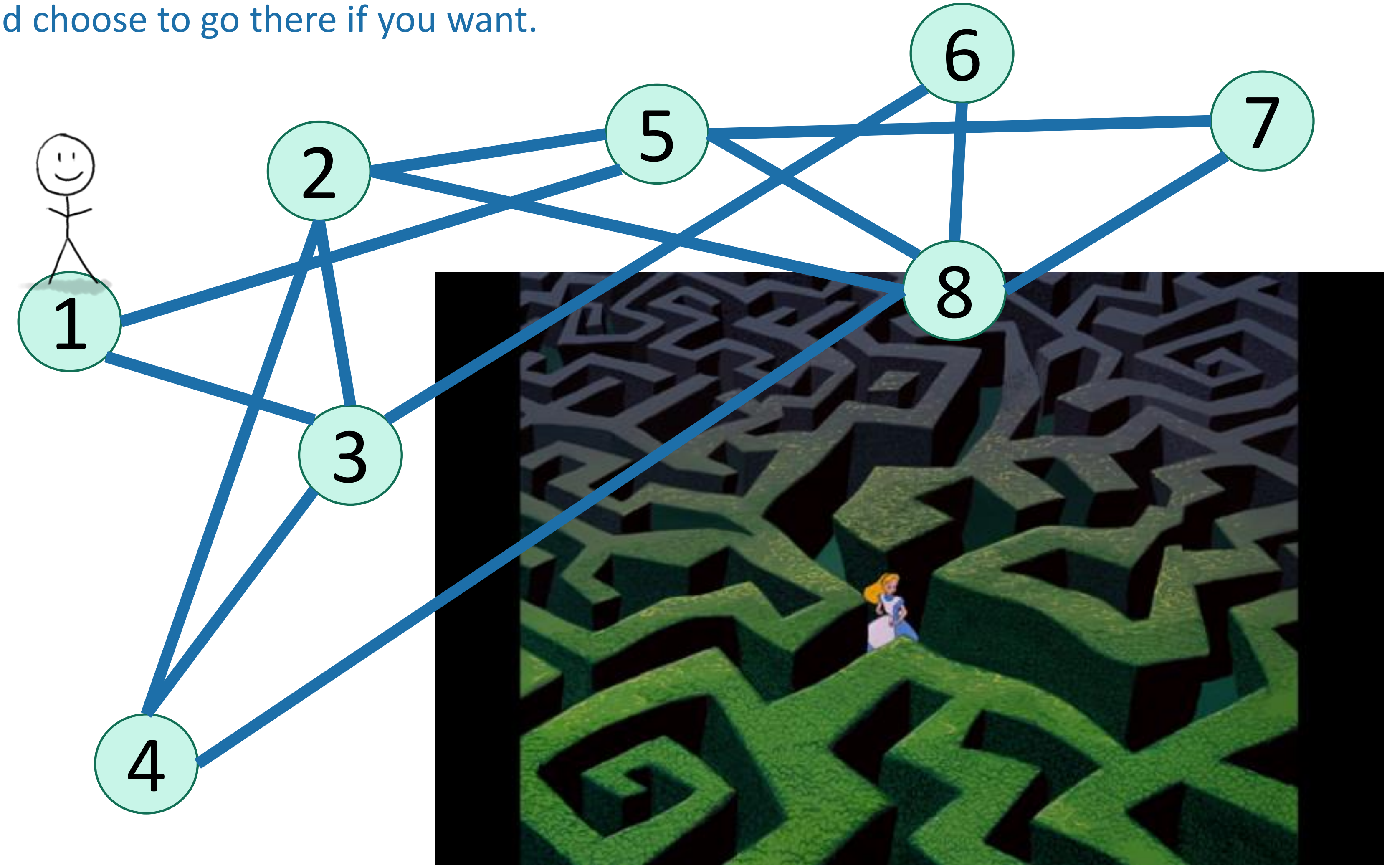
Advanced Data Structures and Algorithms

Depth First Search (DFS)

