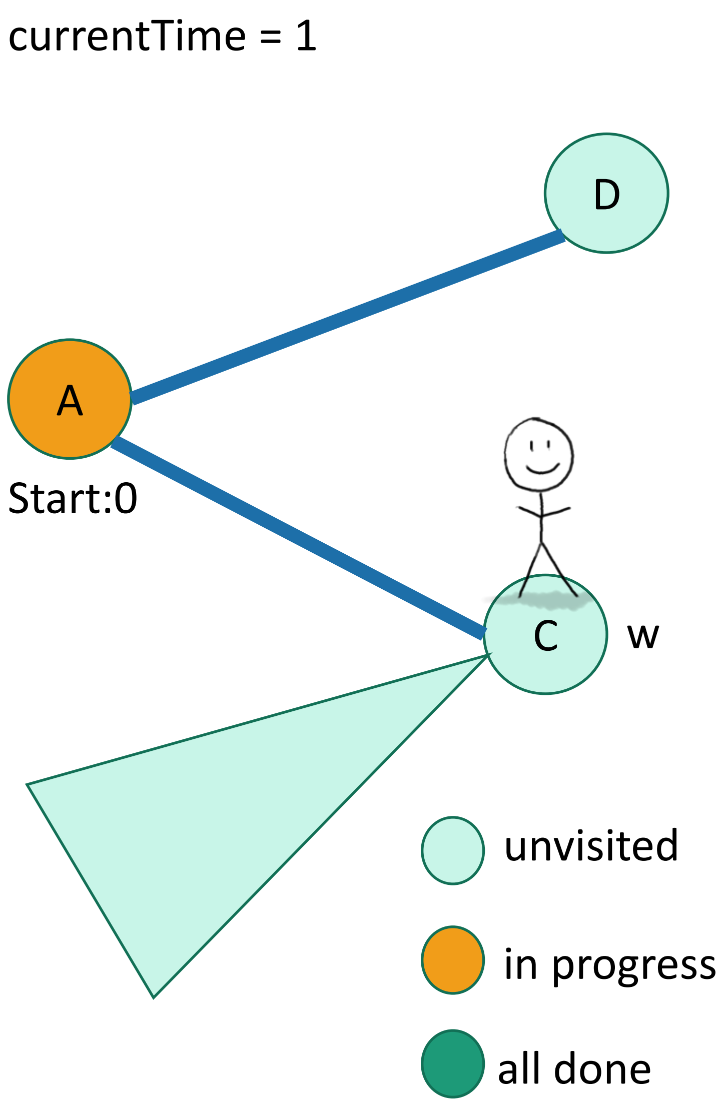
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

Depth First Search



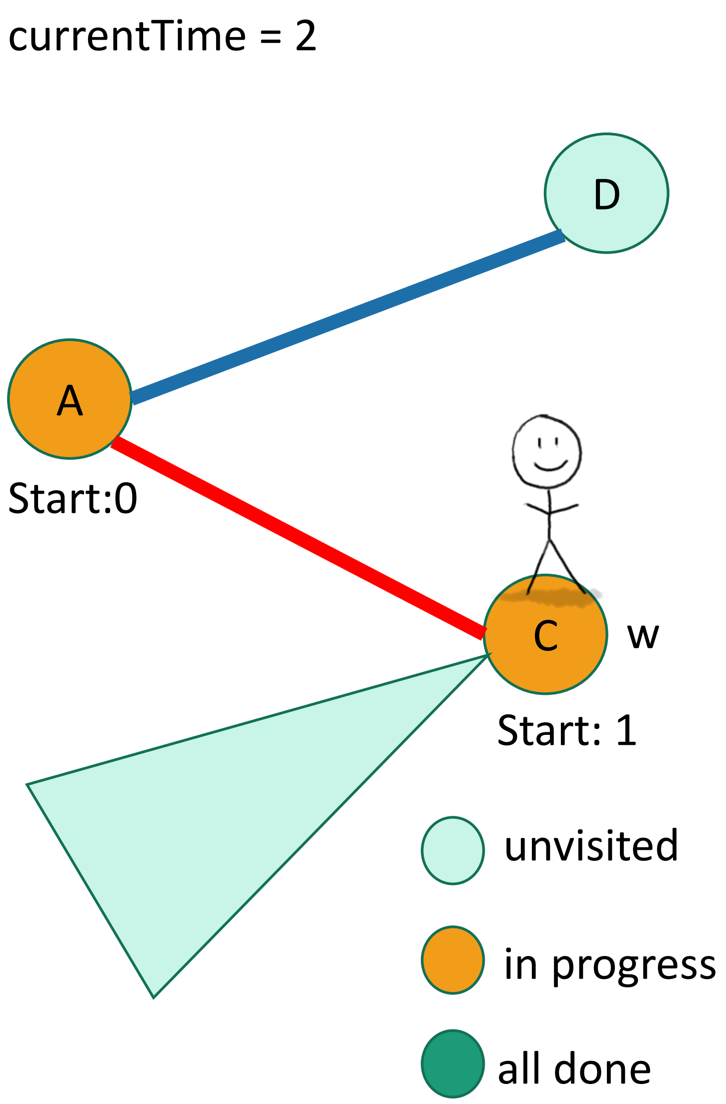
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

Depth First Search



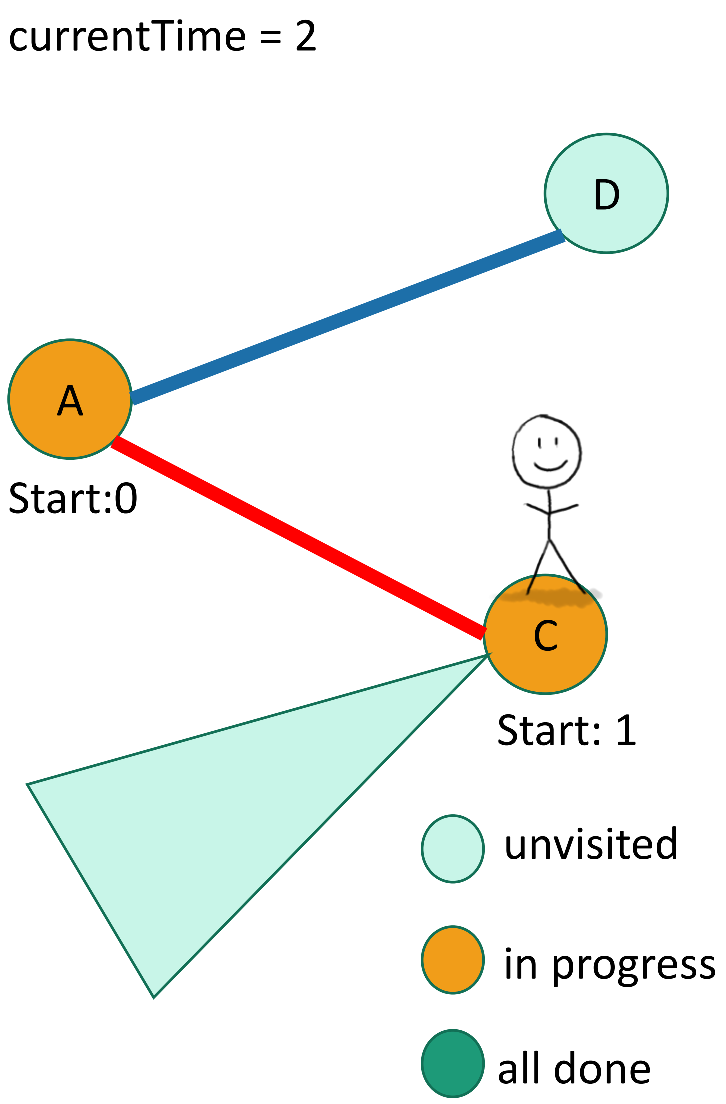
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

Depth First Search



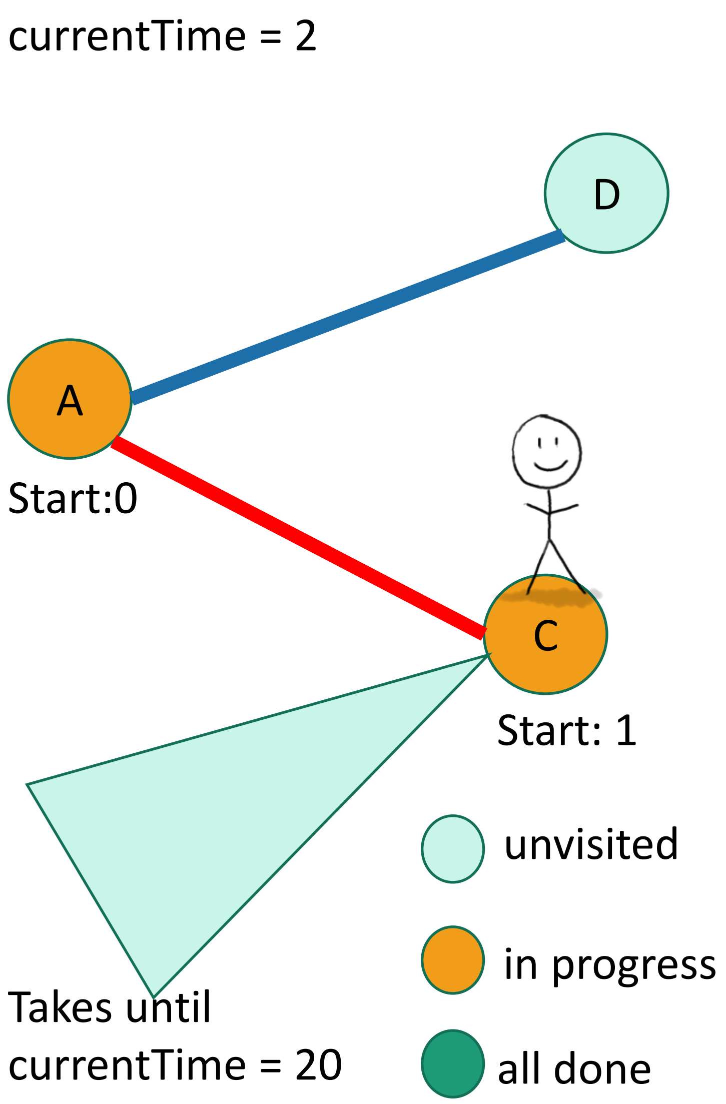
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

Depth First Search



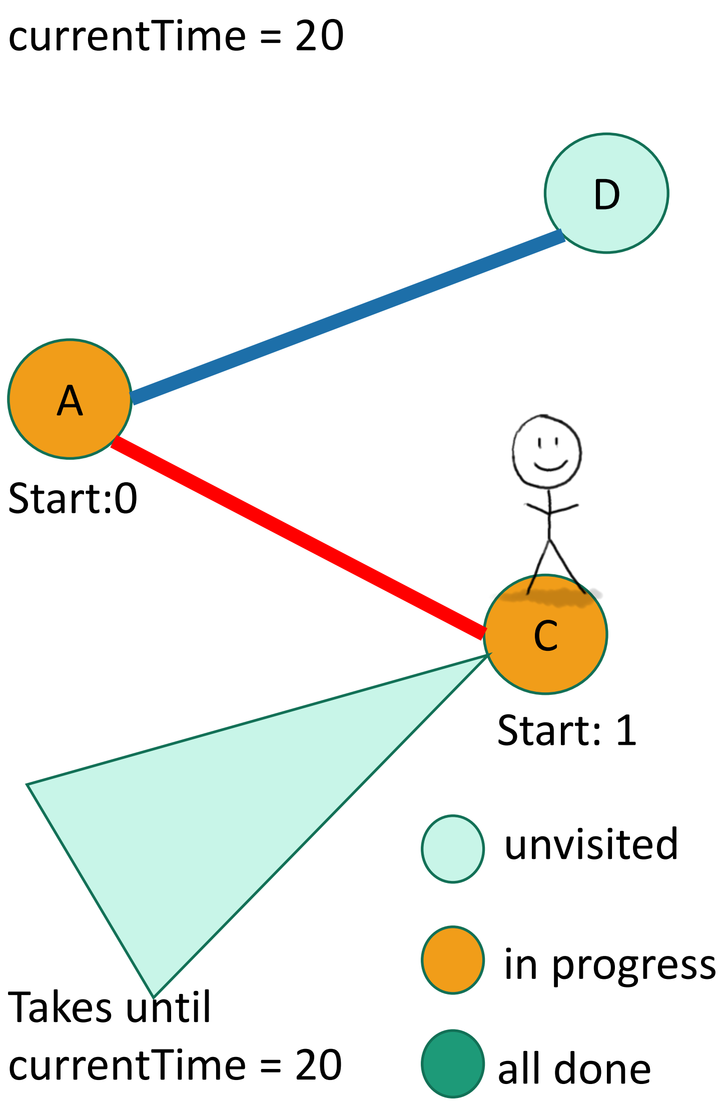
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

Depth First Search



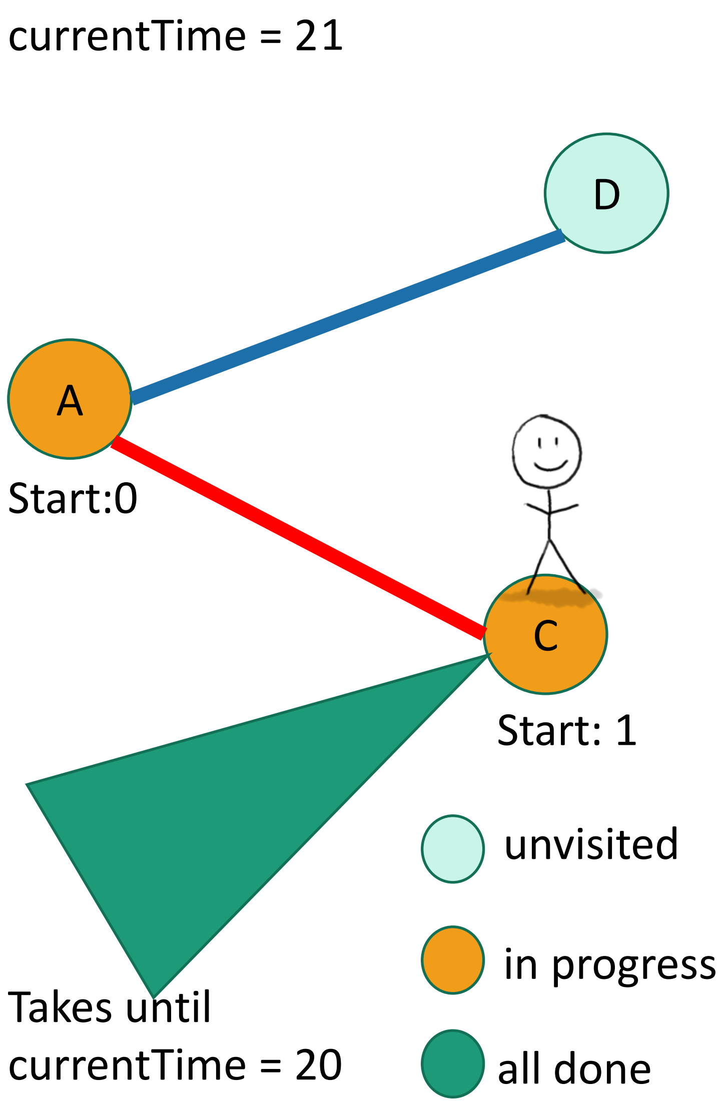
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

Depth First Search



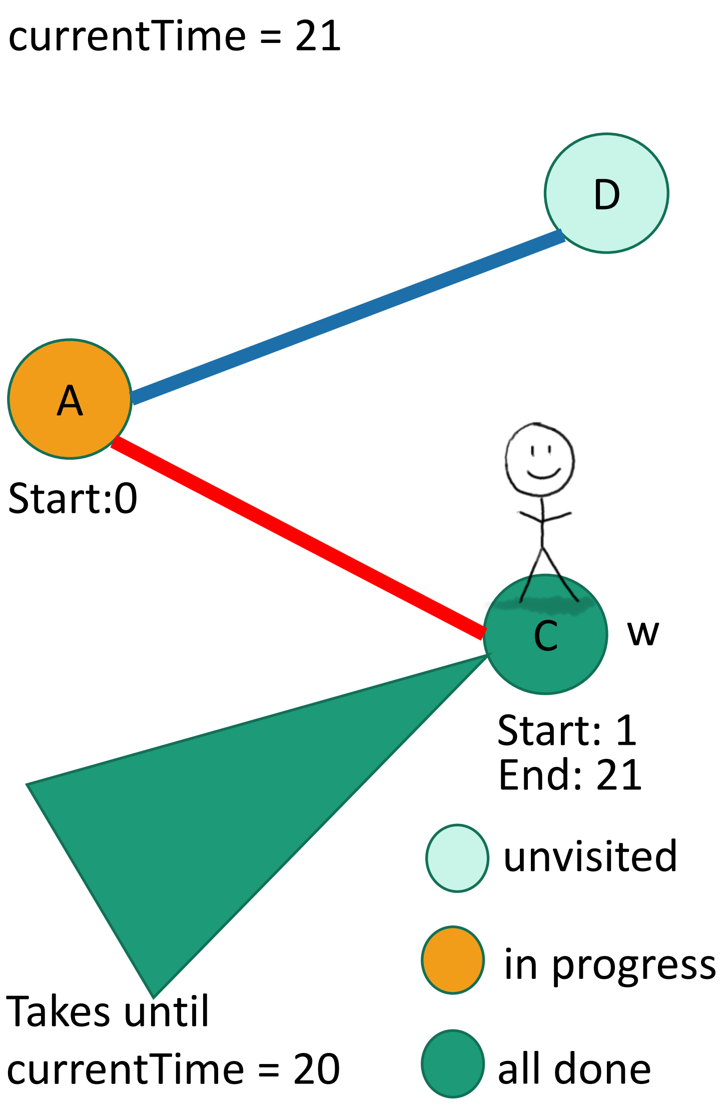
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as in progress.
  - **for** v in w.neighbors:
    - **if** v is unvisited:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

Depth First Search



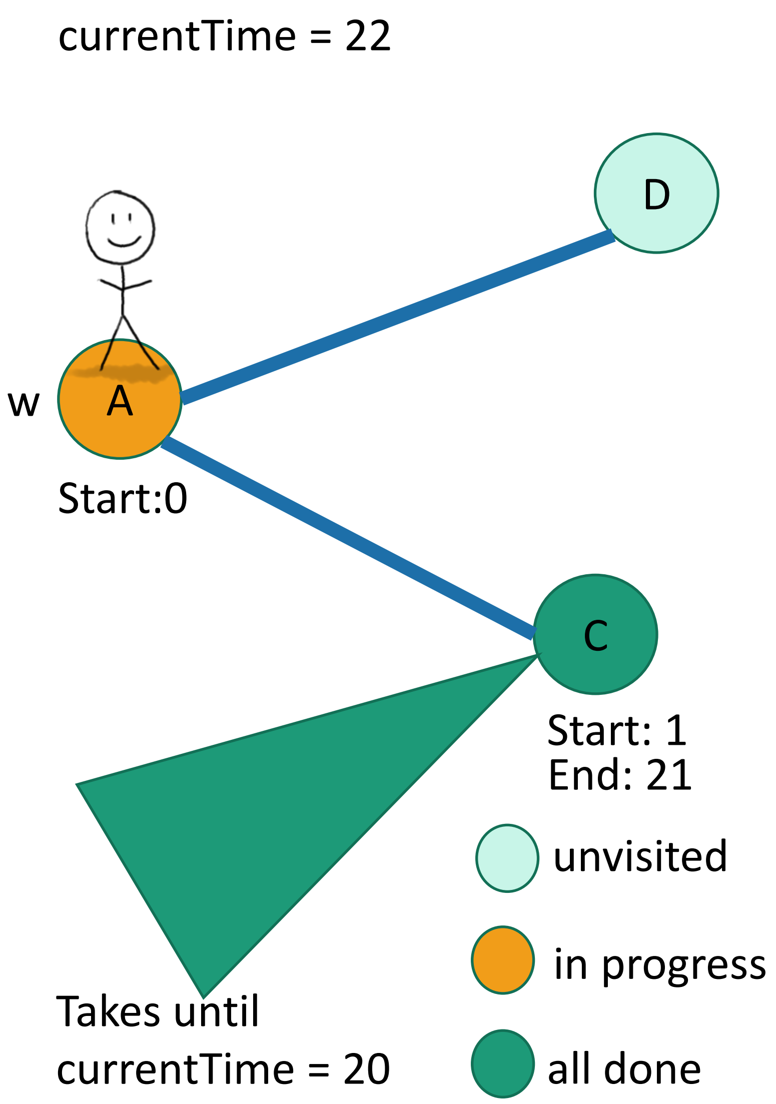
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

Depth First Search



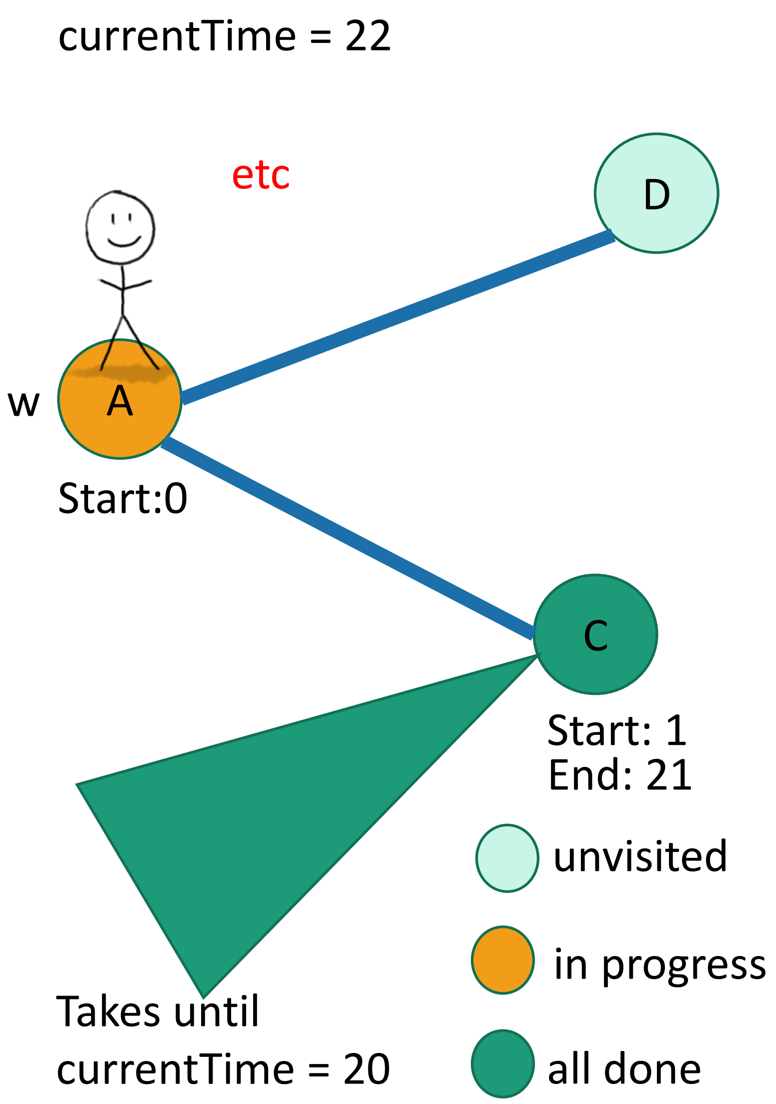
- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

Depth First Search



- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

Depth First Search

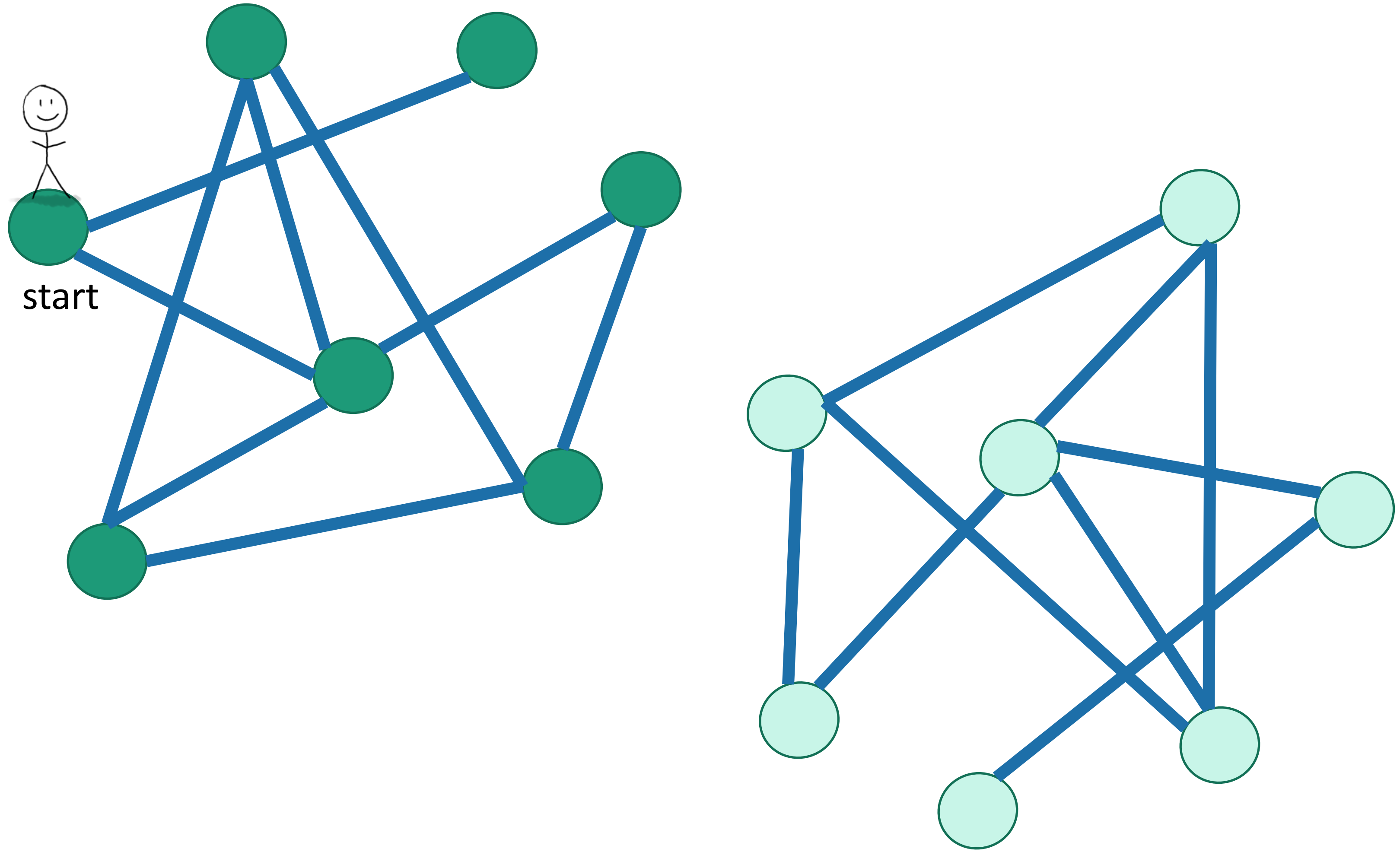


- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as **in progress**.
  - **for** v in w.neighbors:
    - **if** v is **unvisited**:
      - currentTime
        = **DFS**(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as **all done**
  - **return** currentTime

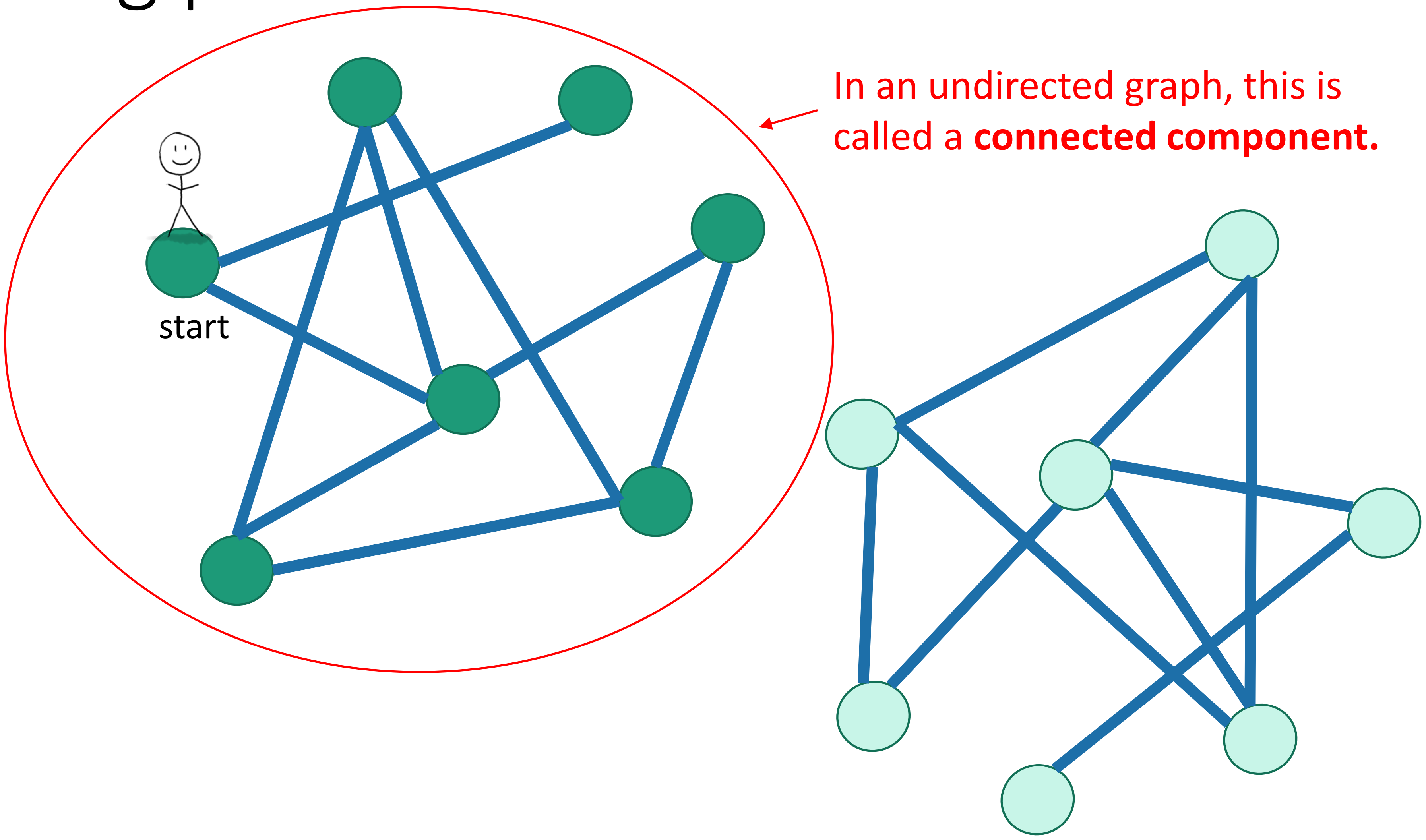
Fun exercise

- Write pseudocode for an iterative version of DFS.

DFS finds all the nodes reachable from the starting point

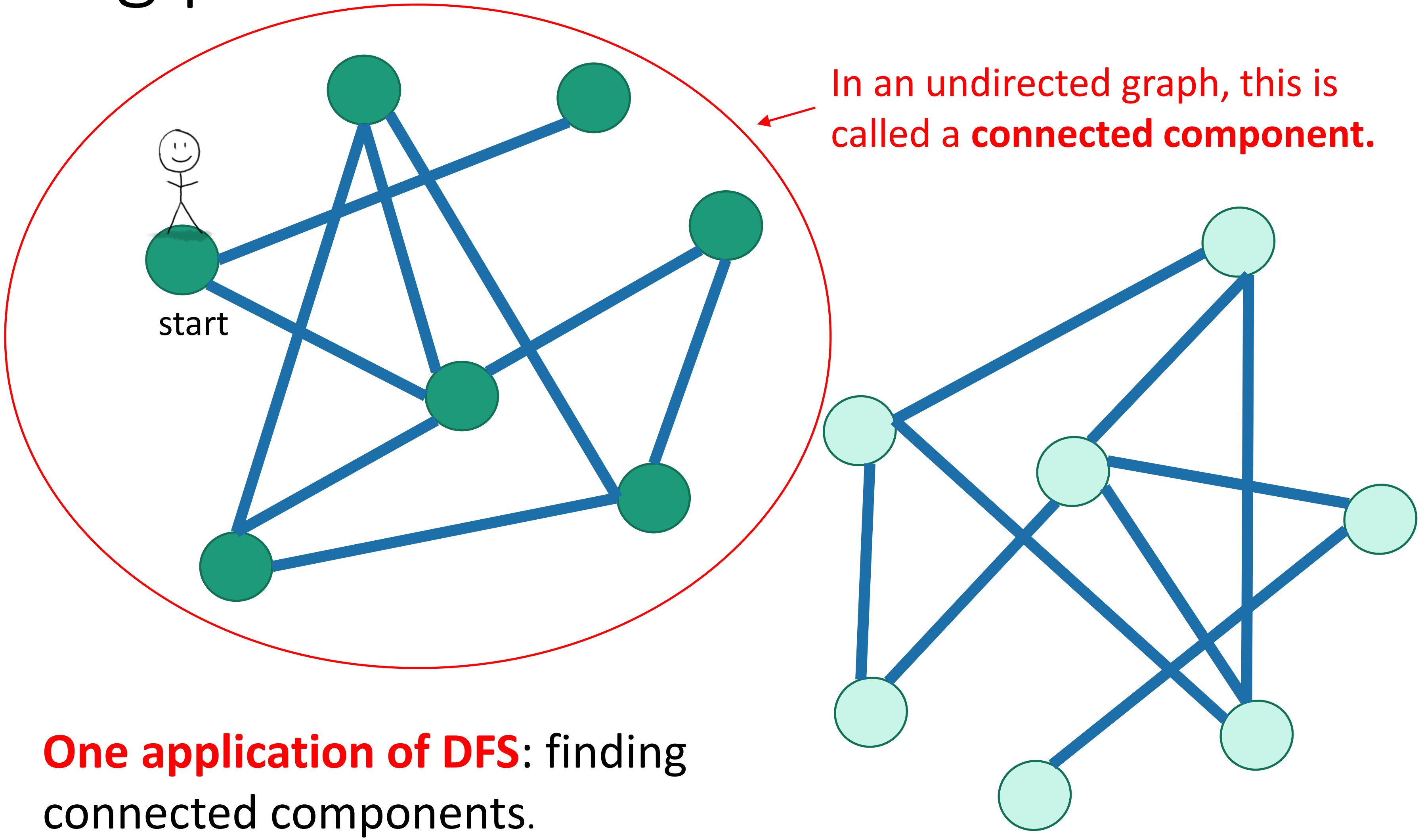


DFS finds all the nodes reachable from the starting point



In an undirected graph, this is called a **connected component.**

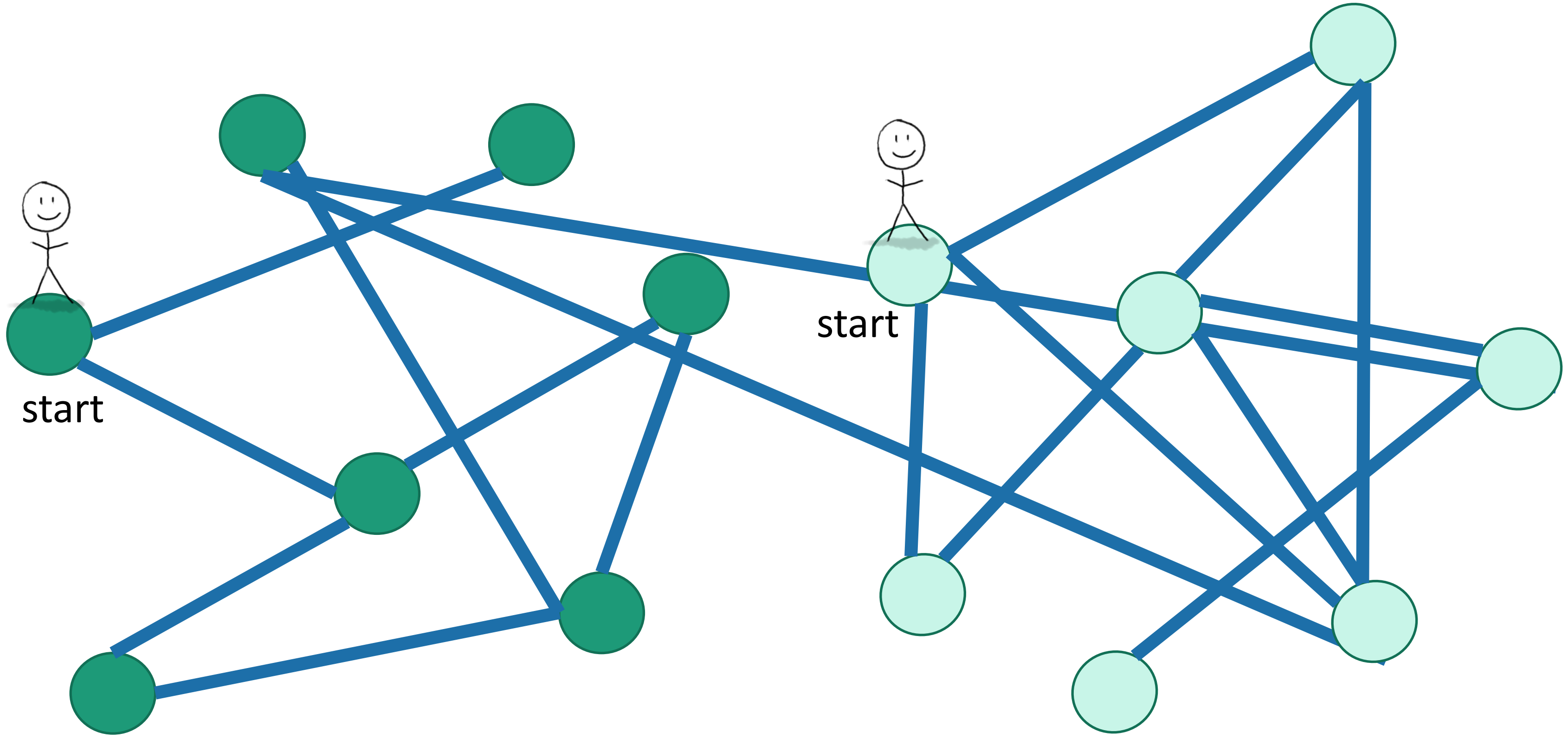
DFS finds all the nodes reachable from the starting point



One application of DFS: finding connected components.

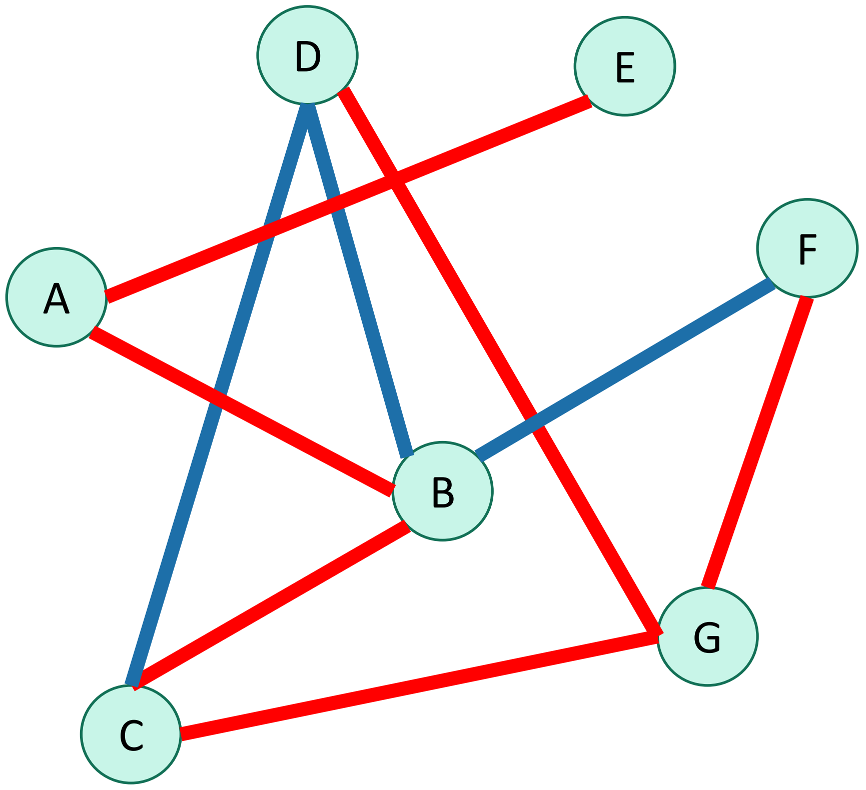
To explore the whole graph

- Do it repeatedly!