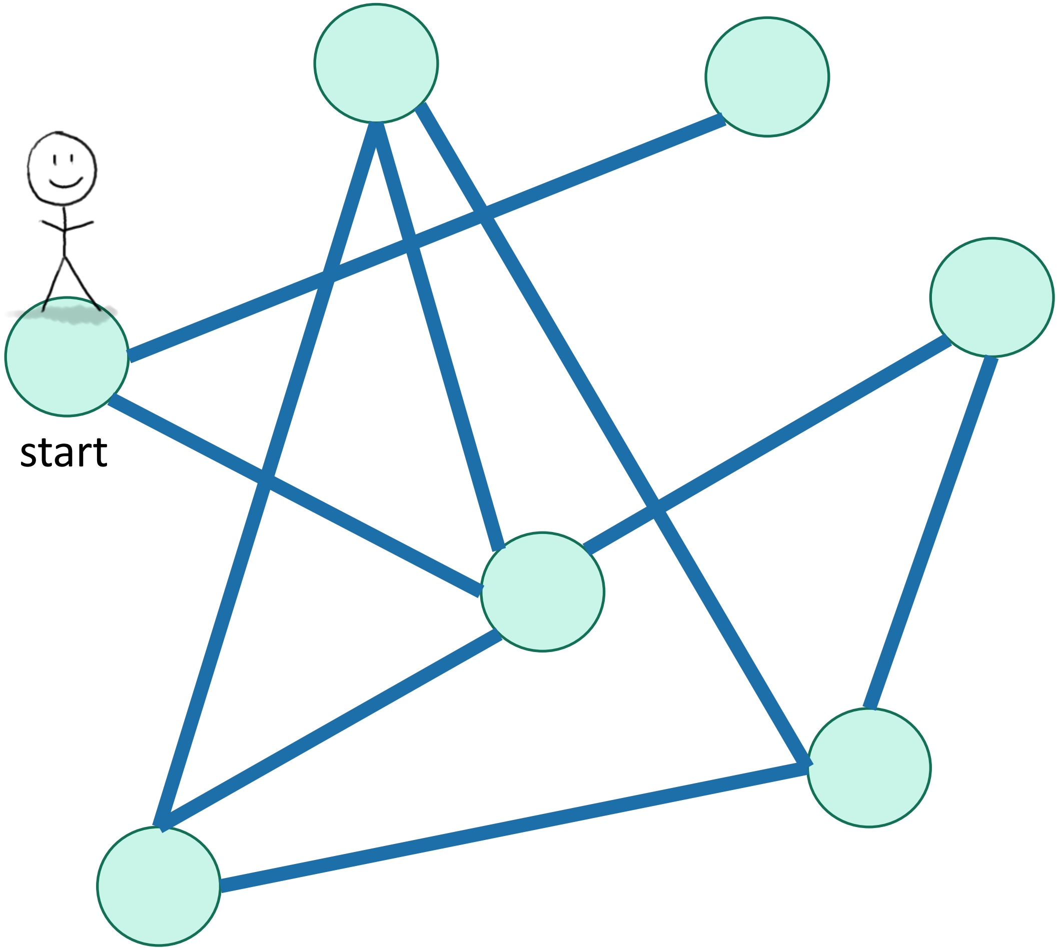
Depth-first search

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

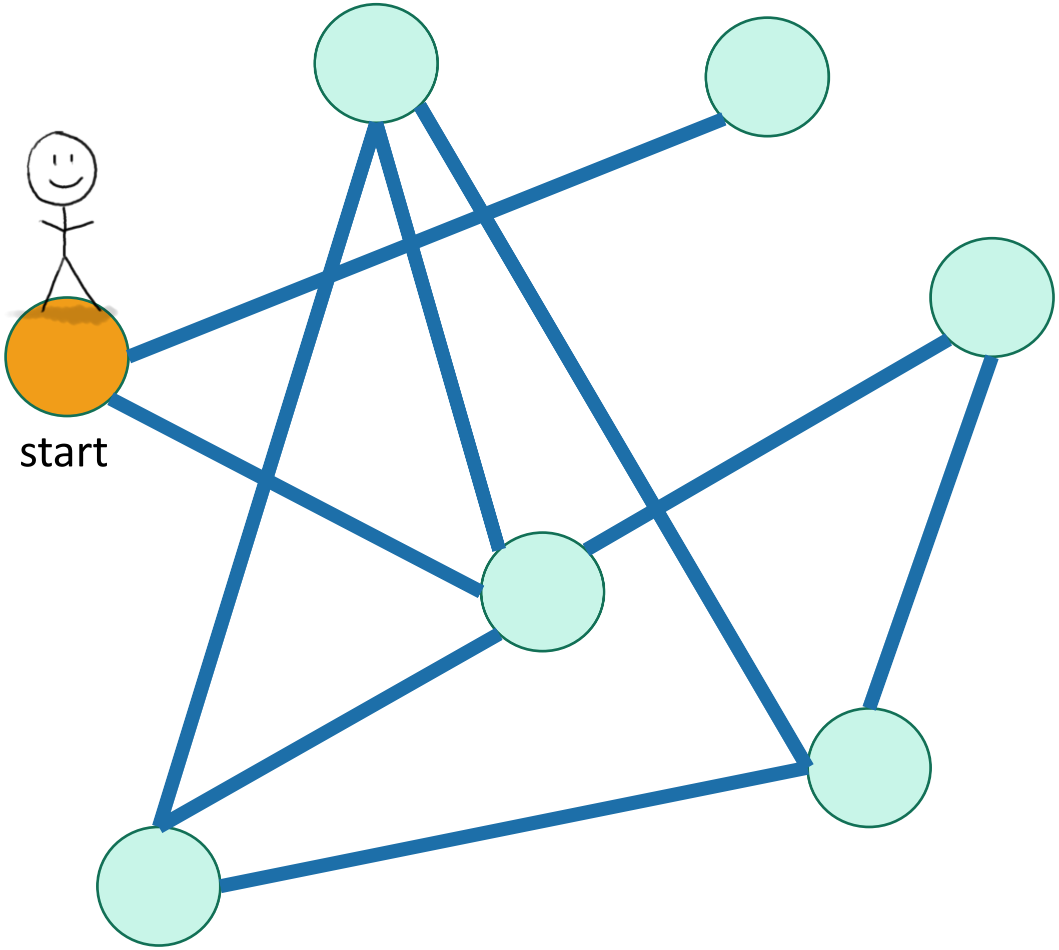


○ Not been there yet

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

🟢 Been there, have explored all the paths out.

Depth First Search

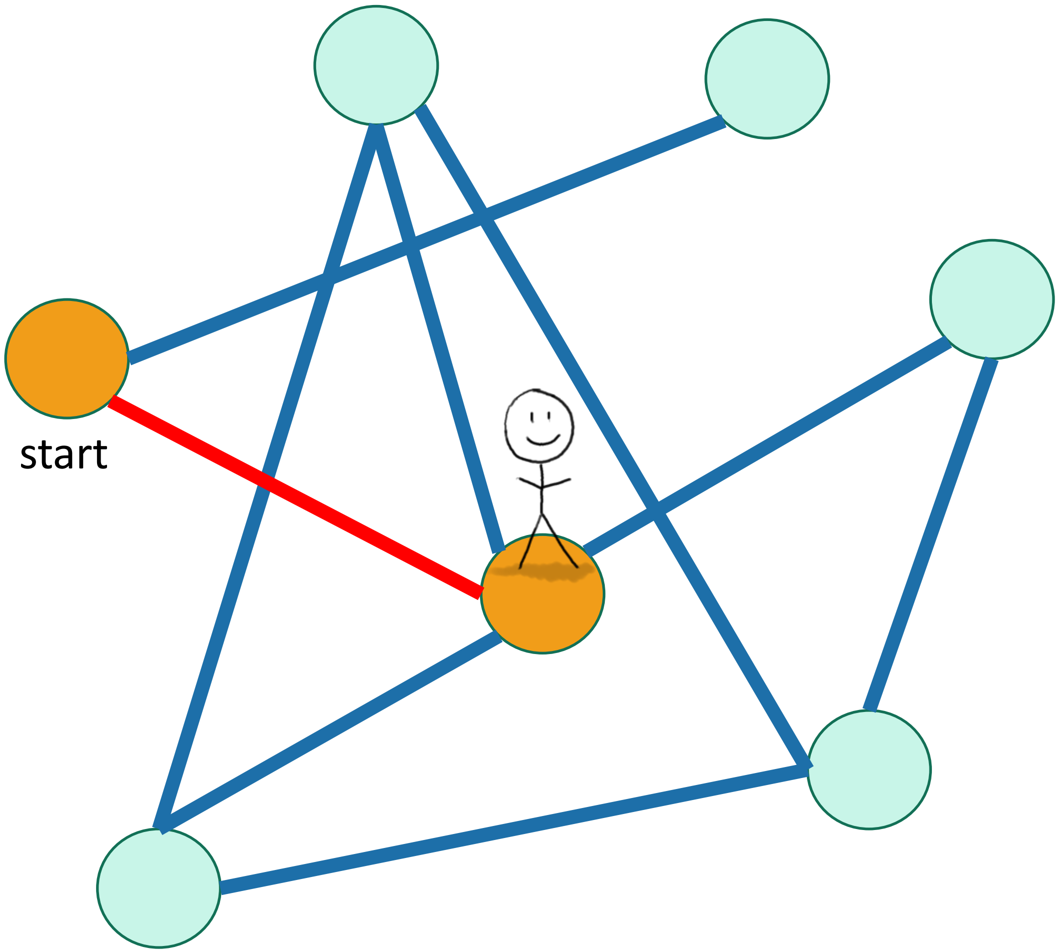


○ Not been there yet

🟠 Been there, haven’t explored all the paths out.

🟢 Been there, have explored all the paths out.