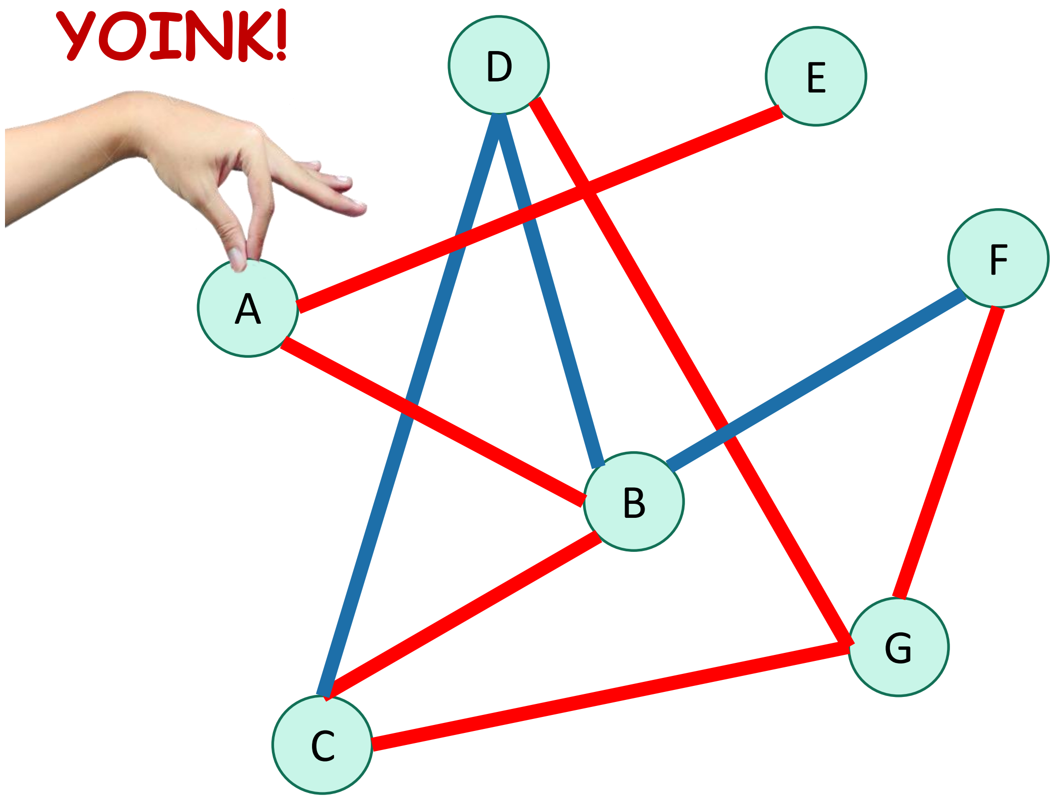
Why is it called depth-first?

- We are implicitly building a tree:



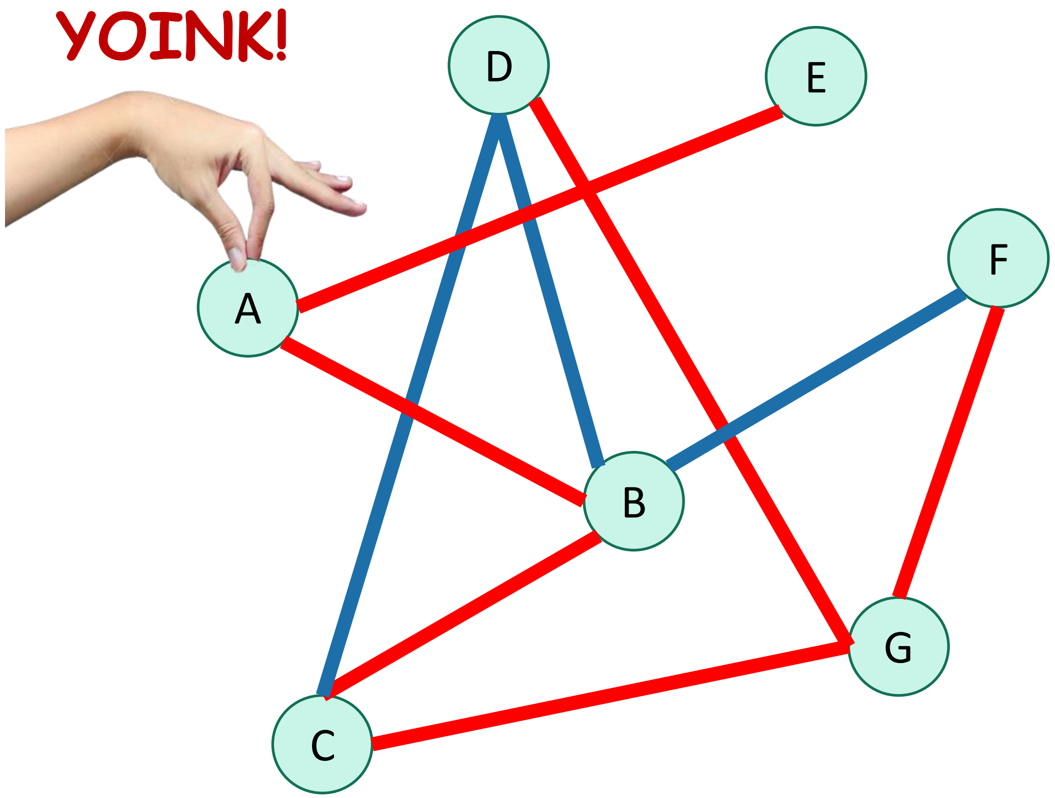
Why is it called depth-first?

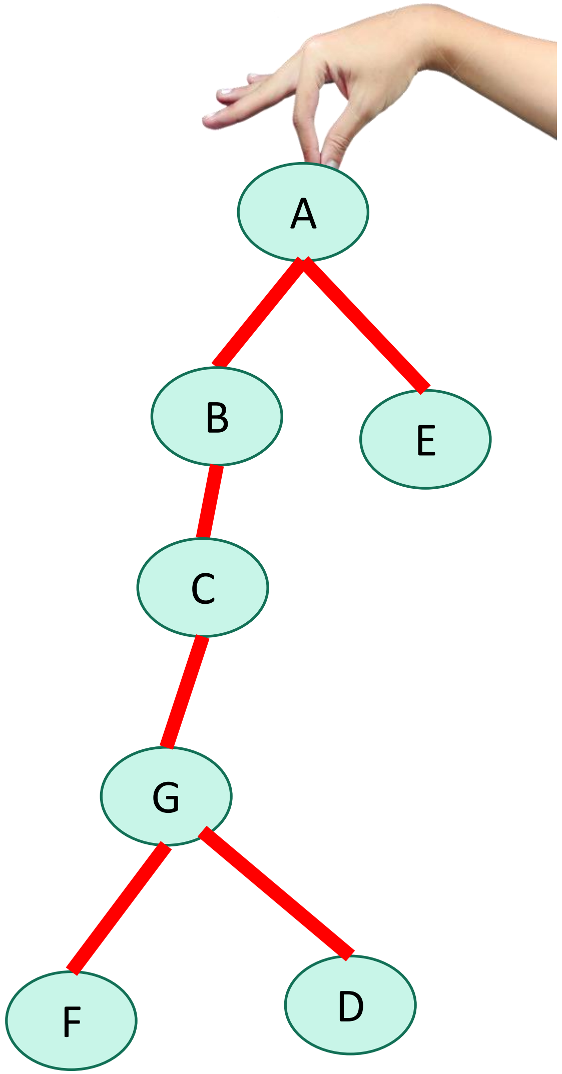
- We are implicitly building a tree:



Why is it called depth-first?

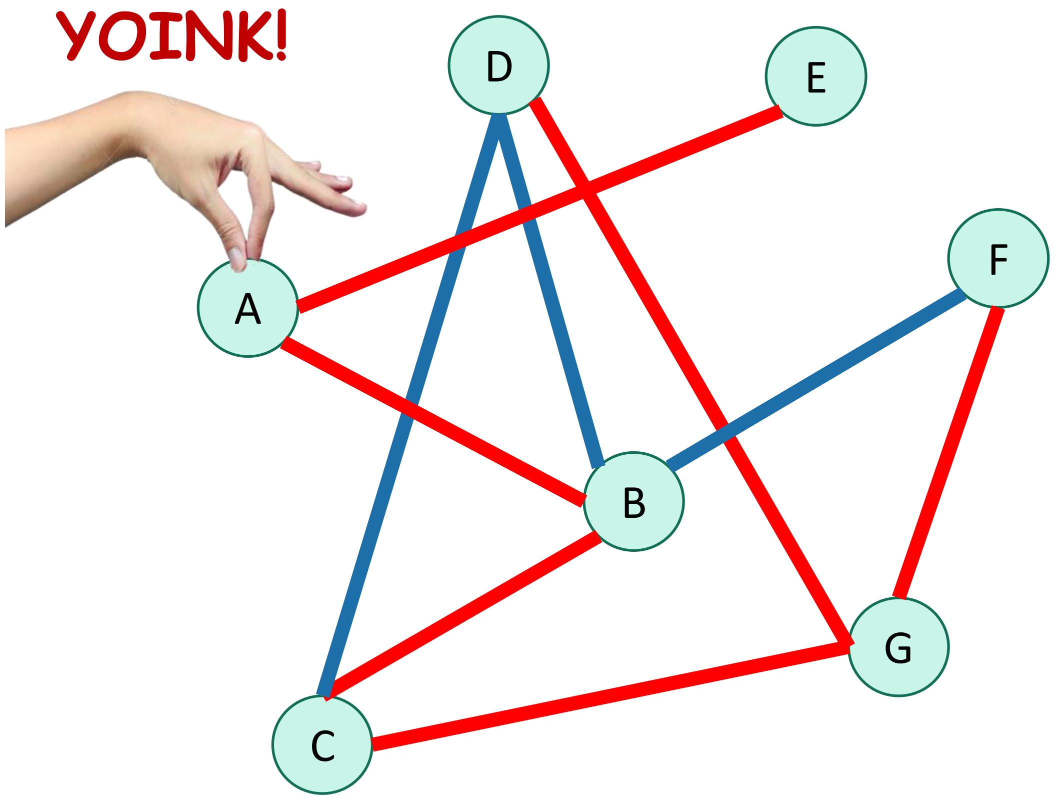
- We are implicitly building a tree:

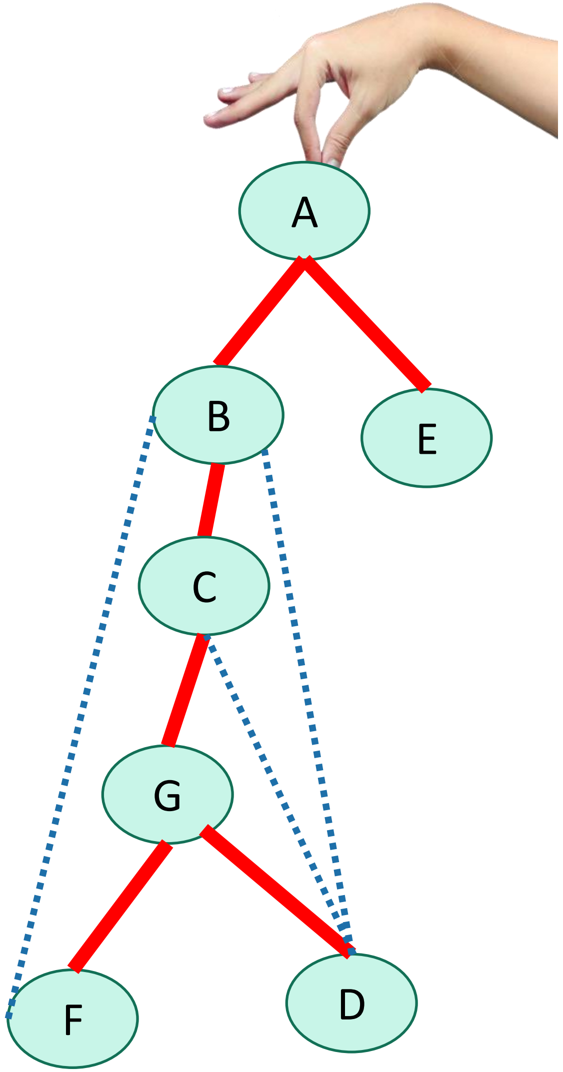




Why is it called depth-first?

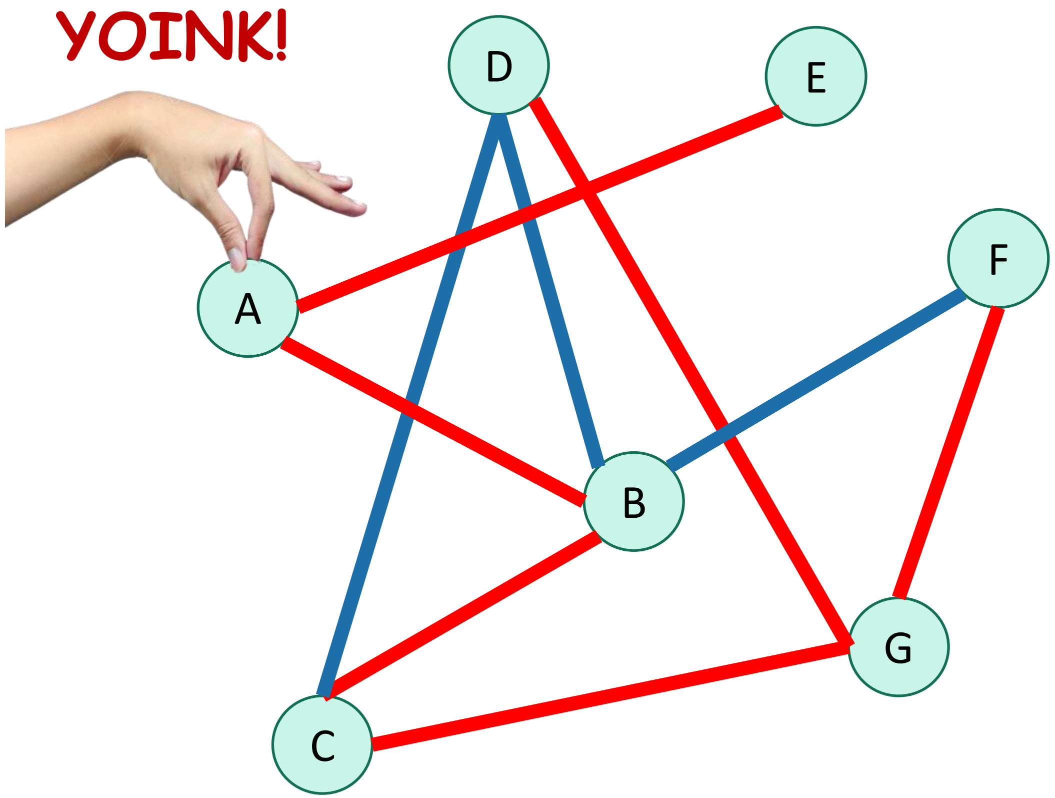
- We are implicitly building a tree:

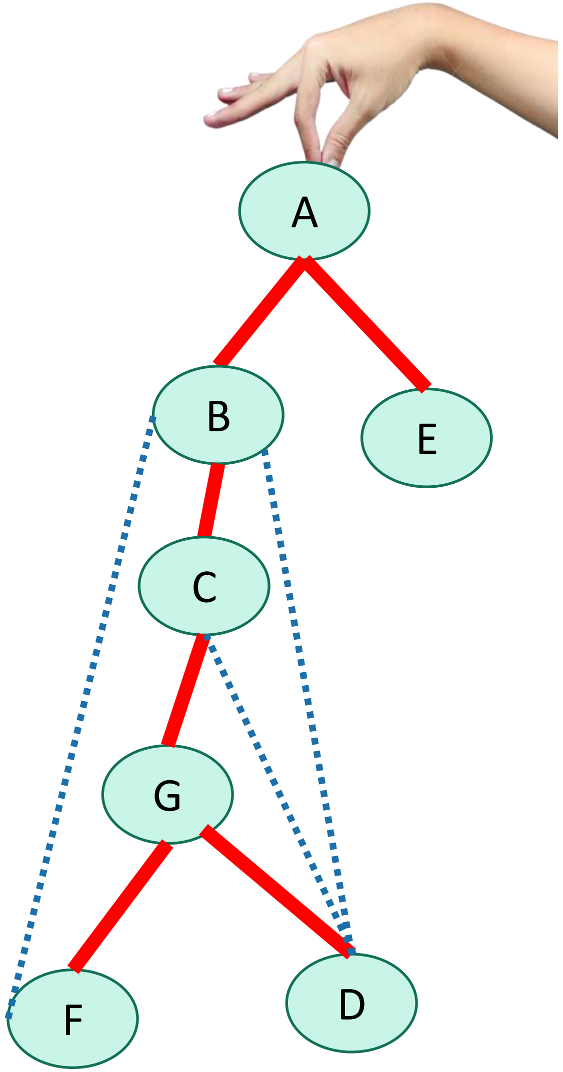




Why is it called depth-first?

- We are implicitly building a tree:

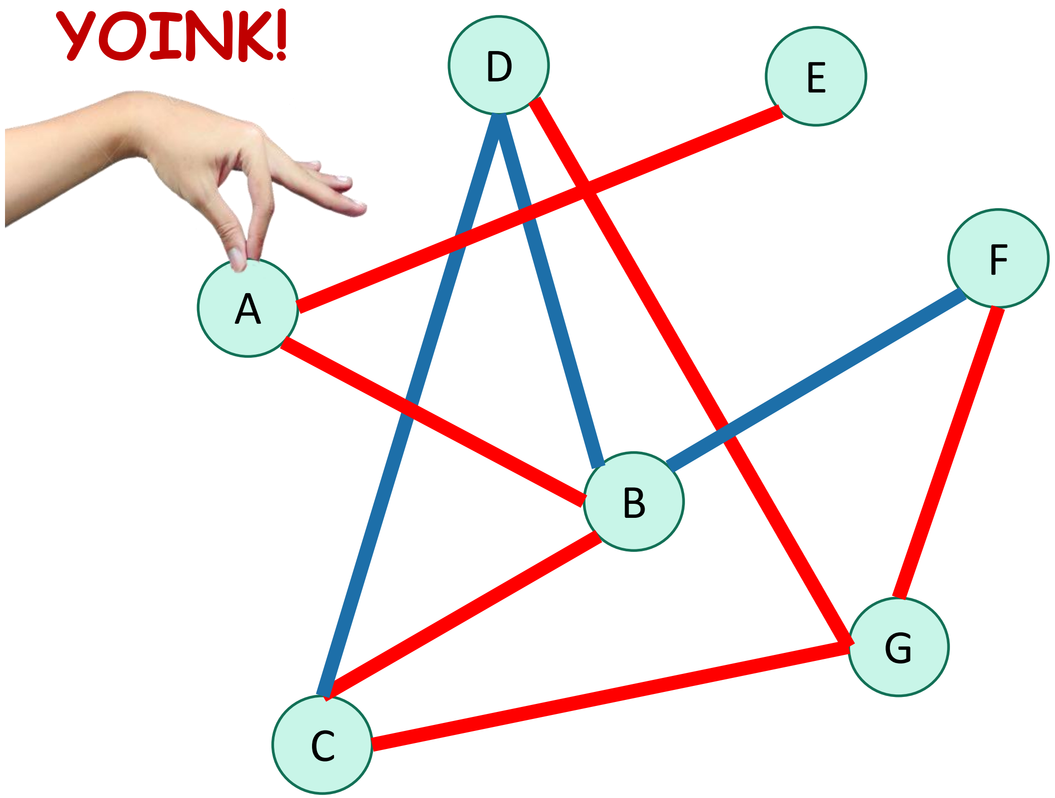


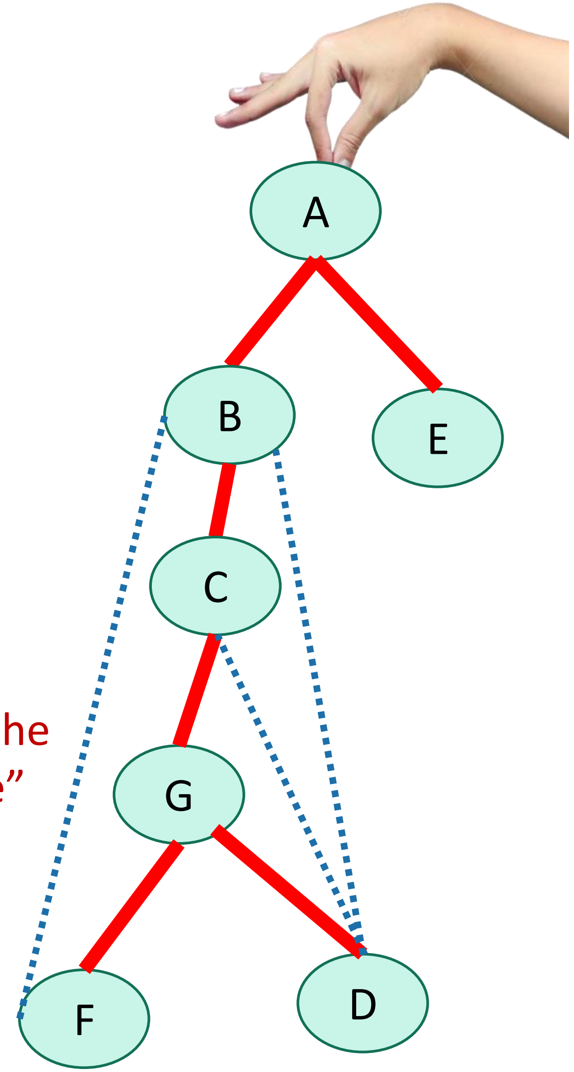


- First, we go as deep as we can.

Why is it called depth-first?

- We are implicitly building a tree:





Call this the "DFS tree"

- First, we go as deep as we can.

Running time

To explore **just the connected component** we started in

- We look at each edge at most twice.
  - Once from each of its endpoints
- And basically we don’t do anything else.
- So…

Running time

To explore **just the connected component** we started in

- We look at each edge at most twice.
  - Once from each of its endpoints
- And basically we don't do anything else.
- So…

$$O(m)$$

Running time

To explore just the connected component we started in

- Assume we are using the linked-list format for G.
- Say C = (V', E') is a connected component.
- We visit each vertex in C exactly once.
  - Here, "visit" means "call DFS on"

Running time

To explore just the connected component we started in

- Assume we are using the linked-list format for G.
- Say C = (V', E') is a connected component.
- We visit each vertex in C exactly once.
  - Here, "visit" means "call DFS on"
- At each vertex w, we:
  - Do some book-keeping: O(1)
  - Loop over w's neighbors and check if they are visited (and then potentially make a recursive call): O(1) per neighbor or O(deg(w)) total.

Running time

To explore just the connected component we started in

- Assume we are using the linked-list format for G.
- Say C = (V', E') is a connected component.
- We visit each vertex in C exactly once.
  - Here, "visit" means "call DFS on"
- At each vertex w, we:
  - Do some book-keeping: O(1)
  - Loop over w's neighbors and check if they are visited (and then potentially make a recursive call): O(1) per neighbor or O(deg(w)) total.
- Total time:
  - $\sum_{w \in V'} (O(\deg(w)) + O(1))$
  - $= O(|E'| + |V'|)$
  - $= O(|E'|)$

Running time

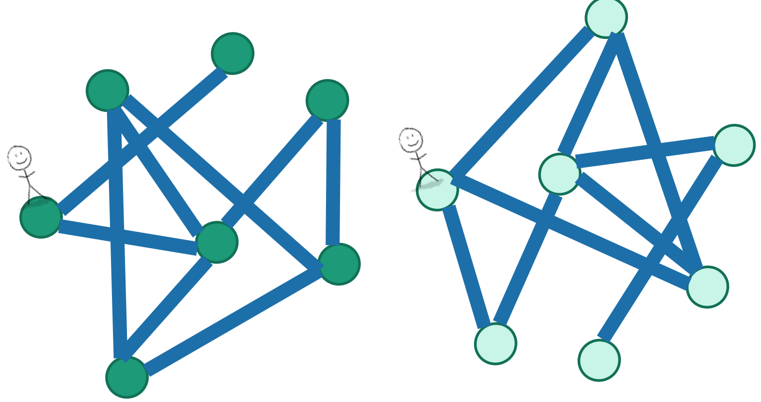
To explore just the connected component we started in

- Assume we are using the linked-list format for G.
- Say C = (V', E') is a connected component.
- We visit each vertex in C exactly once.
  - Here, "visit" means "call DFS on"
- At each vertex w, we:
  - Do some book-keeping: O(1)
  - Loop over w's neighbors and check if they are visited (and then potentially make a recursive call): O(1) per neighbor or O(deg(w)) total.
- Total time:
  - $\sum_{w \in V'} (O(\deg(w)) + O(1))$
  - $= O(|E'| + |V'|)$
  - $= O(|E'|)$

In a connected graph, $|V'| \leq |E'| + 1$.

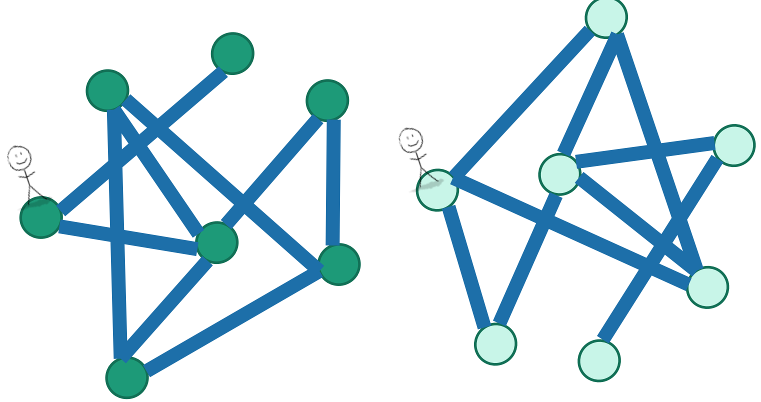
Running time

To explore the whole graph



Running time

To explore the whole graph

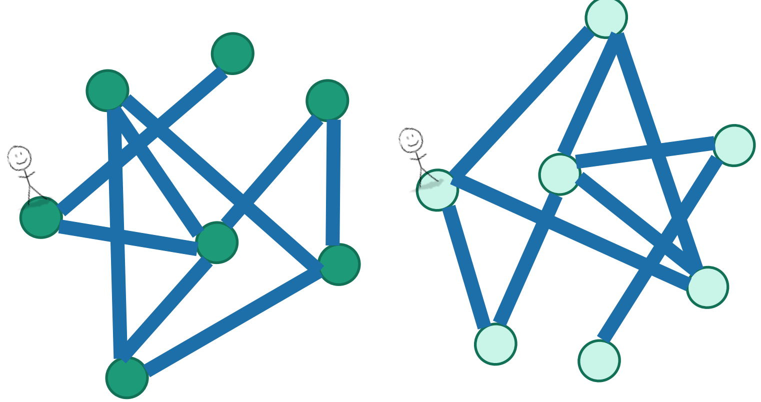


- Explore the connected components one-by-one.
- This takes time O(n + m)
  - Same computation as before:

$$\sum_{w \in V} \left(O(\deg(w)) + O(1) \right) = O(|E| + |V|) = O(n + m)$$

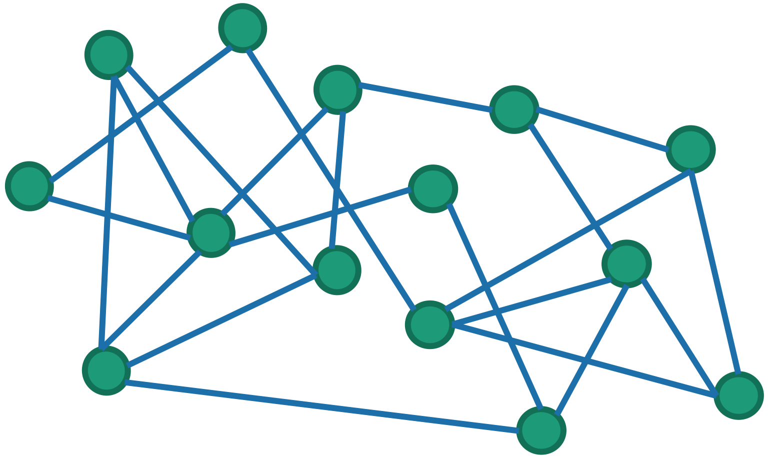
Running time

To explore the whole graph



- Explore the connected components one-by-one.
- This takes time O(n + m)
  - Same computation as before:

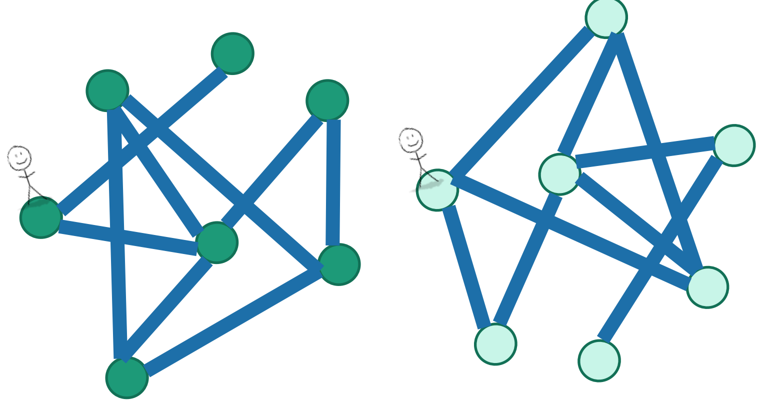
$$\sum_{w \in V} (O(\deg(w)) + O(1)) = O(|E| + |V|) = O(n + m)$$



Here the running time is O(m) like before

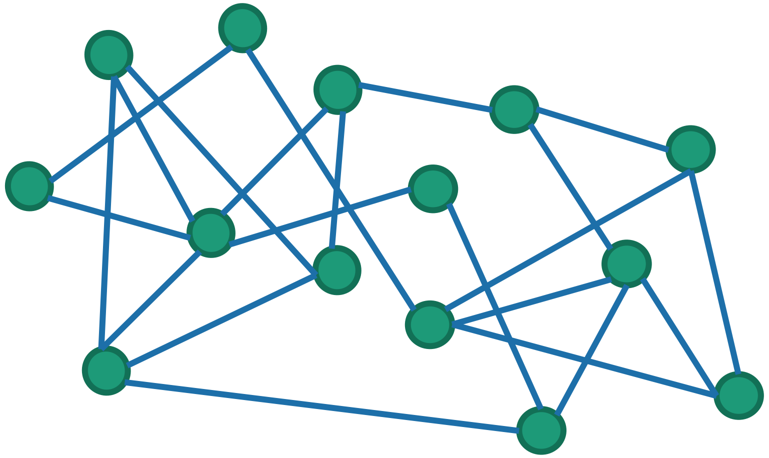
Running time

To explore the whole graph



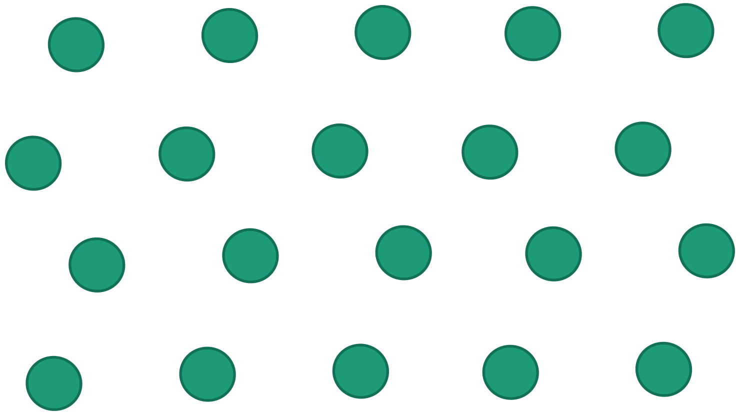
- Explore the connected components one-by-one.
- This takes time O(n + m)
  - Same computation as before:

$$\sum_{w \in V} (O(\deg(w)) + O(1)) = O(|E| + |V|) = O(n + m)$$



Here the running time is O(m) like before

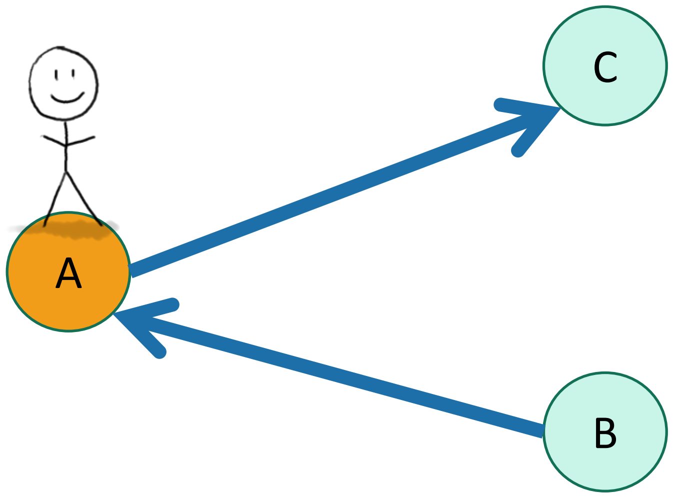
or



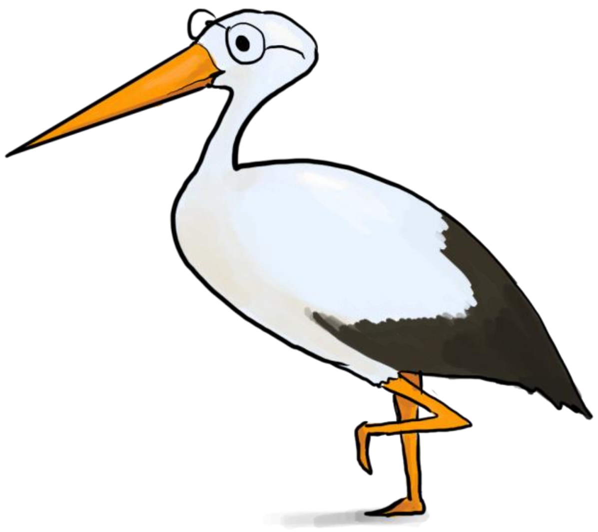
Here m=0 but it still takes time O(n) to explore the graph.

You check:

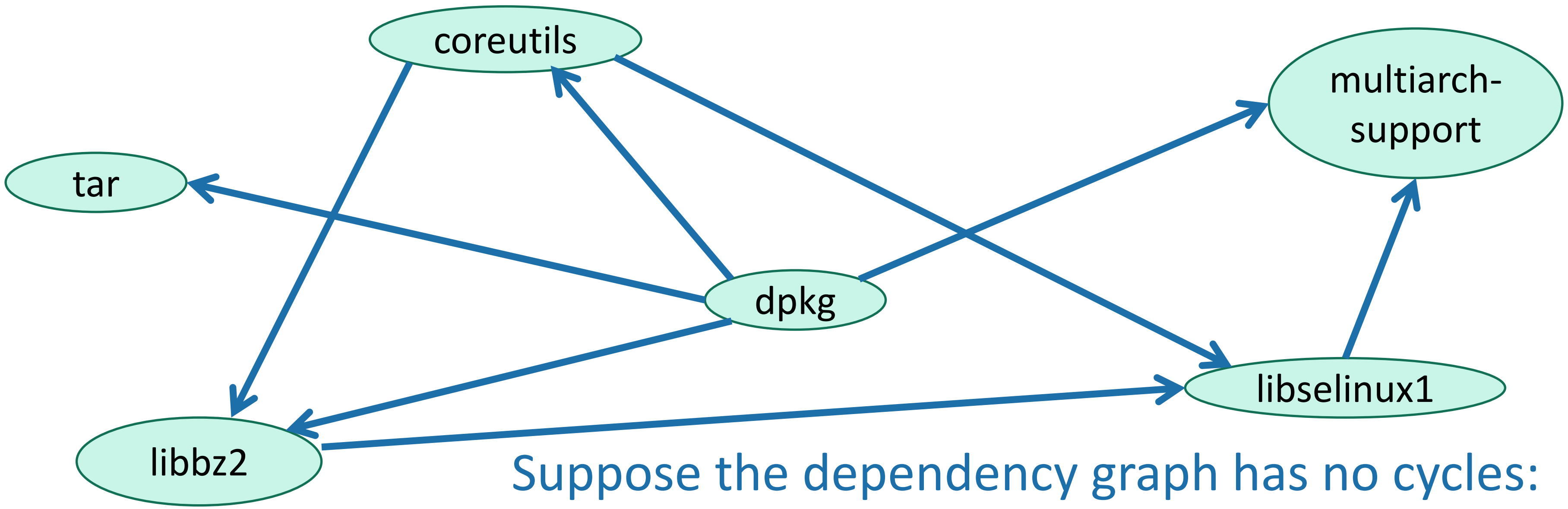
DFS works fine on directed graphs too!



Only walk to C, not to B.