Depth First Search

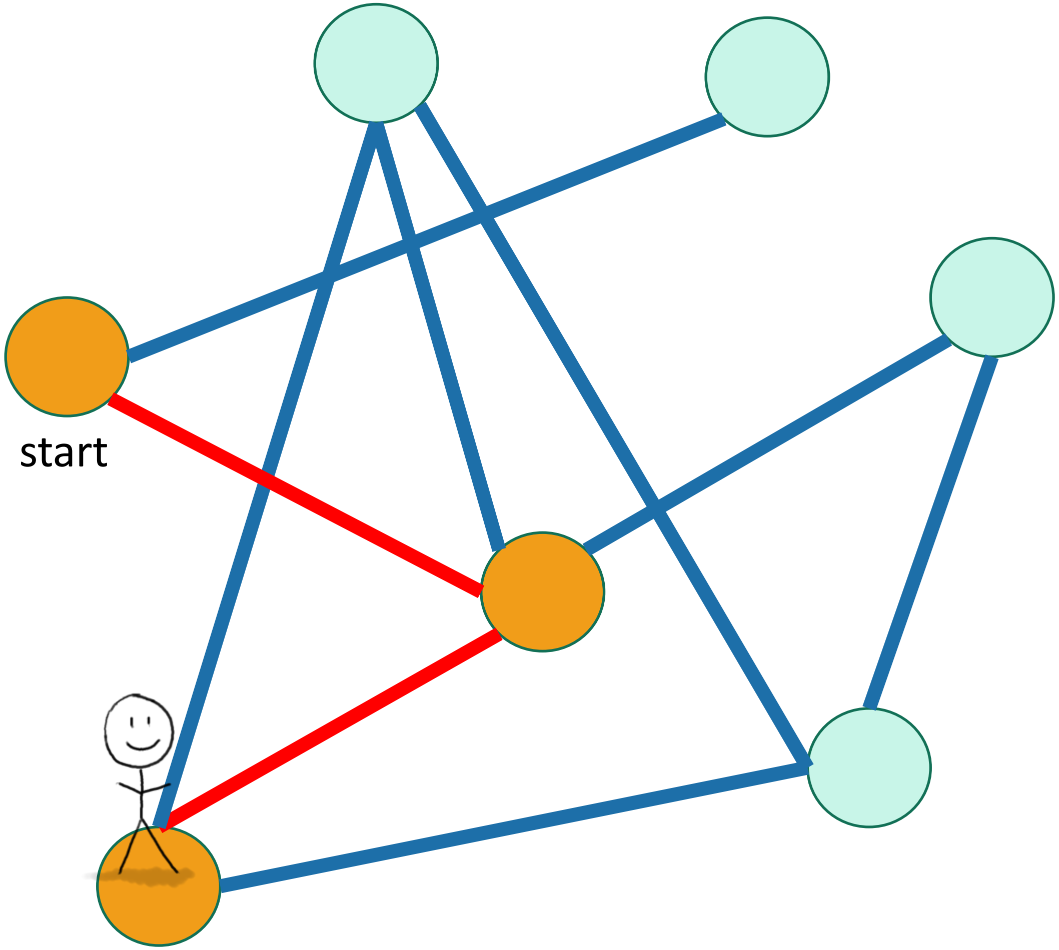


○ Not been there yet

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

🟢 Been there, have explored all the paths out.

Depth First Search

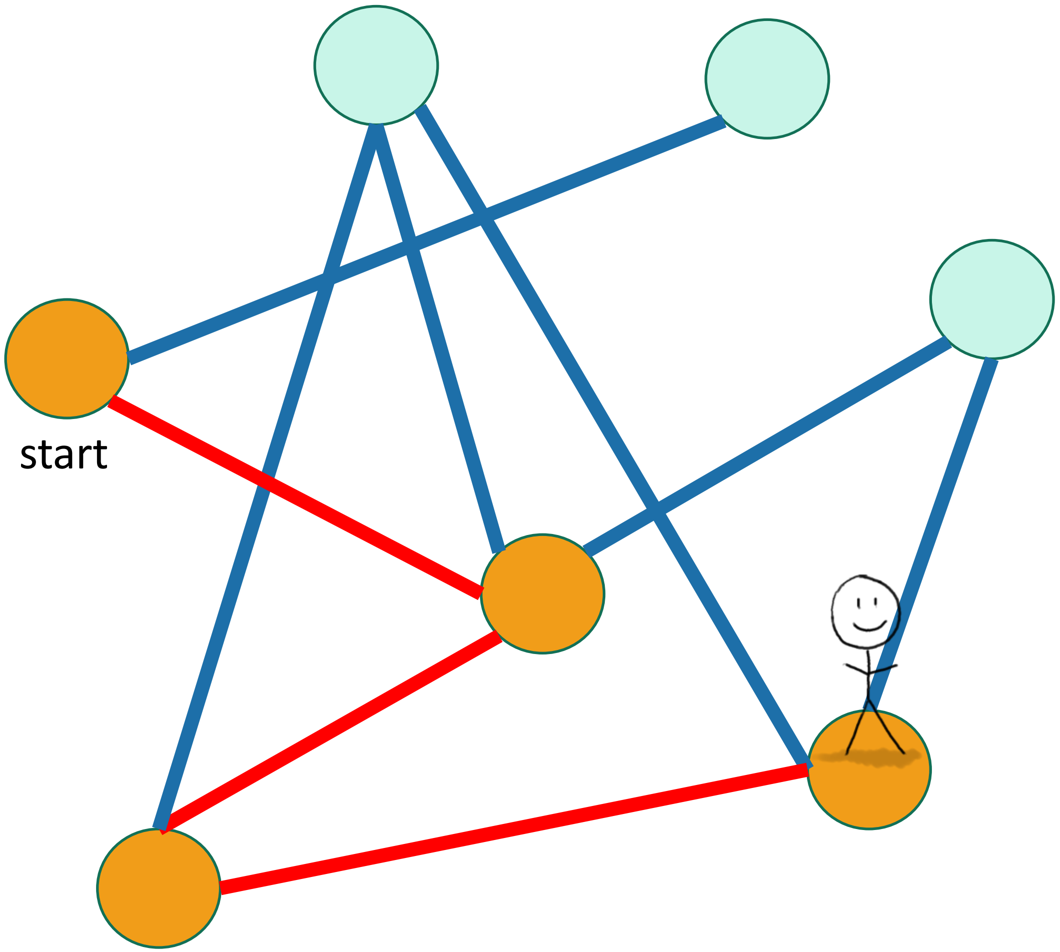


○ Not been there yet

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

● Been there, have explored all the paths out.

Depth First Search

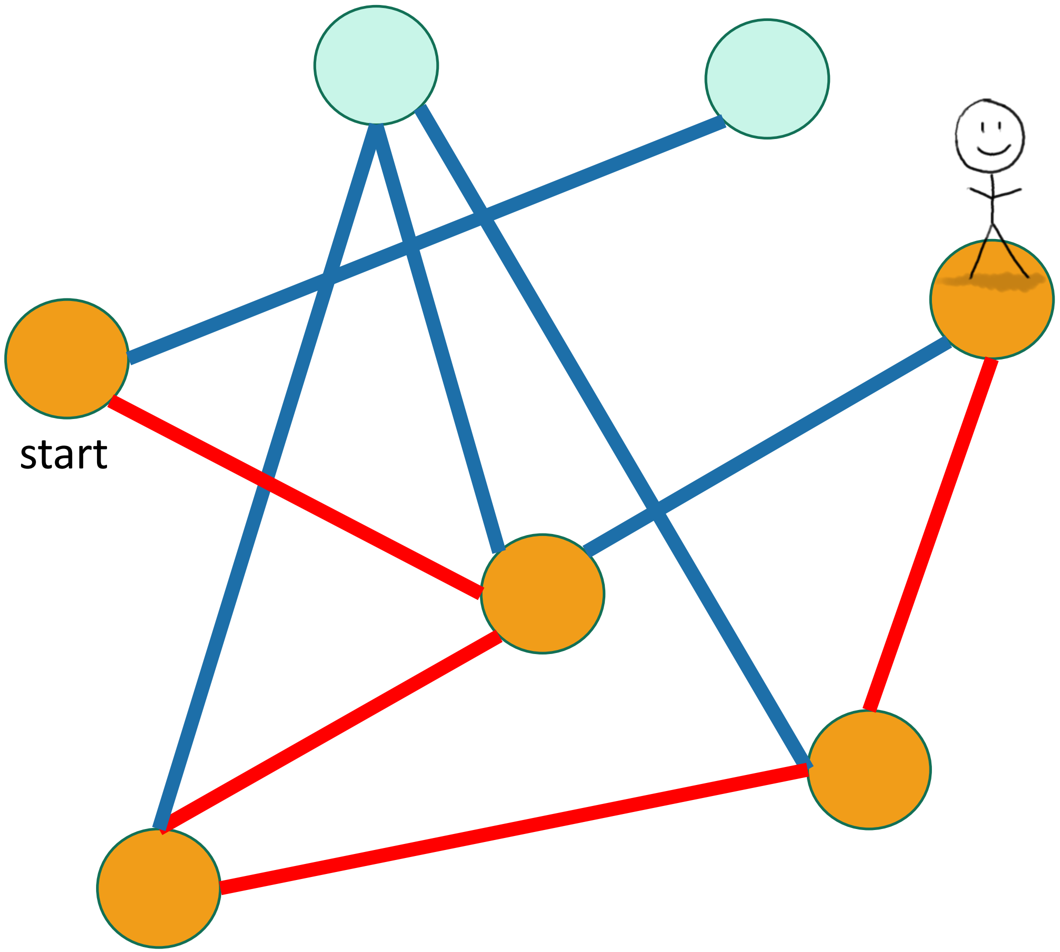


○ Not been there yet

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

🟢 Been there, have explored all the paths out.

Depth First Search

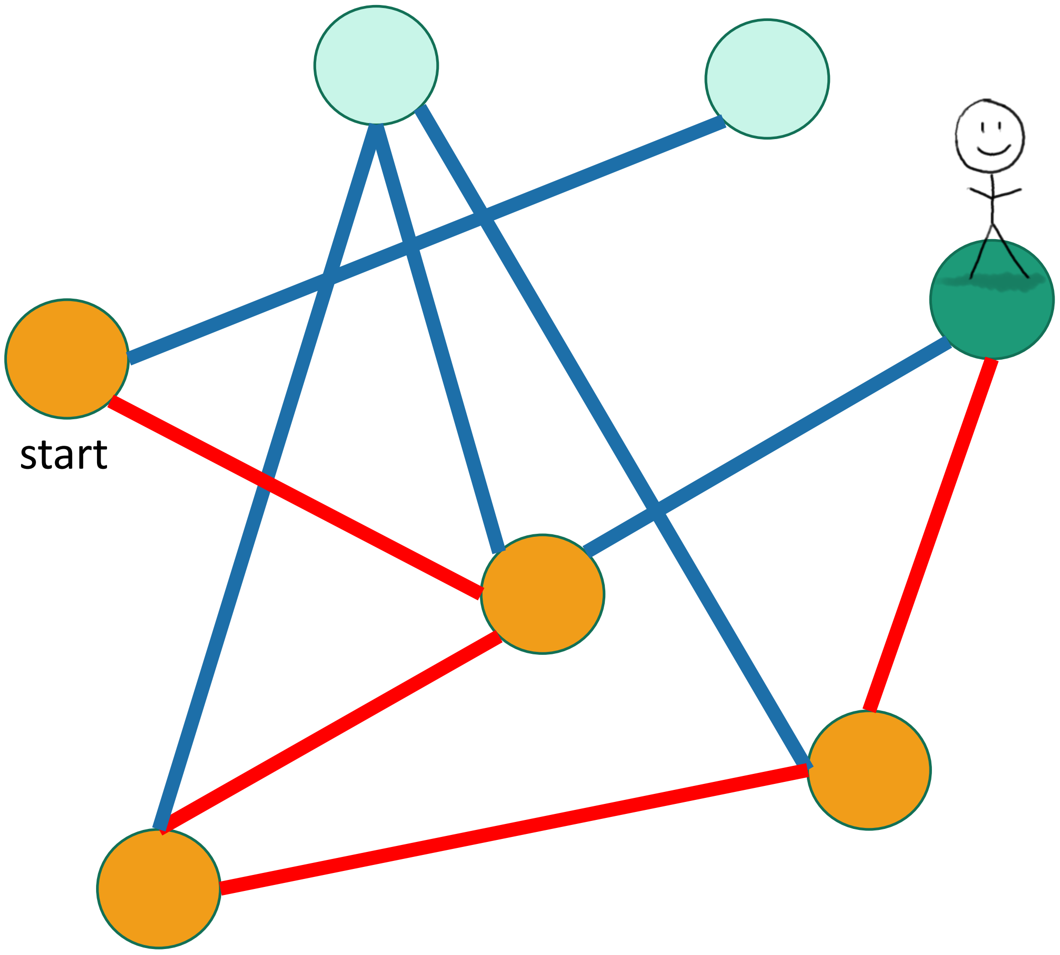


○ Not been there yet

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

🟢 Been there, have explored all the paths out.

Depth First Search

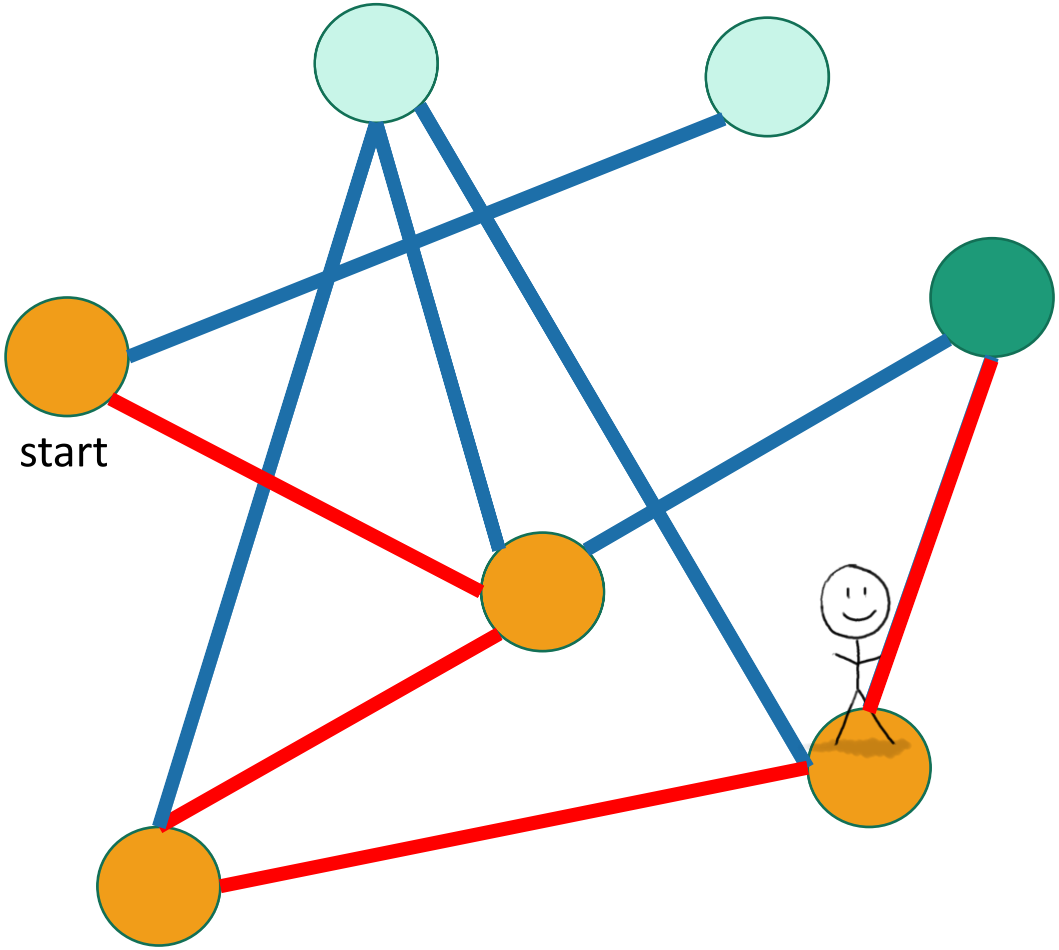


○ Not been there yet

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

🟢 Been there, have explored all the paths out.