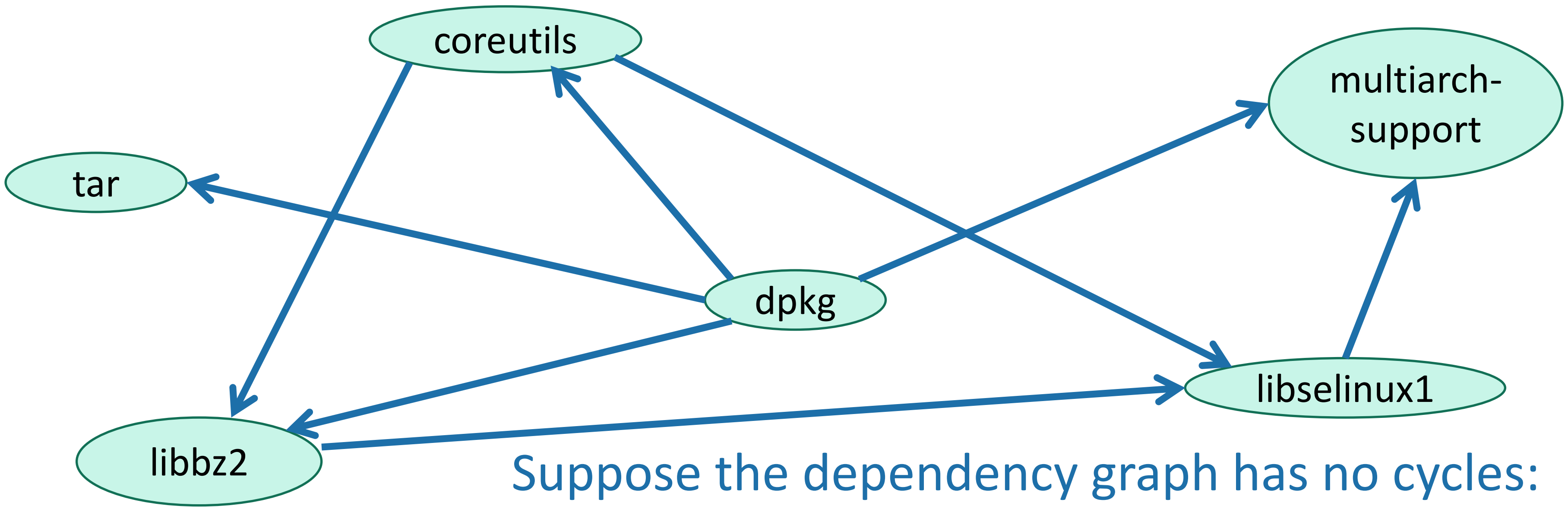
Application of DFS: topological sorting



Suppose the dependency graph has no cycles:
it is a **Directed Acyclic Graph (DAG)**

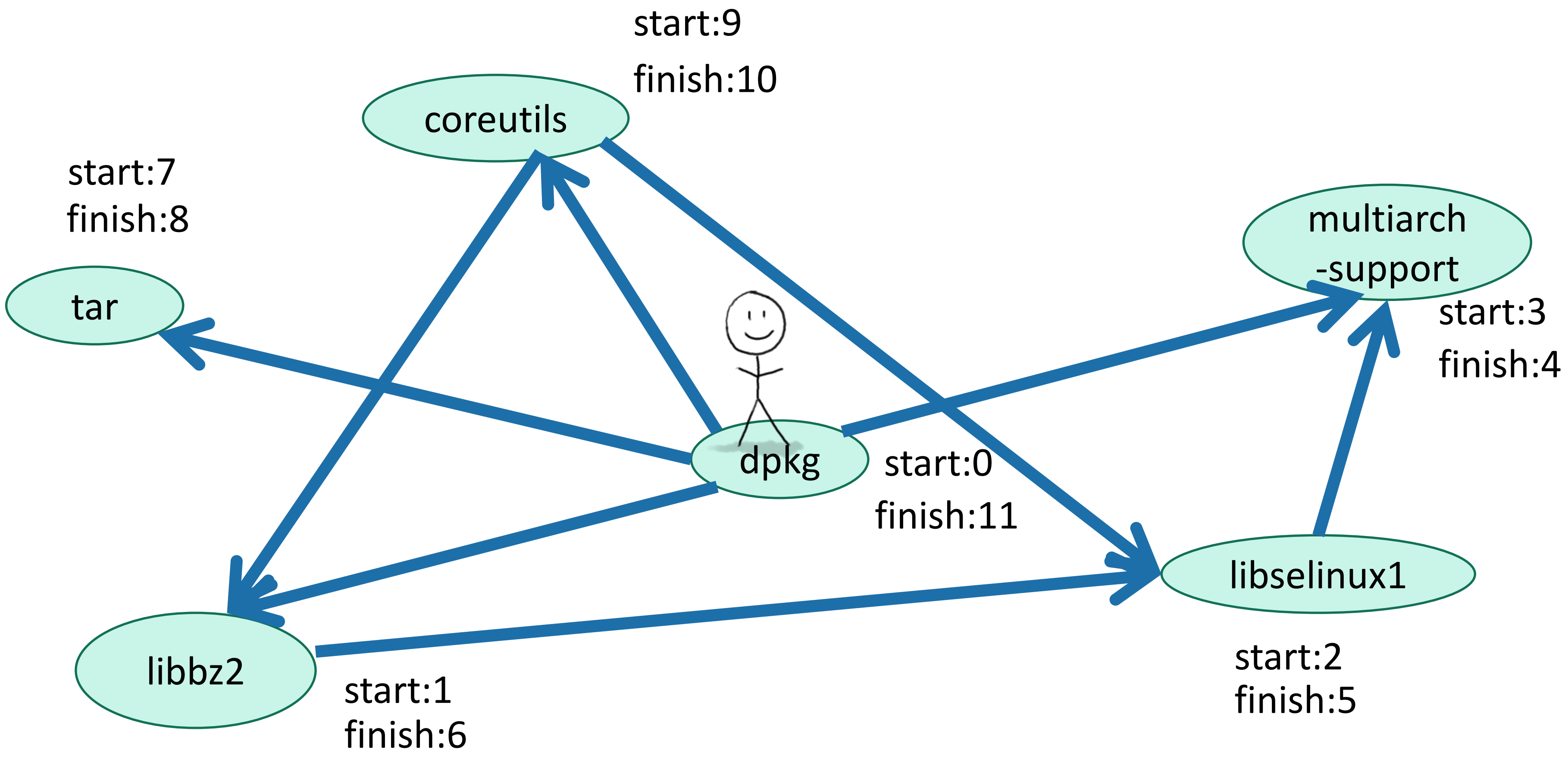
Application of DFS: topological sorting

- Find an ordering of vertices so that all of the dependency requirements are met.
  - Aka, if v comes before w in the ordering, there is not an edge from w to v.



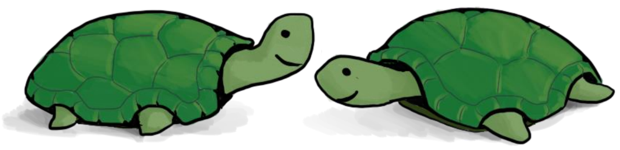
Suppose the dependency graph has no cycles: it is a **Directed Acyclic Graph (DAG)**

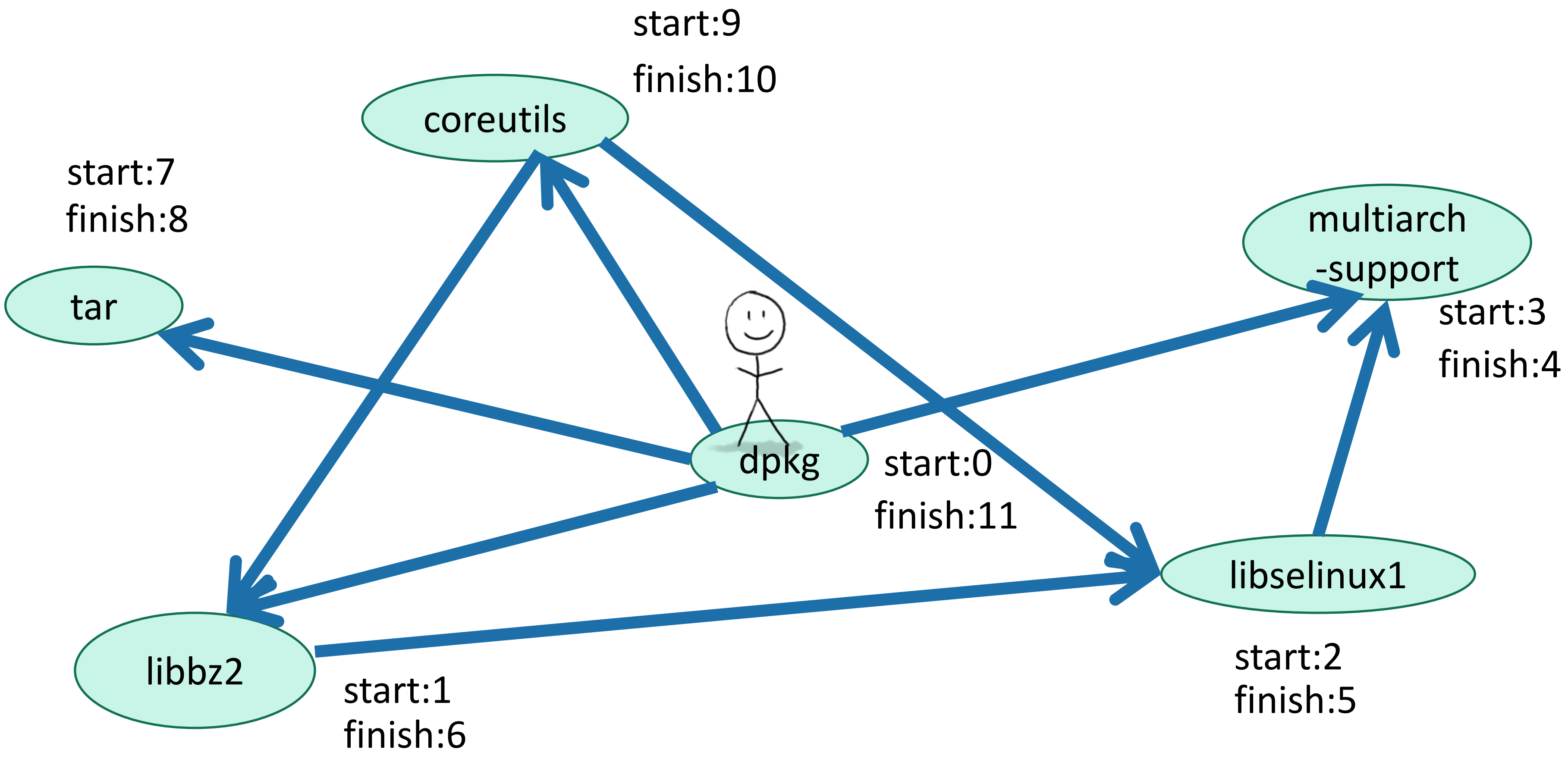
Let’s do DFS


start:9
finish:10
coreutils
start:7
finish:8
tar
multiarch
-support
start:3
finish:4
dpkg
start:0
finish:11
libselinux1
start:2
finish:5
libbz2
start:1
finish:6

Let’s do DFS

What do you notice about the finish times? Any ideas for how we should do topological sort?

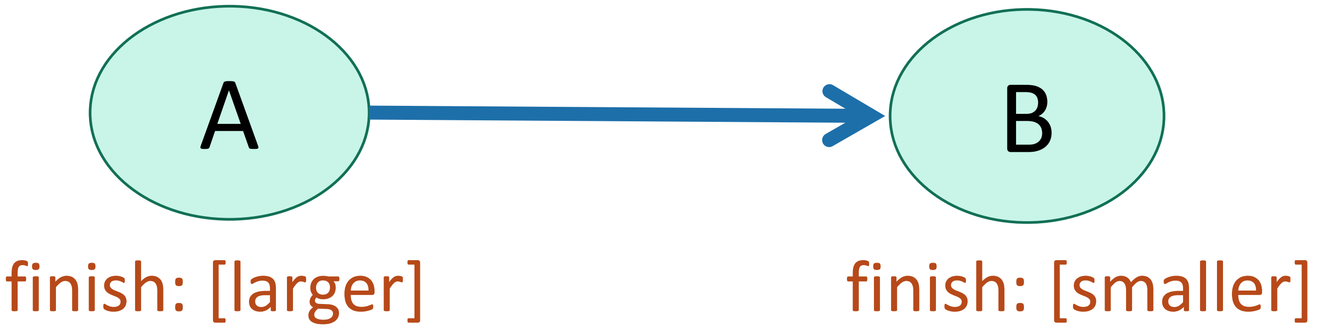




Suppose the underlying graph has no cycles

Finish times seem useful

Claim: In general, we'll always have:



Finish times seem useful

Suppose the underlying graph has no cycles

Claim: In general, we'll always have:

finish: [larger]

finish: [smaller]

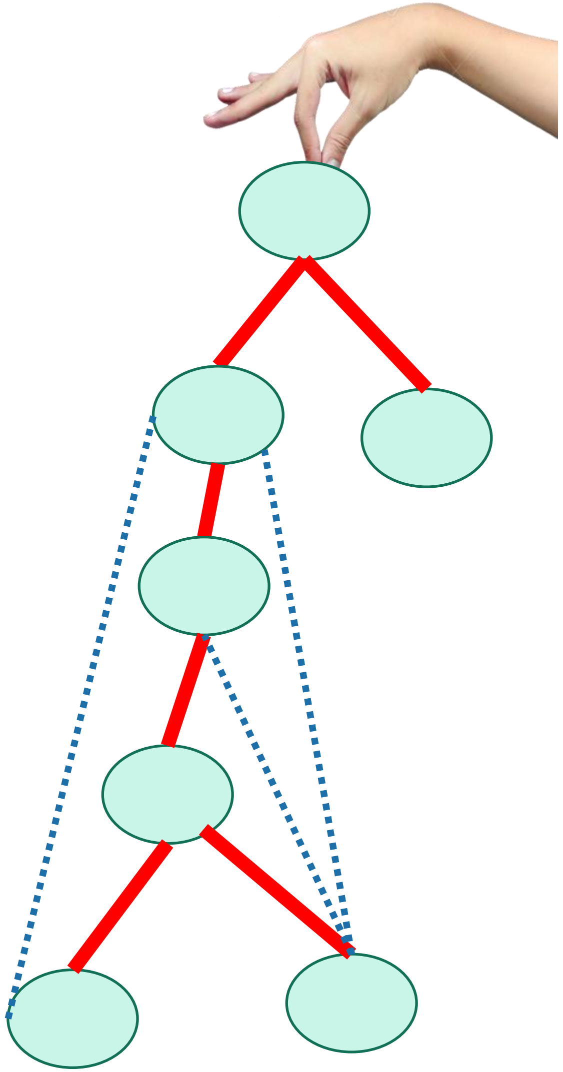
To understand why, let's go back to that DFS tree.

A more general statement

(this holds even if there are cycles)

This is called the “parentheses theorem”

(check this statement carefully!)



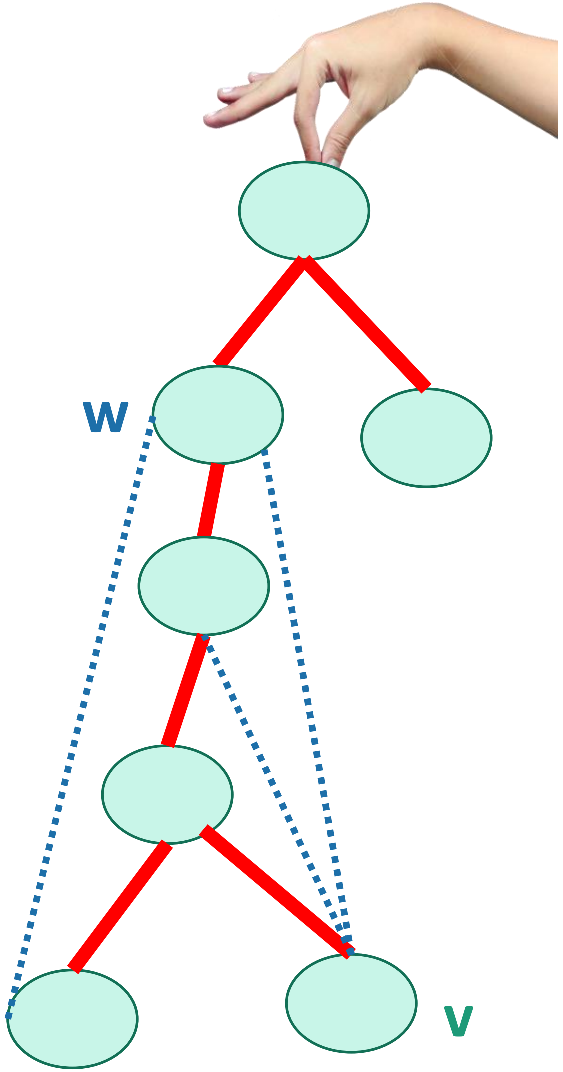
A more general statement

(this holds even if there are cycles)

This is called the “parentheses theorem”

(check this statement carefully!)

- If v is a descendant of w in this tree:



A more general statement

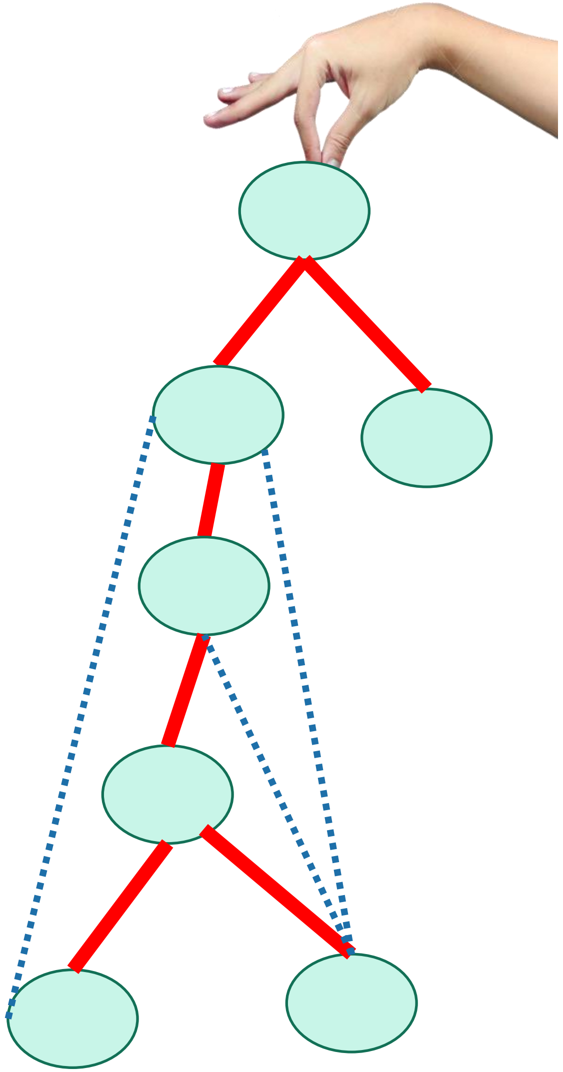
(this holds even if there are cycles)

This is called the "parentheses theorem"

(check this statement carefully!)

- If v is a descendant of w in this tree:

- If w is a descendant of v in this tree:



A more general statement

(this holds even if there are cycles)

This is called the “parentheses theorem”

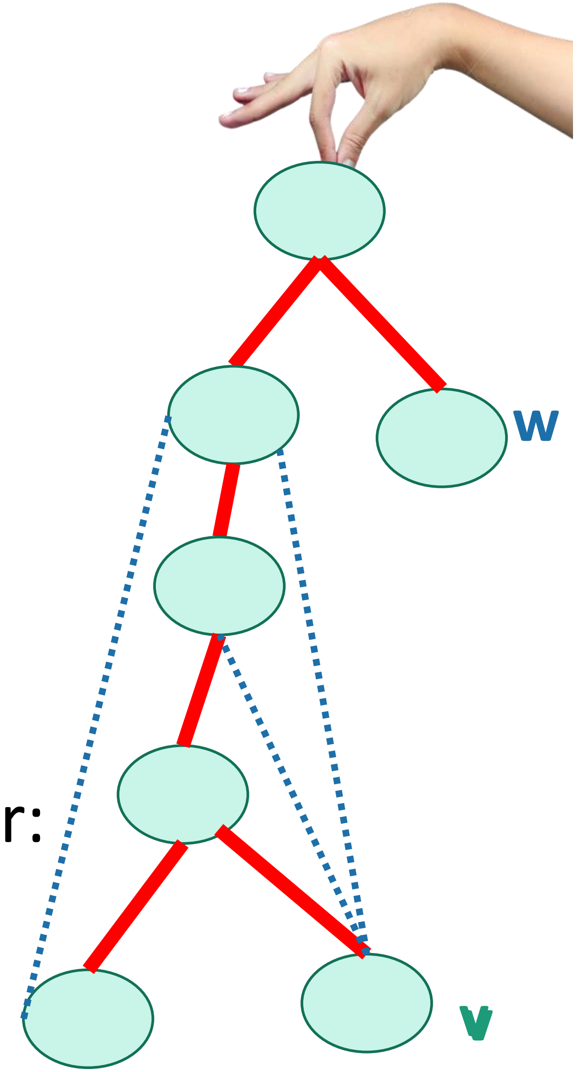
(check this statement carefully!)