Depth First Search

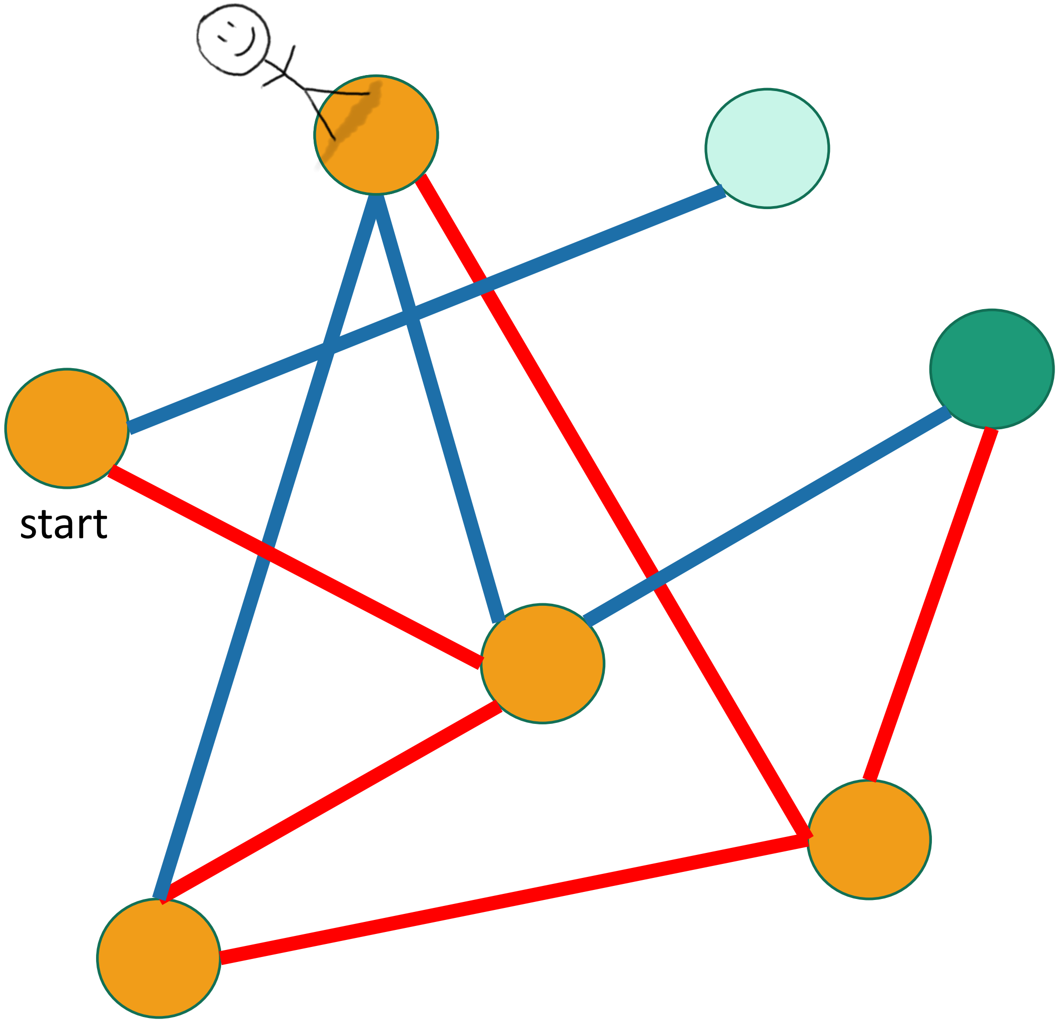


○ Not been there yet

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

🟢 Been there, have explored all the paths out.

Depth First Search

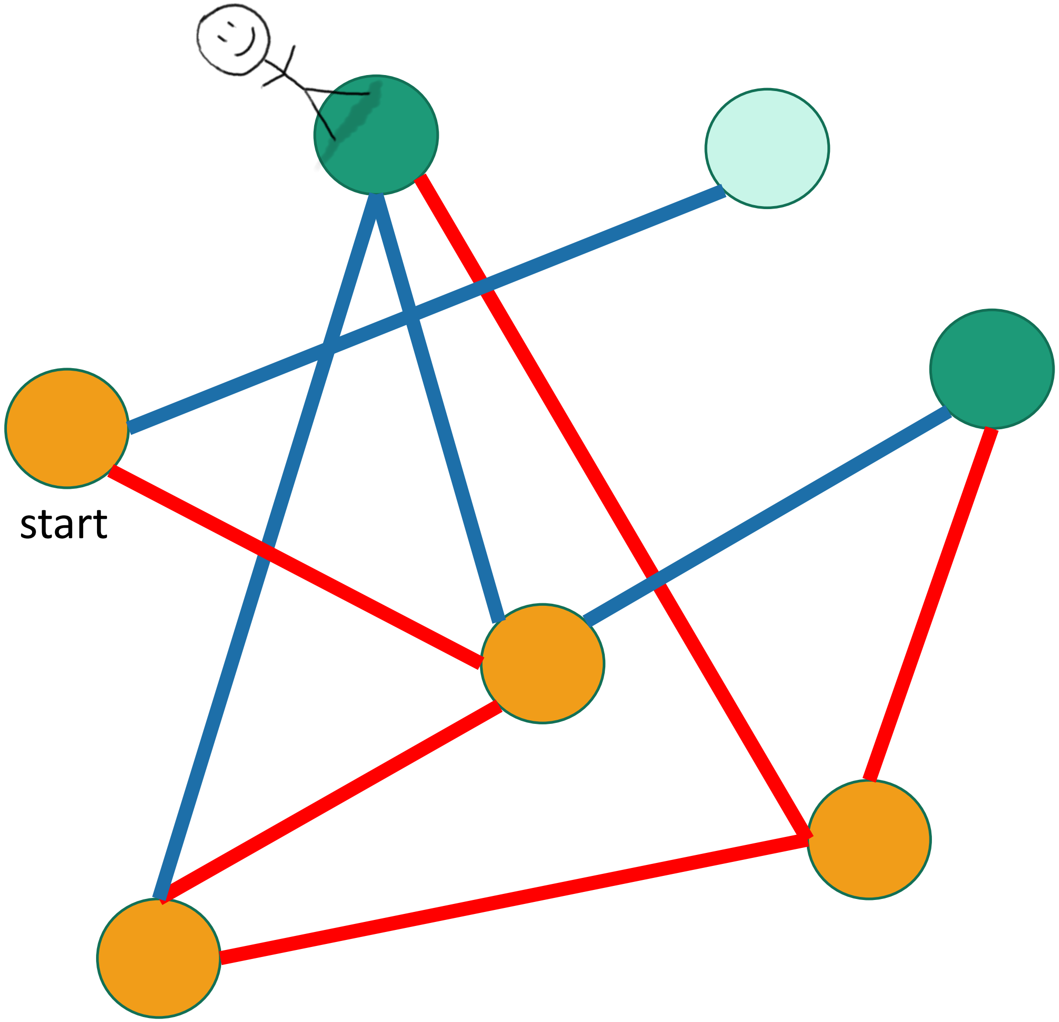


○ Not been there yet

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

🟢 Been there, have explored all the paths out.