- If v is a descendant of w in this tree:

- If w is a descendant of v in this tree:

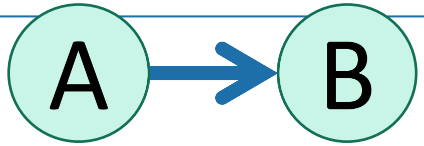
- If neither are descendants of each other:

(or the other way around)



Theorem

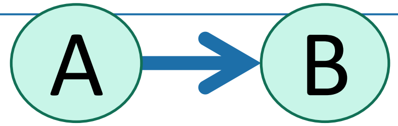
- If we run DFS on a directed acyclic graph,

If 

Then B.finishTime < A.finishTime

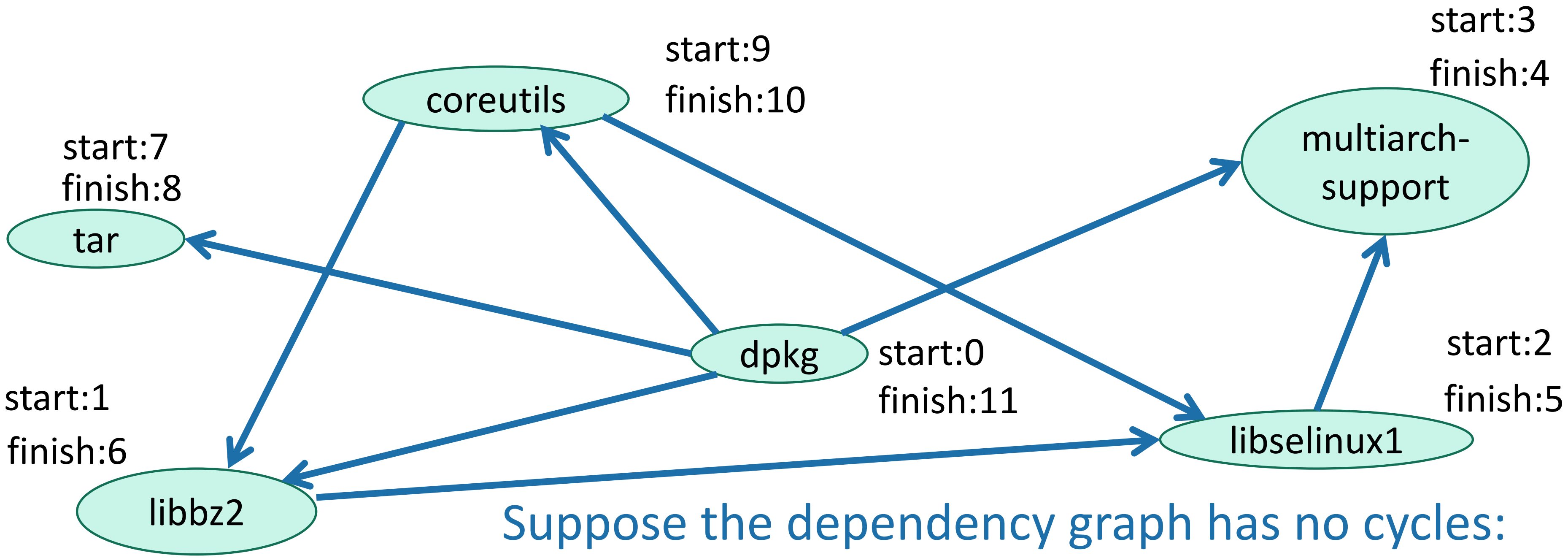
Back to topological sorting

If



Then B.finishTime < A.finishTime

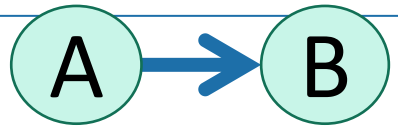
- In what order should I install packages?



Suppose the dependency graph has no cycles: it is a **Directed Acyclic Graph (DAG)**

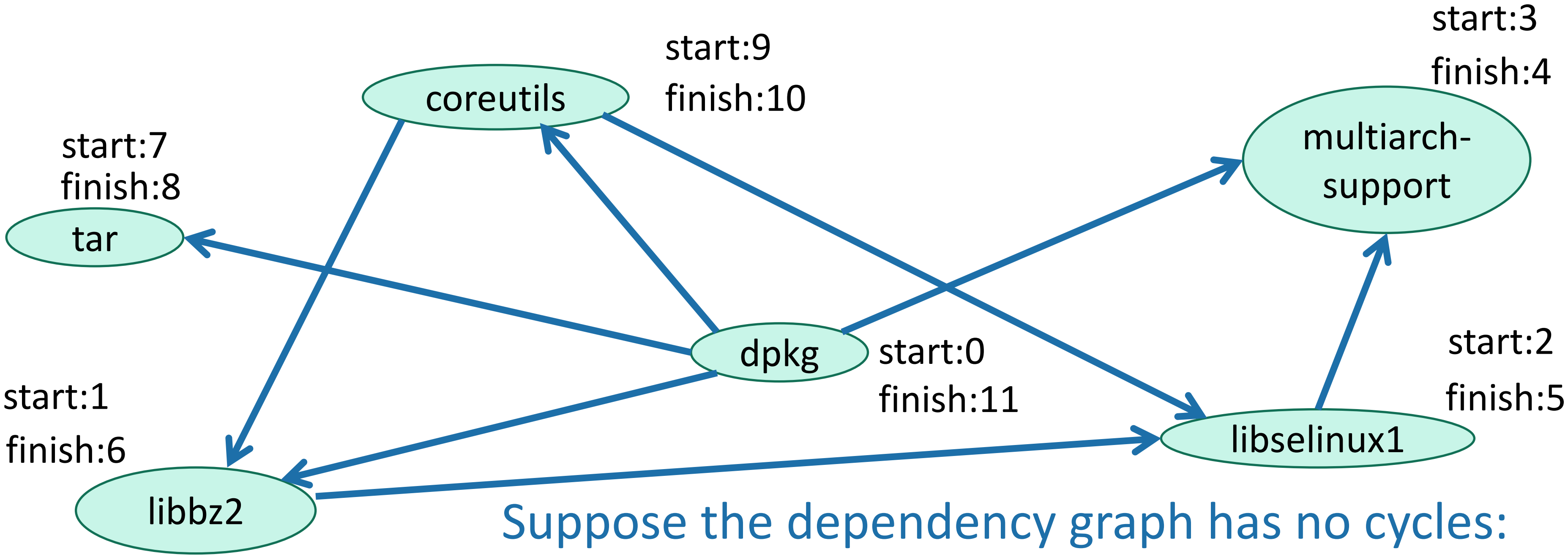
Back to topological sorting

If



Then B.finishTime < A.finishTime

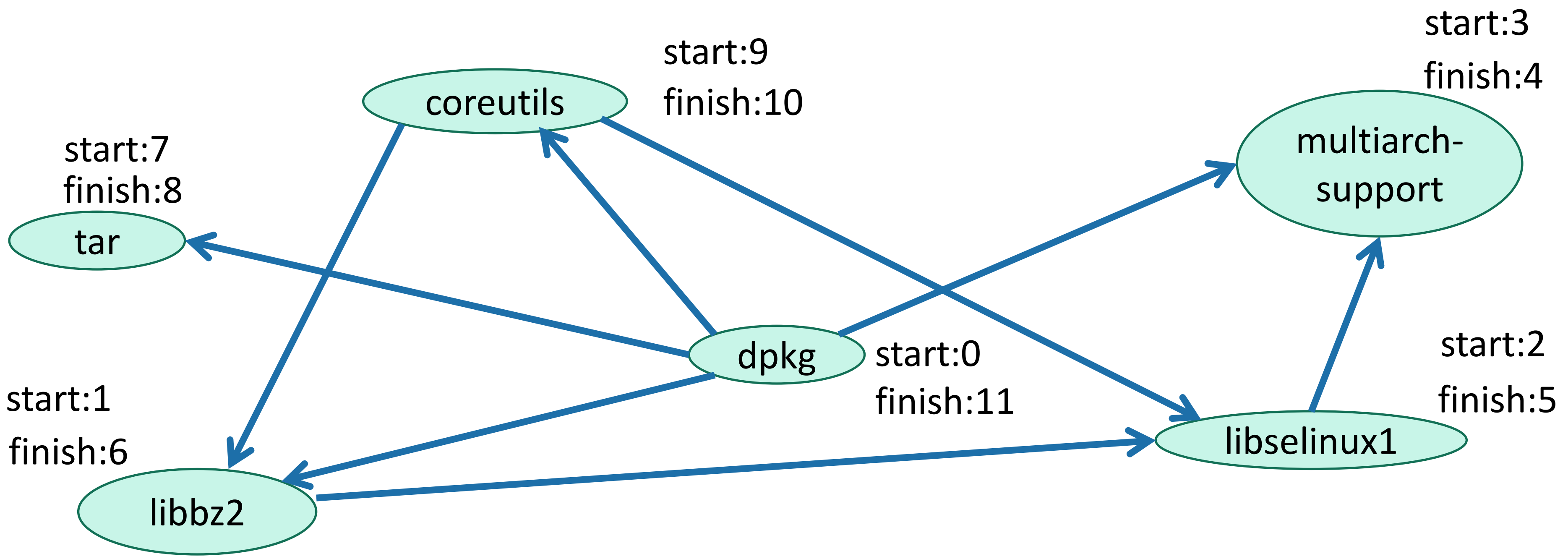
- In what order should I install packages?
- In reverse order of finishing time in DFS!



Suppose the dependency graph has no cycles:
it is a **Directed Acyclic Graph (DAG)**

Topological Sorting (on a DAG)

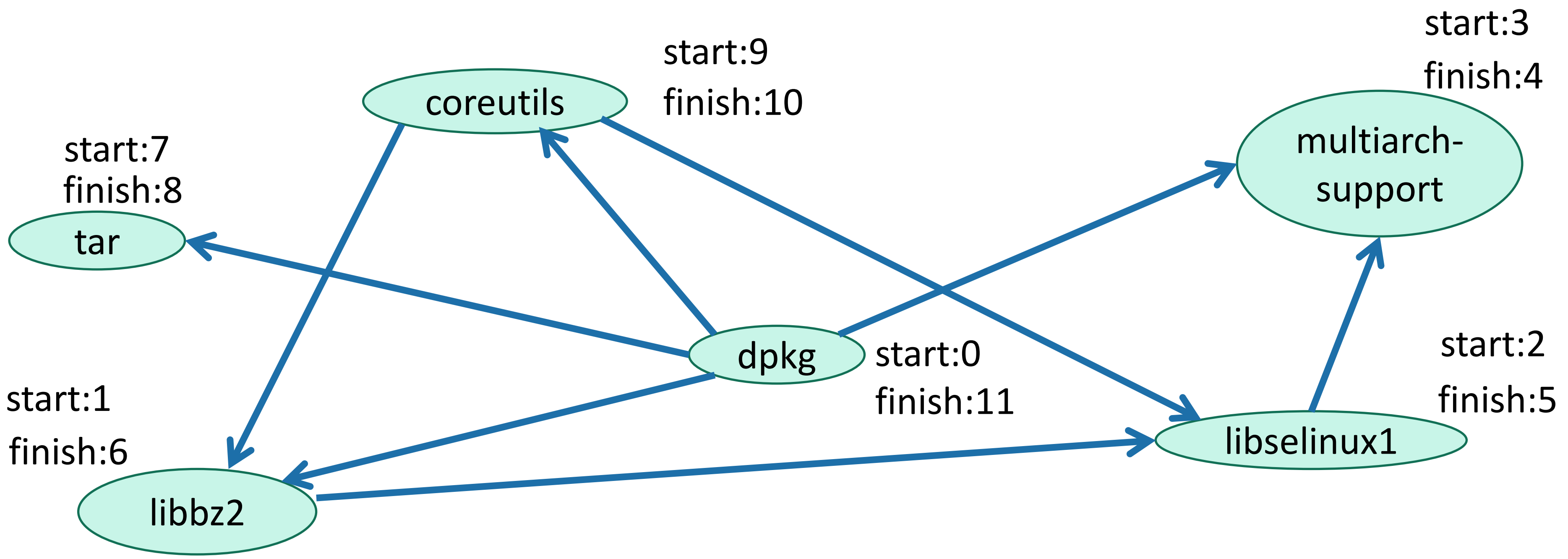
- Do DFS
- When you mark a vertex as **all done**, put it at the **beginning** of the list.



Topological Sorting (on a DAG)

- Do DFS
- When you mark a vertex as **all done,** put it at the **beginning** of the list.

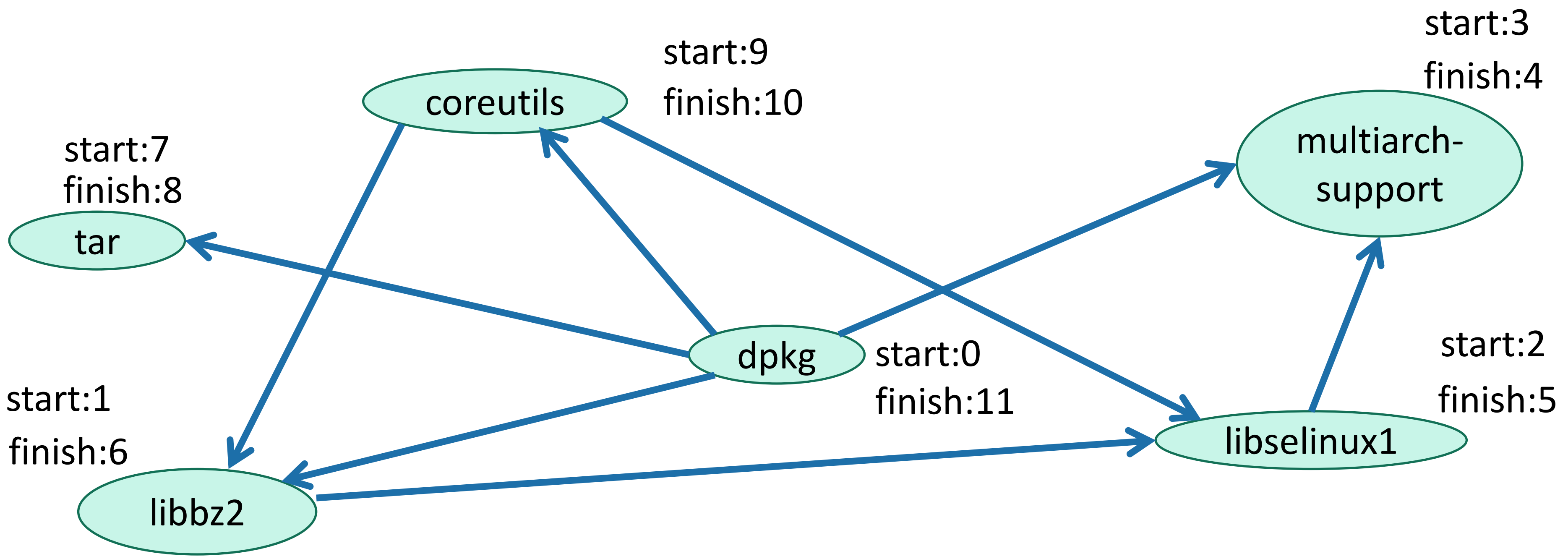
- `multiarch_support`



Topological Sorting (on a DAG)

- Do DFS
- When you mark a vertex as **all done**, put it at the **beginning** of the list.

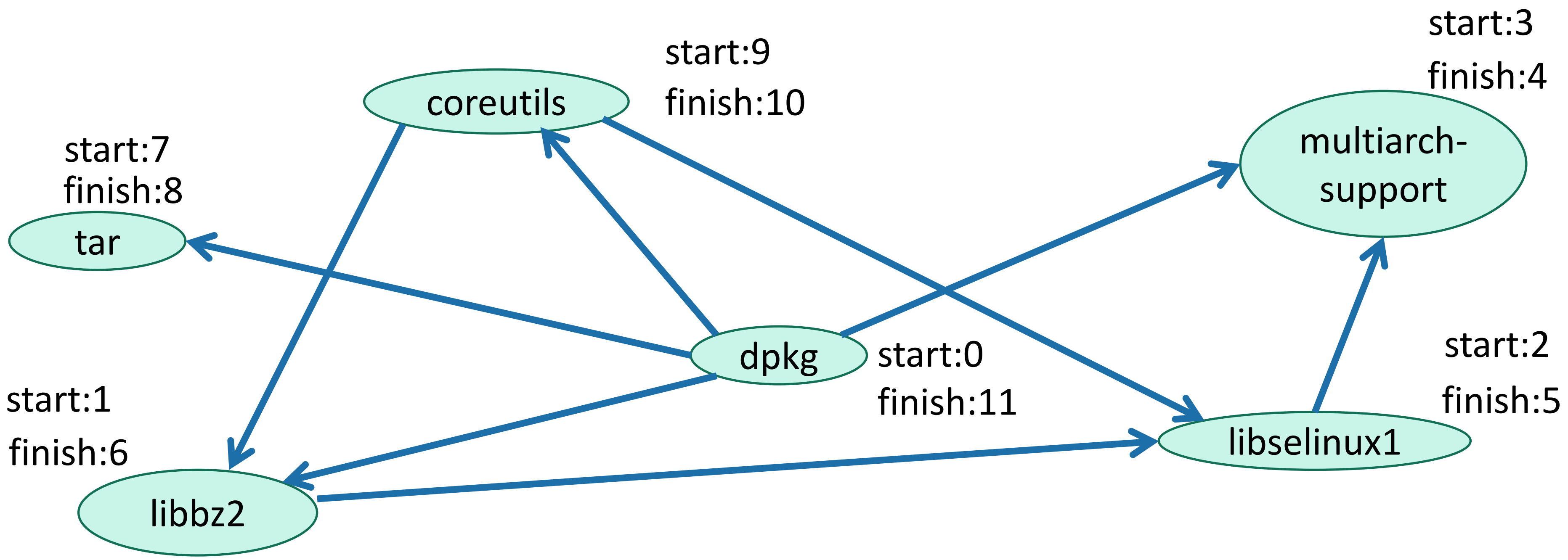
- `libselinux1`
- `multiarch_support`



Topological Sorting (on a DAG)

- Do DFS
- When you mark a vertex as **all done**, put it at the beginning of the list.

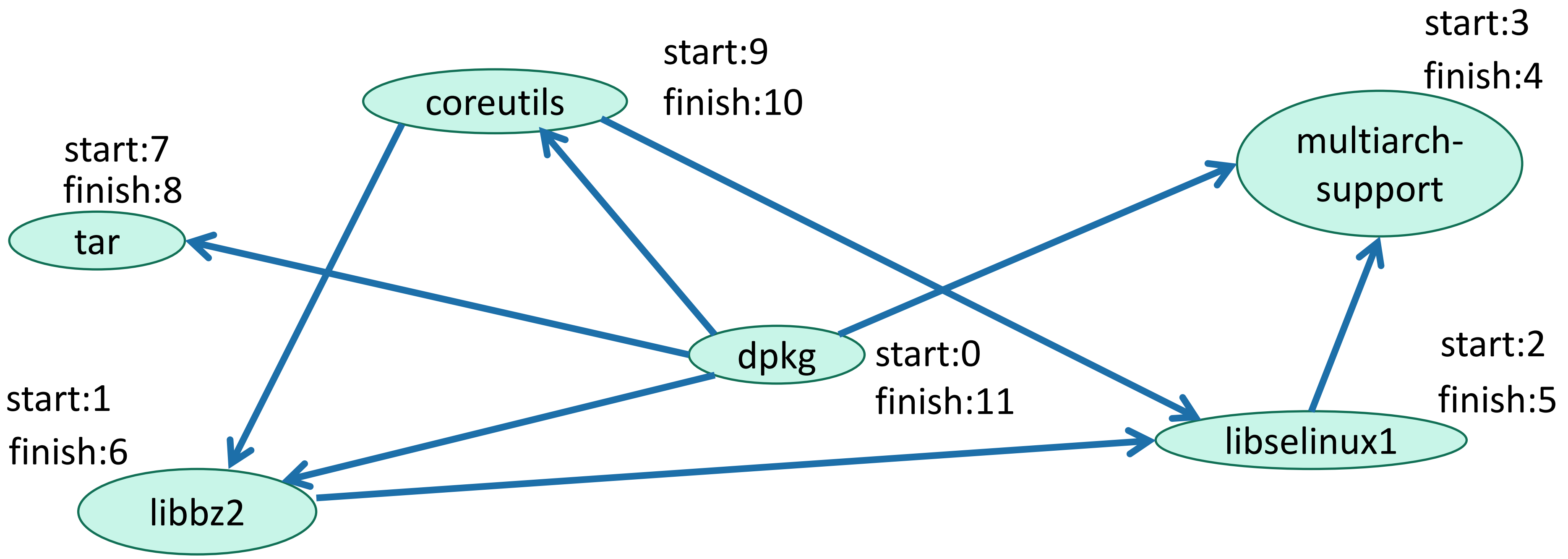
- `libbz2`
- `libselinux1`
- `multiarch_support`



Topological Sorting (on a DAG)

- Do DFS
- When you mark a vertex as **all done**, put it at the **beginning** of the list.