Depth First Search

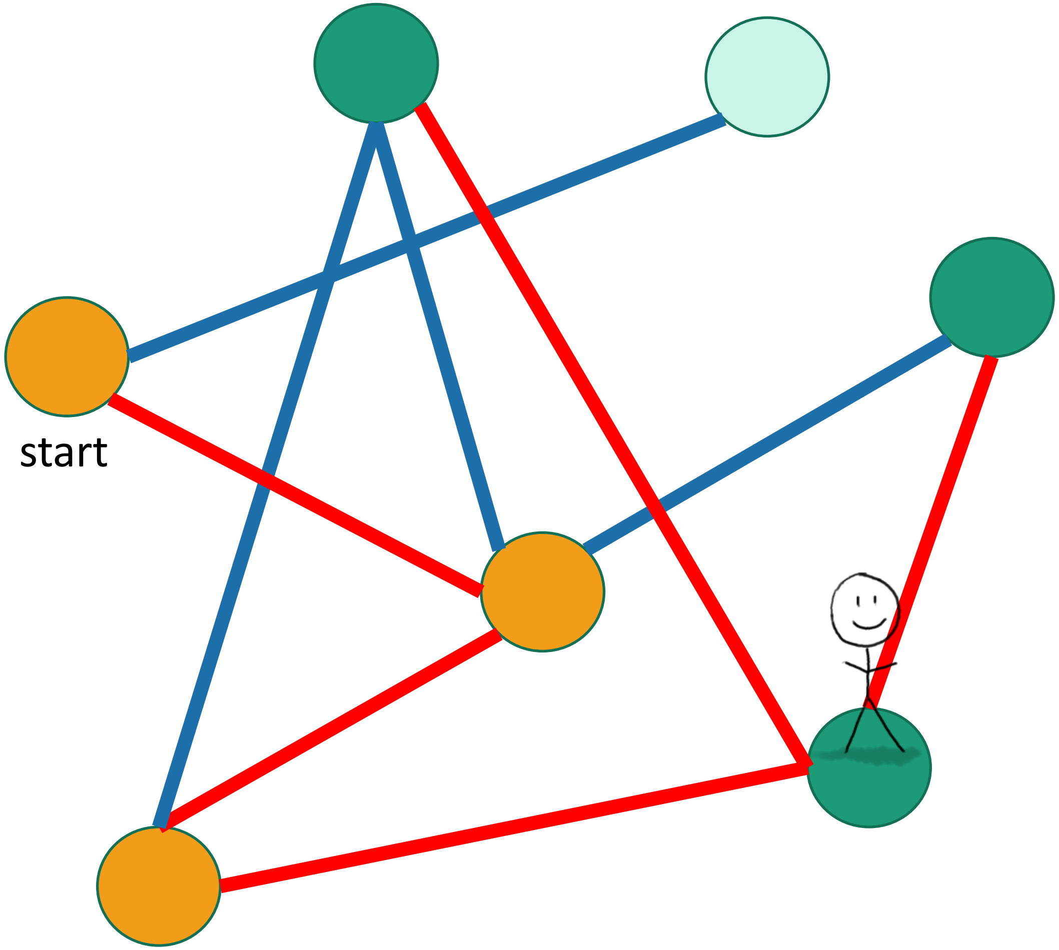


○ Not been there yet

🟠 Been there, haven’t explored all the paths out.

🟢 Been there, have explored all the paths out.

Depth First Search

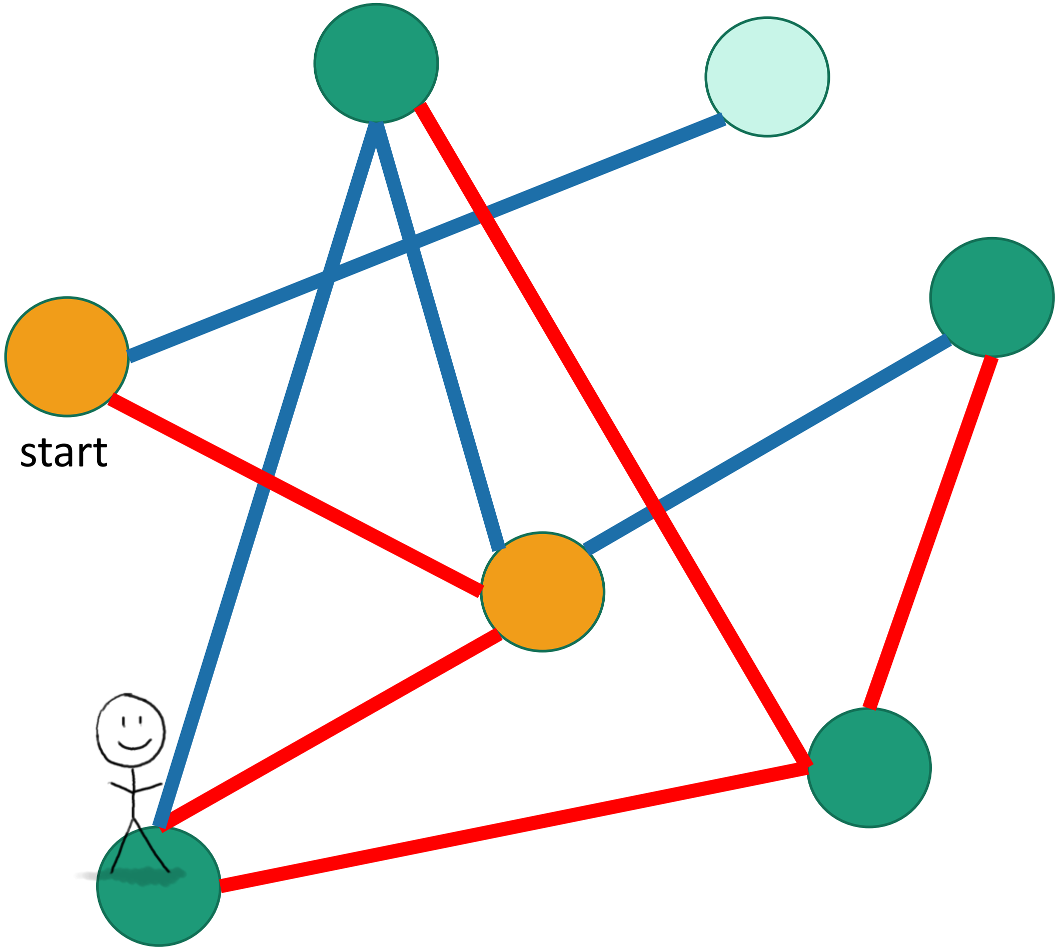


○ Not been there yet

● Been there, haven’t explored all the paths out.

● Been there, have explored all the paths out.

Depth First Search

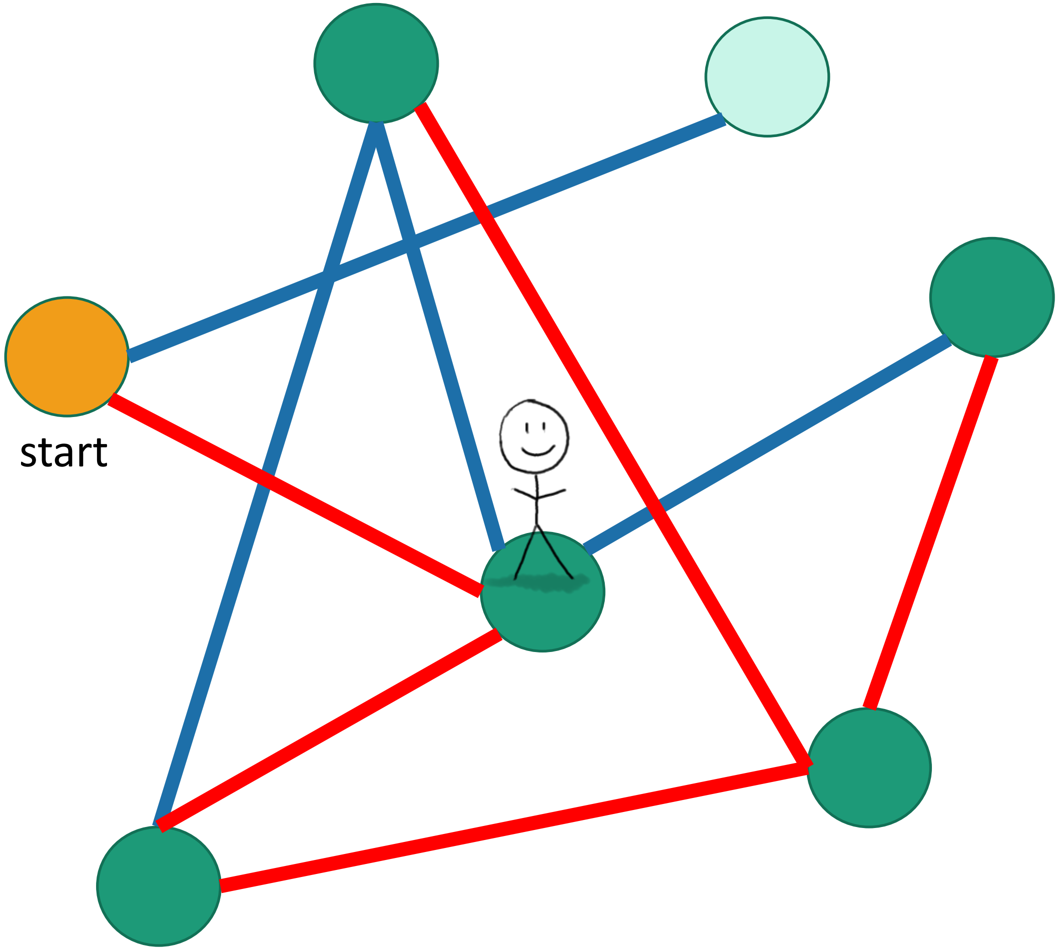


○ Not been there yet

🟠 Been there, haven’t explored all the paths out.