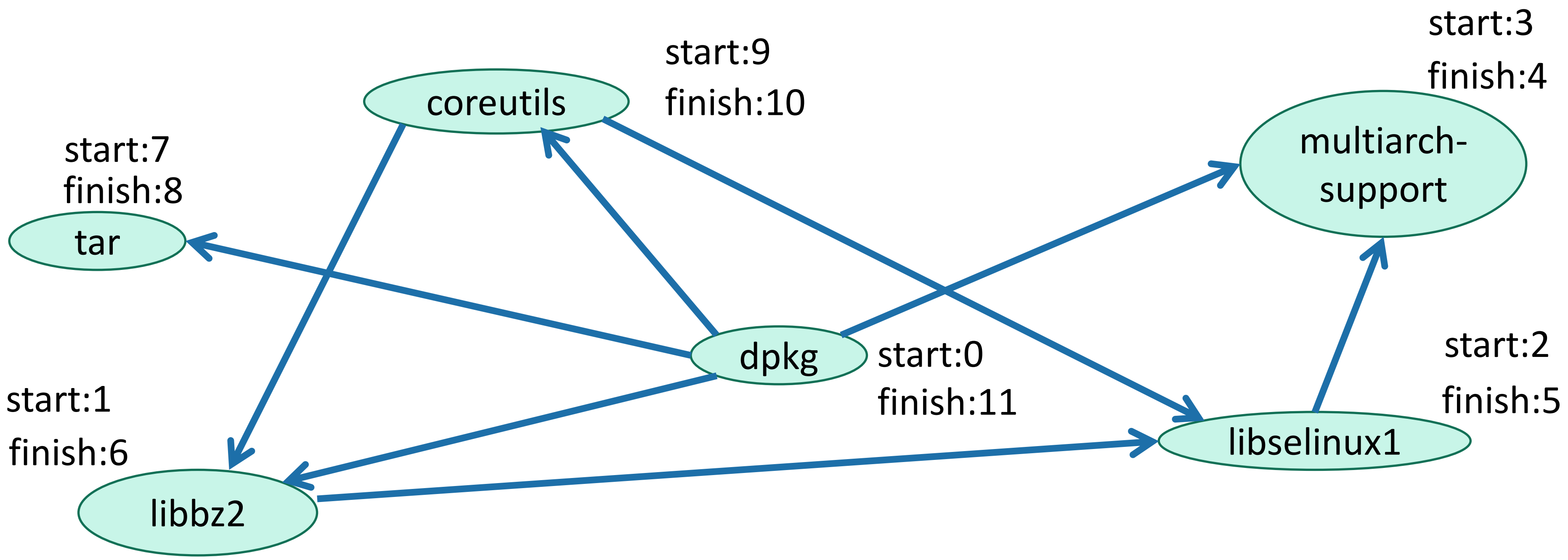
- `tar`
- `libbz2`
- `libselinux1`
- `multiarch_support`



Topological Sorting (on a DAG)

- Do DFS
- When you mark a vertex as **all done**, put it at the beginning of the list.

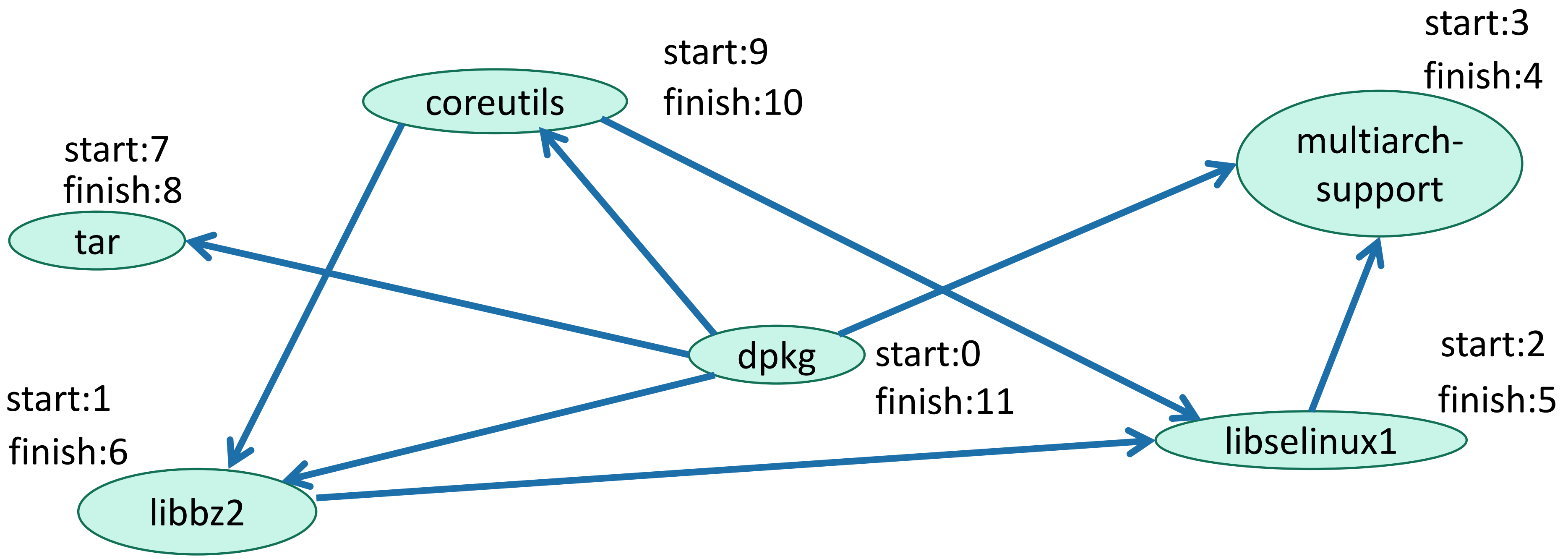
- `coreutils`
- `tar`
- `libbz2`
- `libselinux1`
- `multiarch_support`



Topological Sorting (on a DAG)

- Do DFS
- When you mark a vertex as **all done**, put it at the **beginning** of the list.

- `dpkg`
- `coreutils`
- `tar`
- `libbz2`
- `libselinux1`
- `multiarch_support`

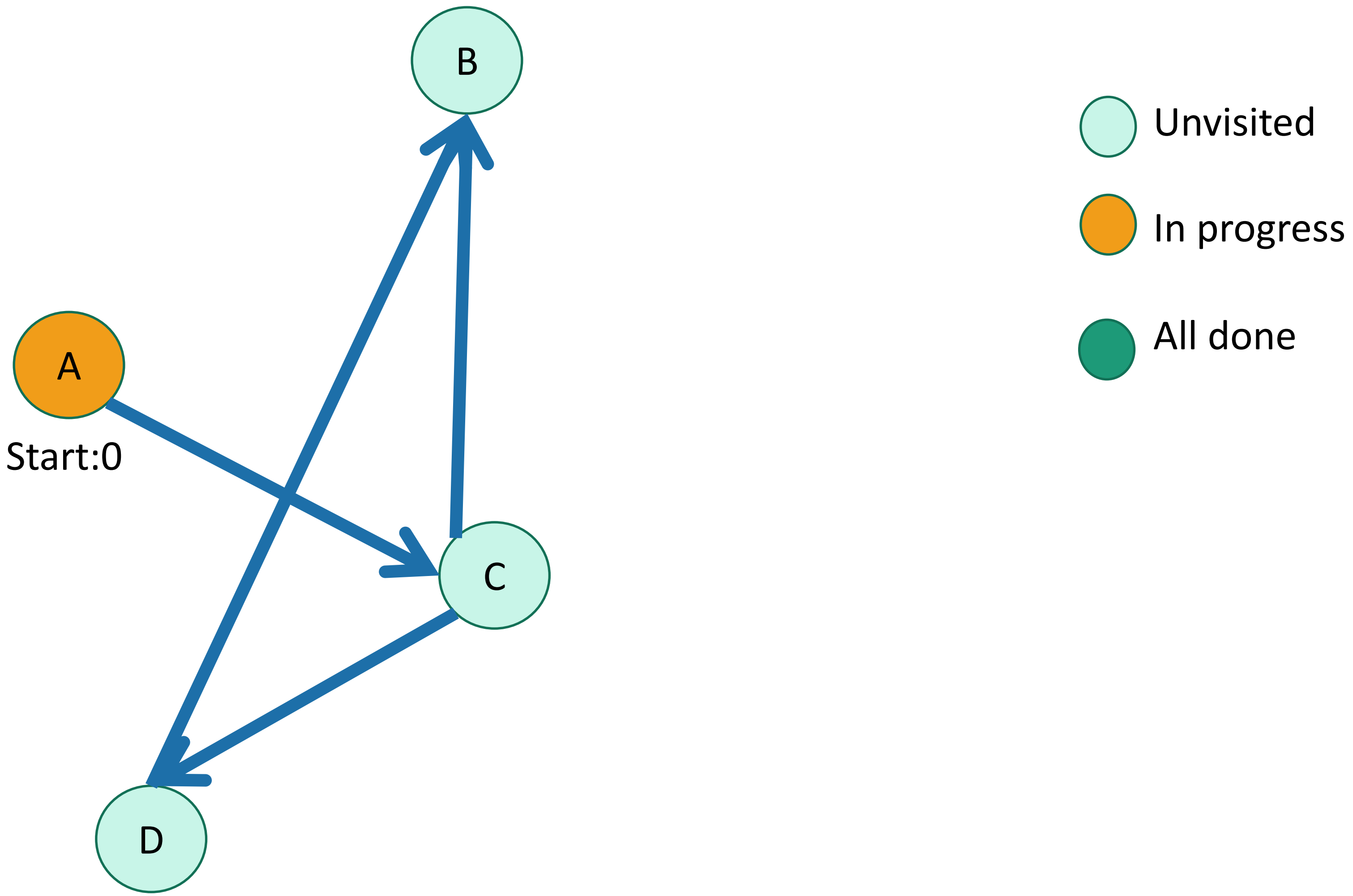


What did we just learn?

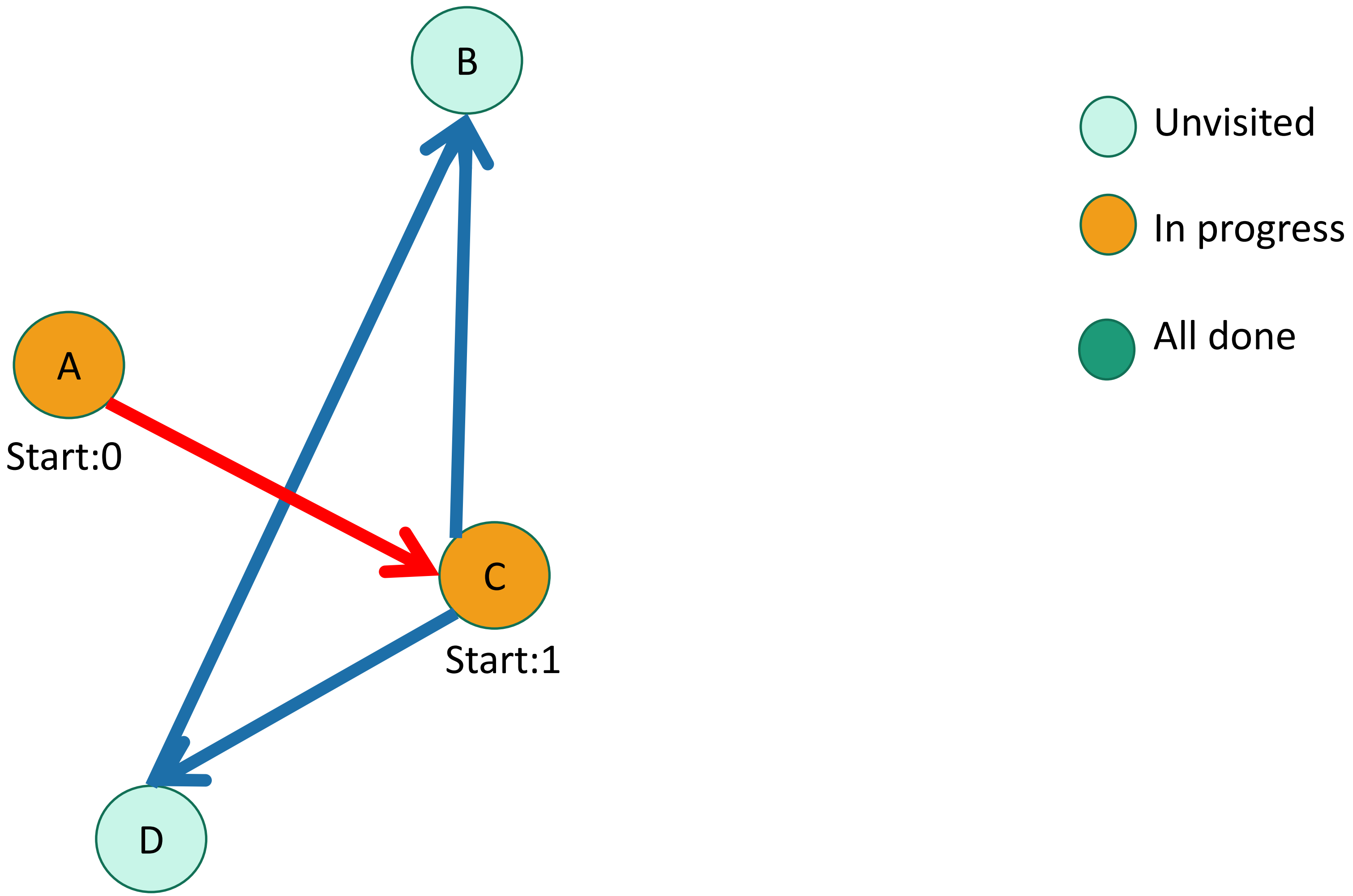
- DFS can help you solve the **topological sorting problem**
  - That’s the fancy name for the problem of finding an ordering that respects all the dependencies

- Thinking about the DFS tree is helpful.

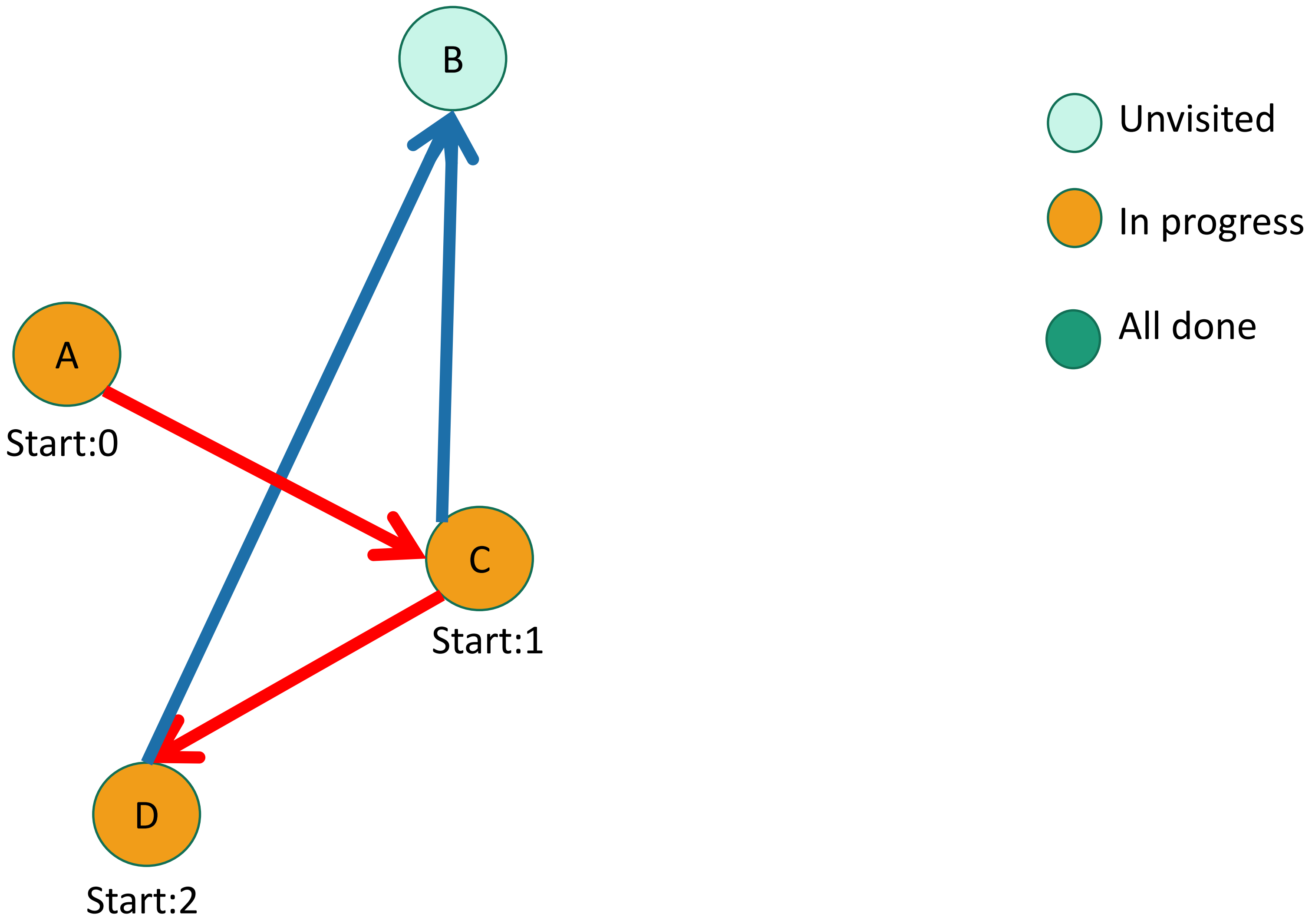
Example:



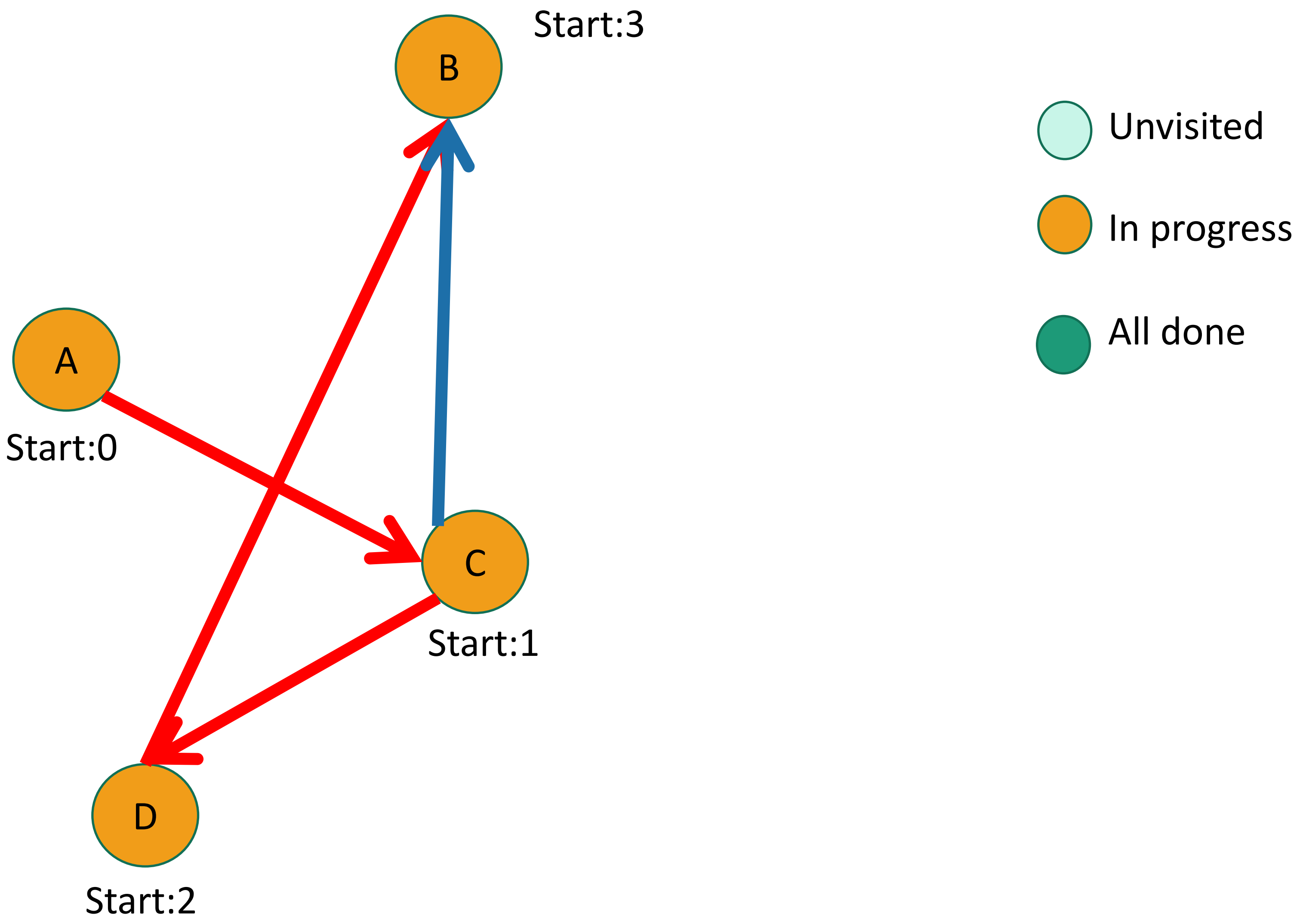
Example



Example

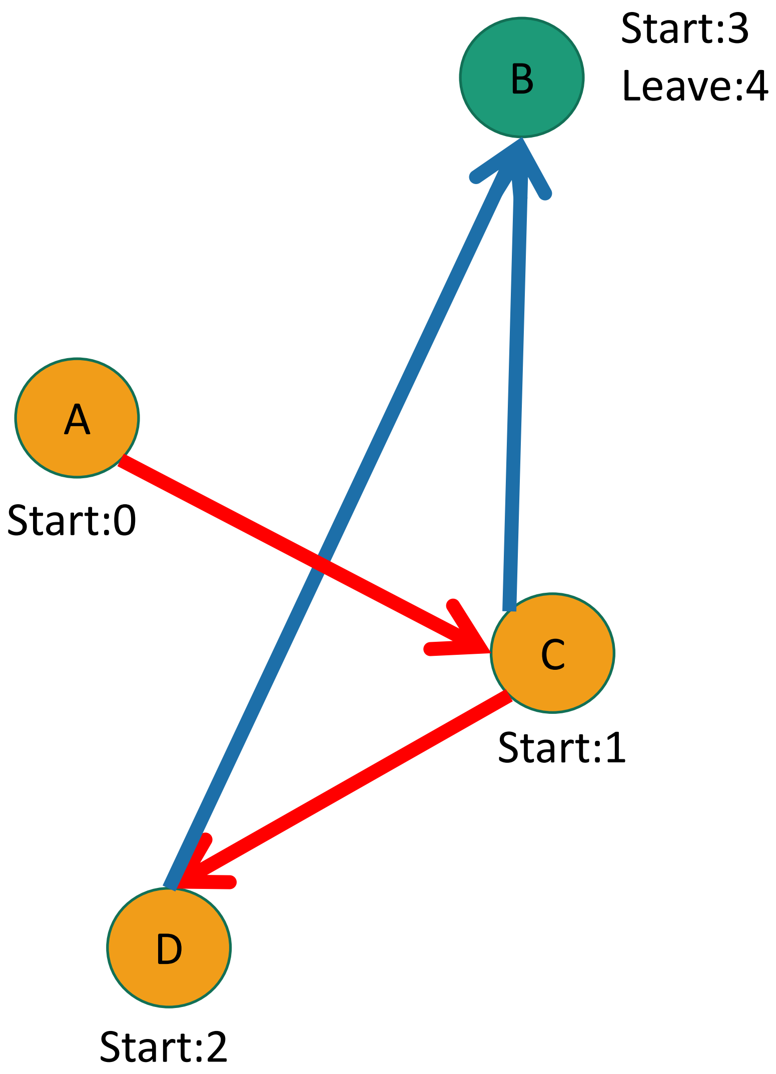


Example

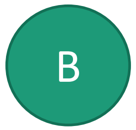


Start:3
B
A
Start:0
C
Start:1
D
Start:2
Unvisited
In progress
All done

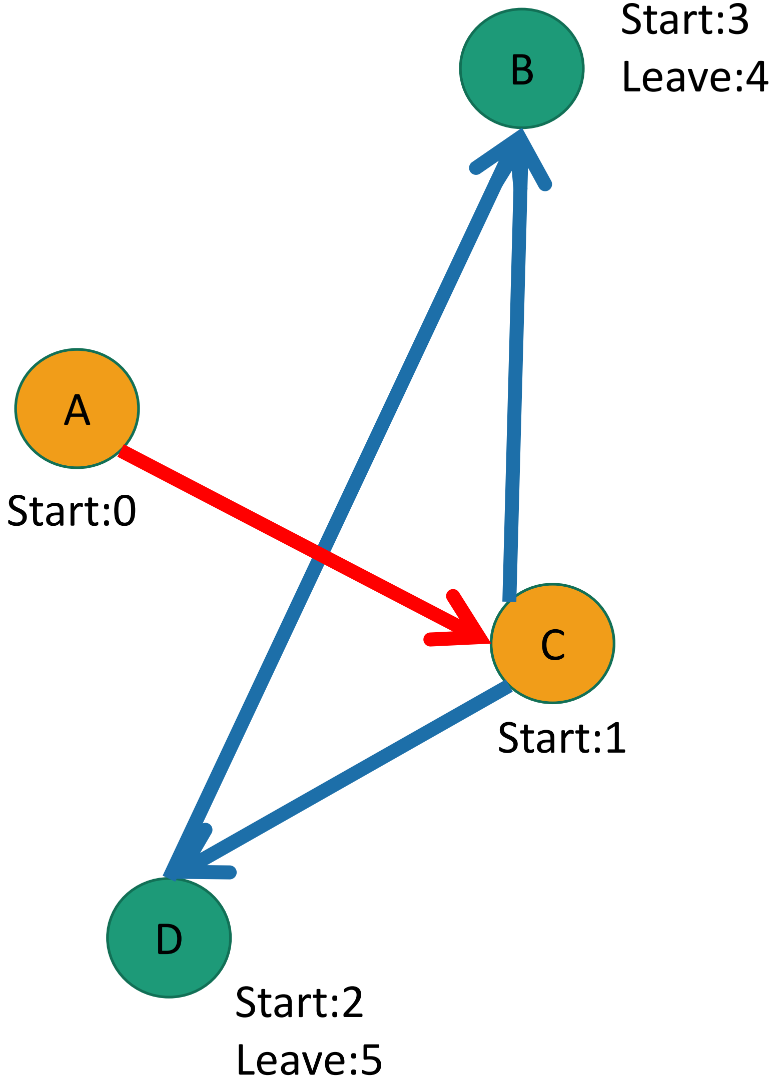
Example


B
Start:3
Leave:4
A
Start:0
C
Start:1
D
Start:2


Unvisited
In progress
All done

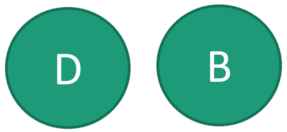

B

Example



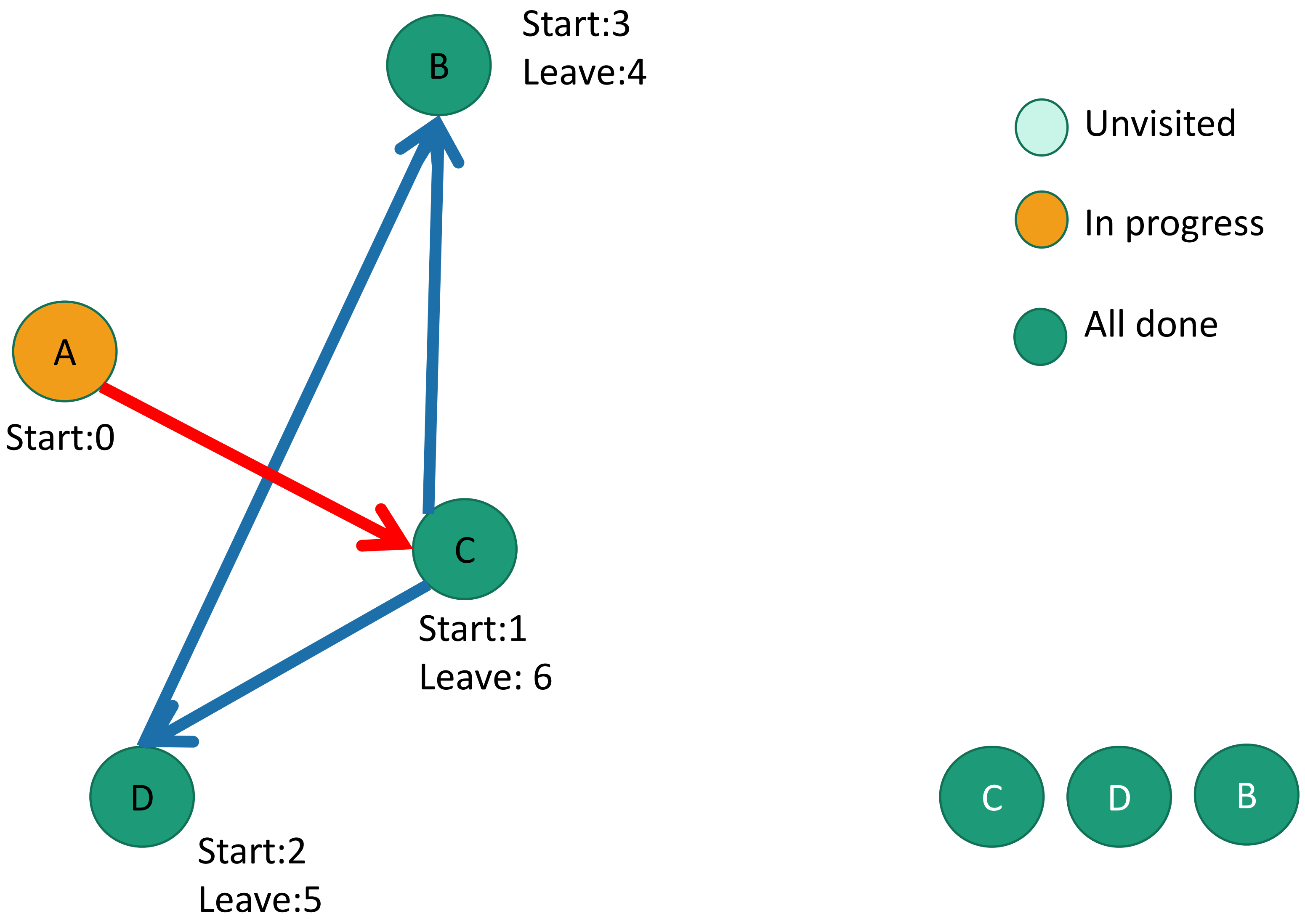
Start:3
Leave:4
B
A
Start:0
C
Start:1
D
Start:2
Leave:5

Unvisited
In progress
All done



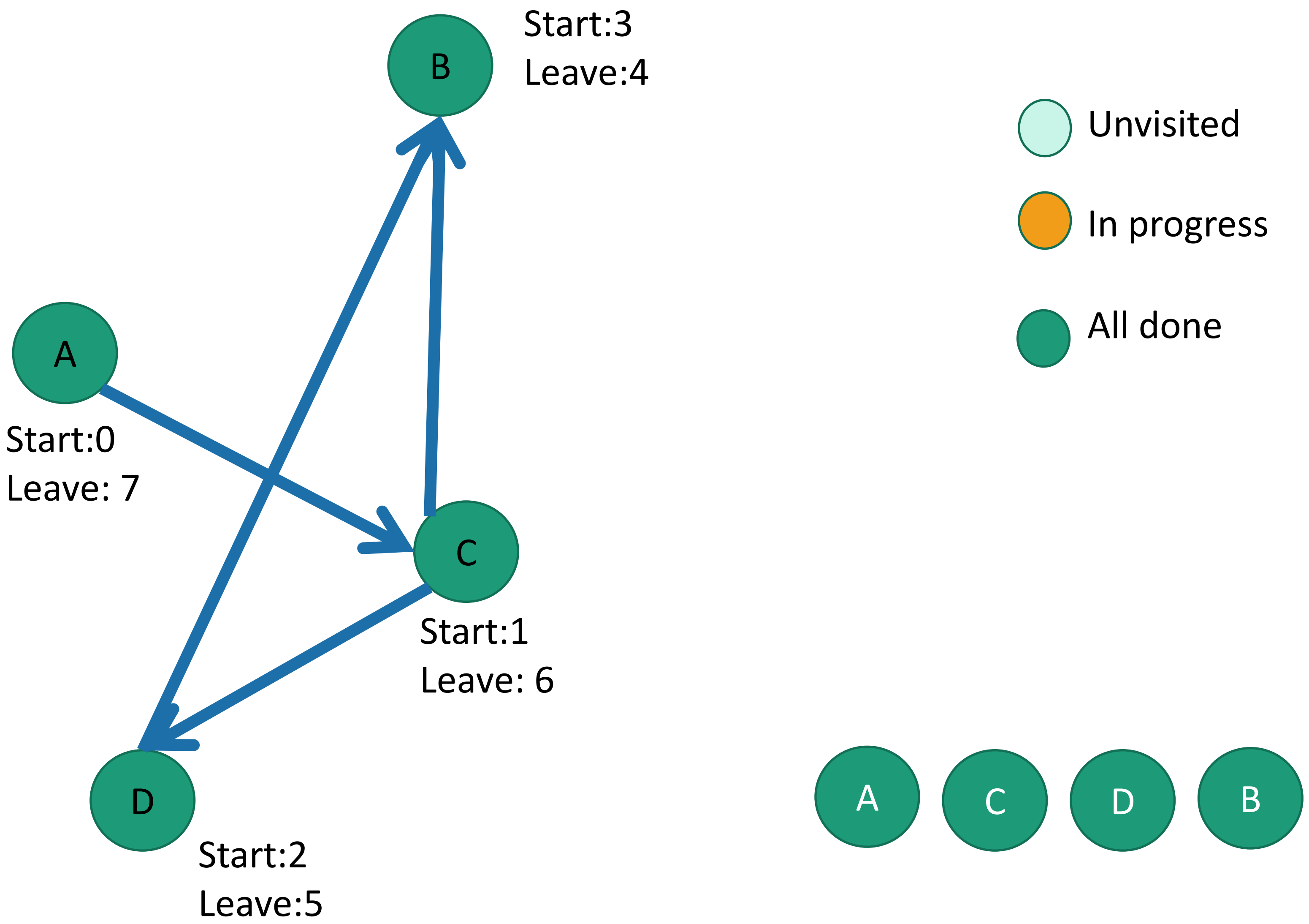
D
B

Example

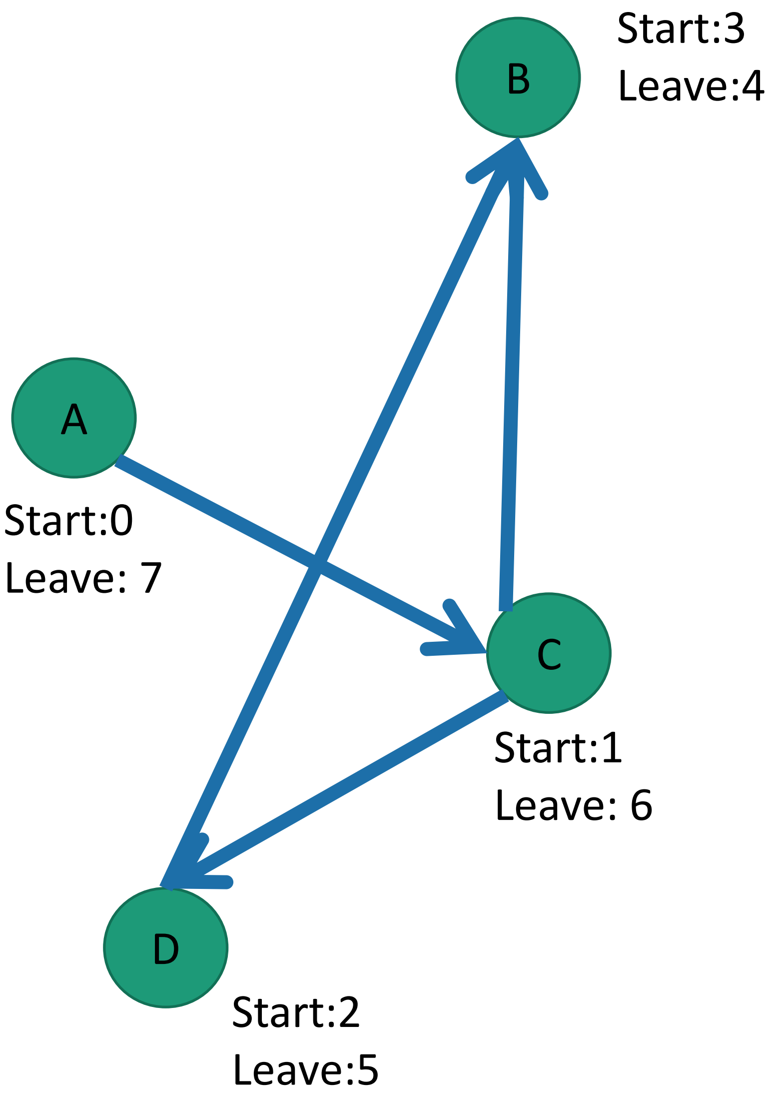


B
Start:3
Leave:4
A
Start:0
C
Start:1
Leave: 6
D
Start:2
Leave:5
Unvisited
In progress
All done
C
D
B

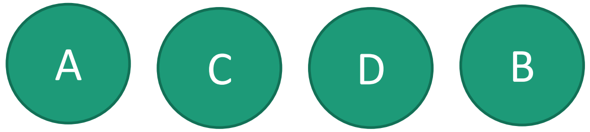
Example


B
Start:3
Leave:4
A
Start:0
Leave: 7
C
Start:1
Leave: 6
D
Start:2
Leave:5
Unvisited
In progress
All done
A
C
D
B

Example

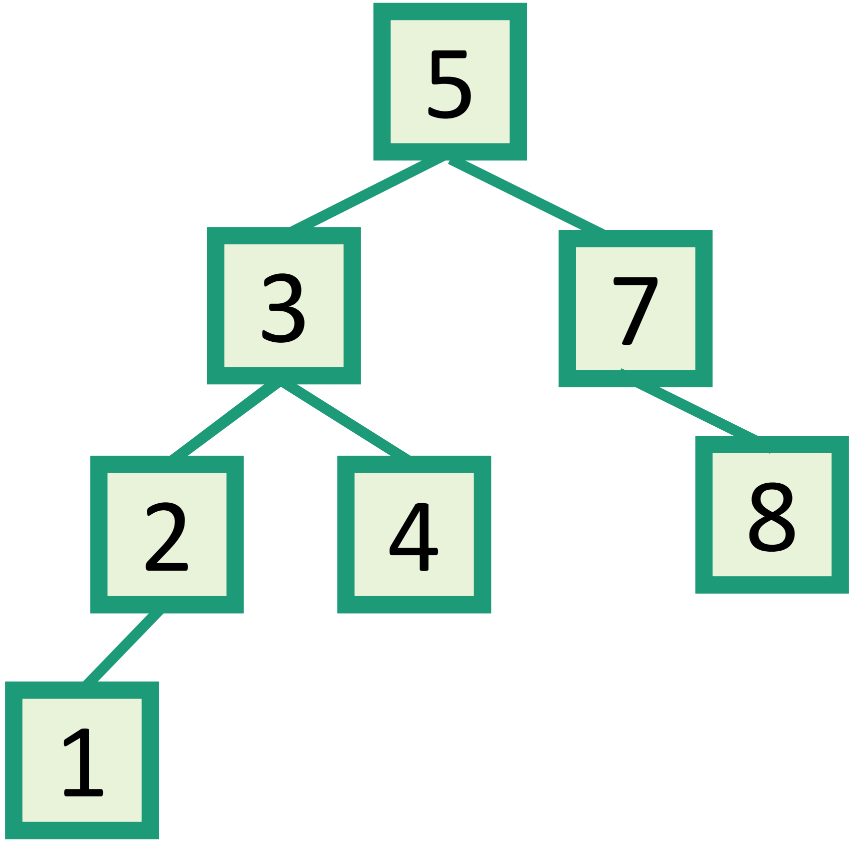


Do them in this order:



Another use of DFS that we've already seen

- In-order enumeration of binary search trees



Do DFS and print a node's label when you are done with the left child and before you begin the right child.

Acknowledgement

- Stanford University

Thank You