🟢 Been there, have explored all the paths out.

Depth First Search

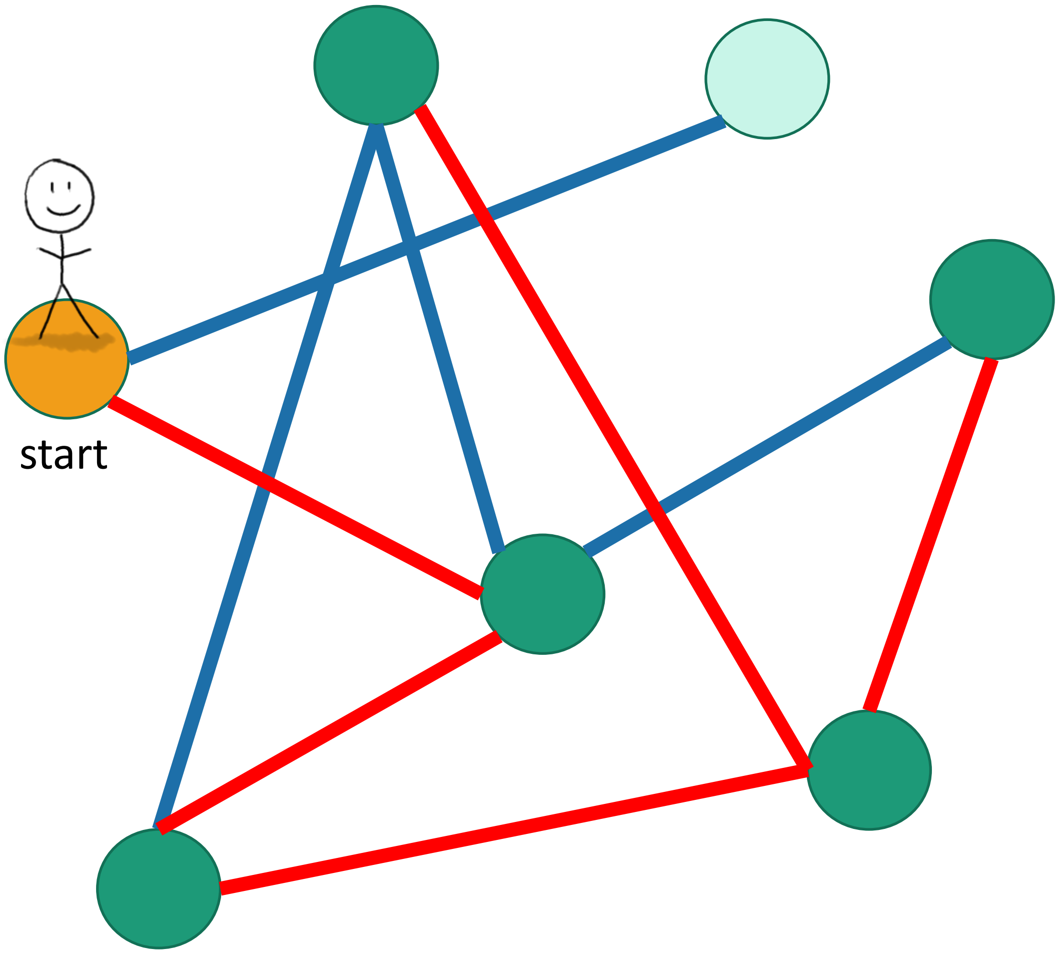


○ Not been there yet

🟠 Been there, haven’t explored all the paths out.

🟢 Been there, have explored all the paths out.

Depth First Search

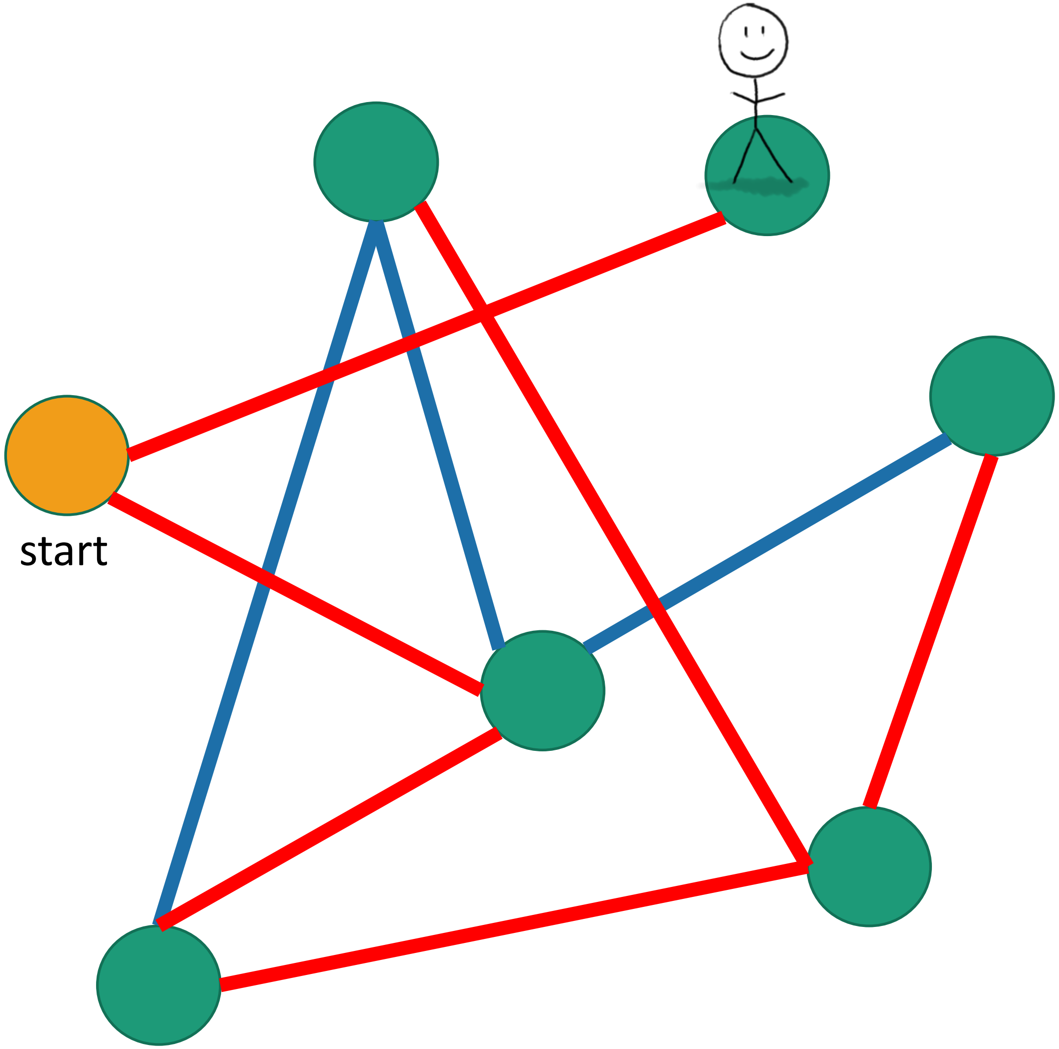


○ Not been there yet

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

🟢 Been there, have explored all the paths out.

Depth First Search

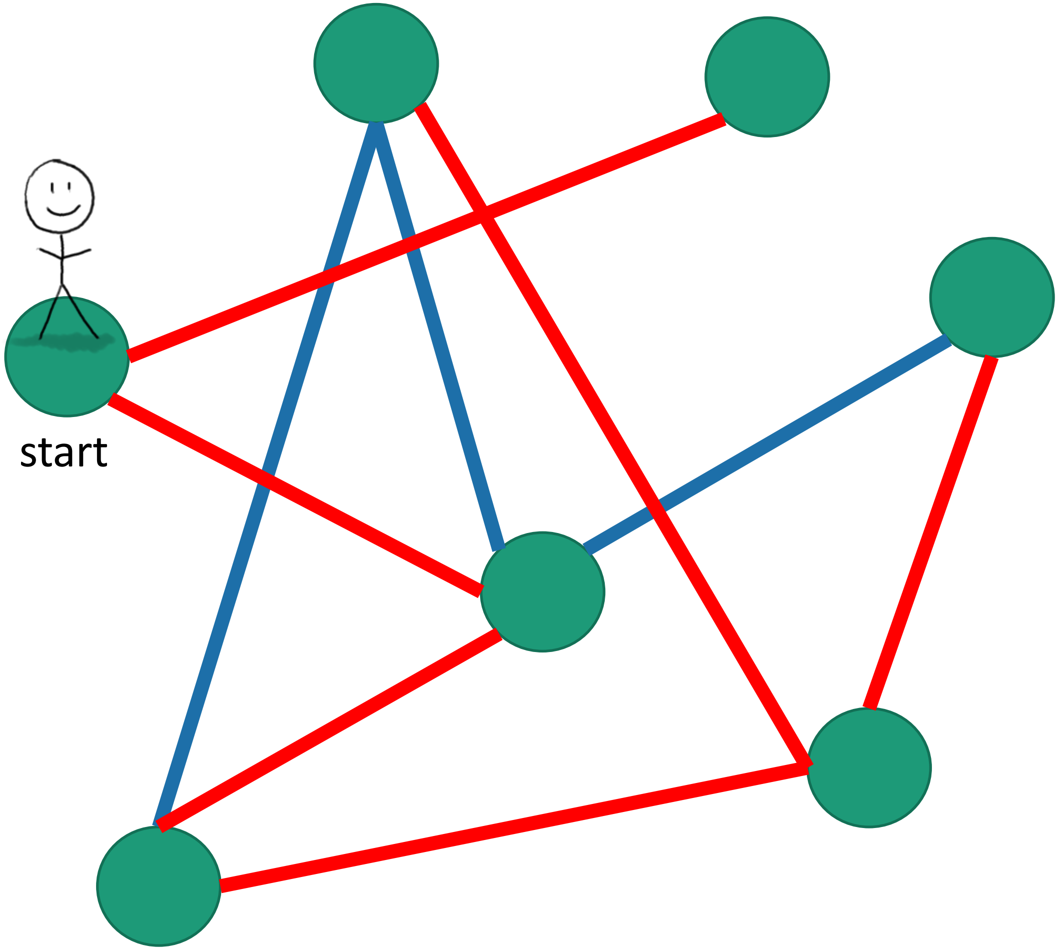


○ Not been there yet

● 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

Pseudocode

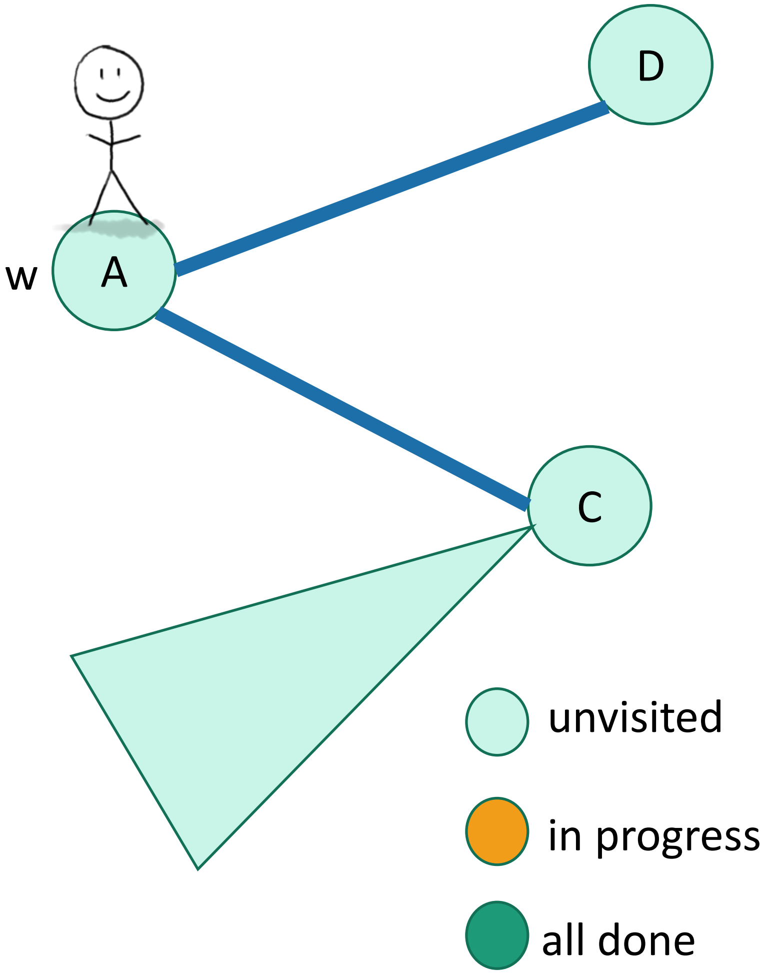
- Each vertex keeps track of whether it is:
    - Unvisited ○
    - In progress 🟠
    - All done 🟢

- 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

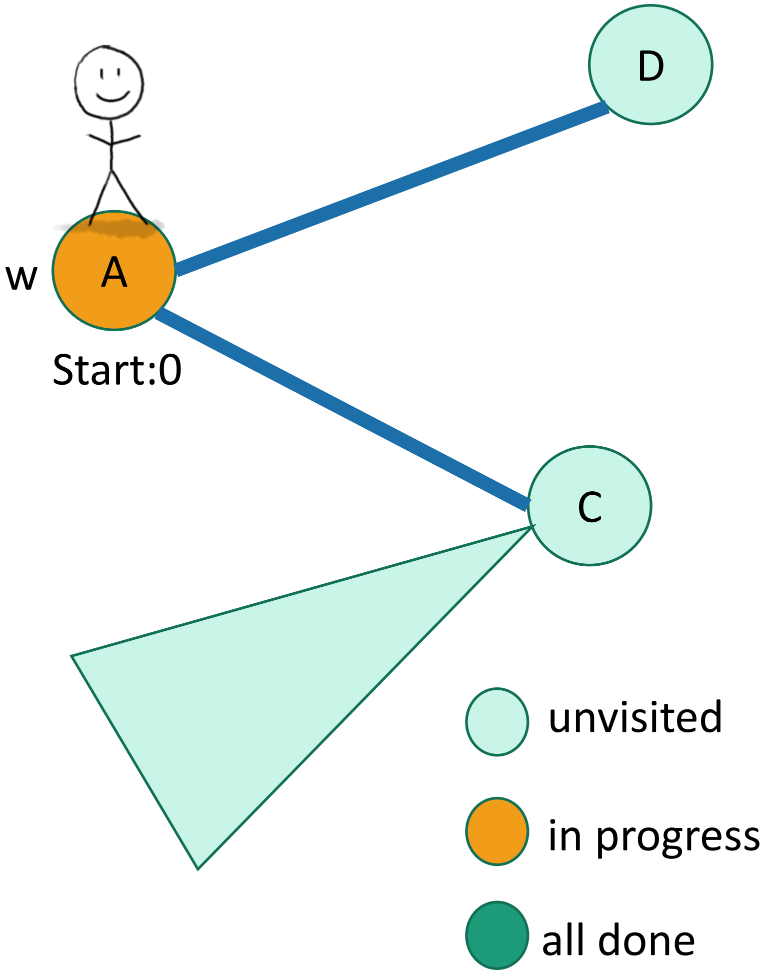
currentTime = 0



```
• 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
```

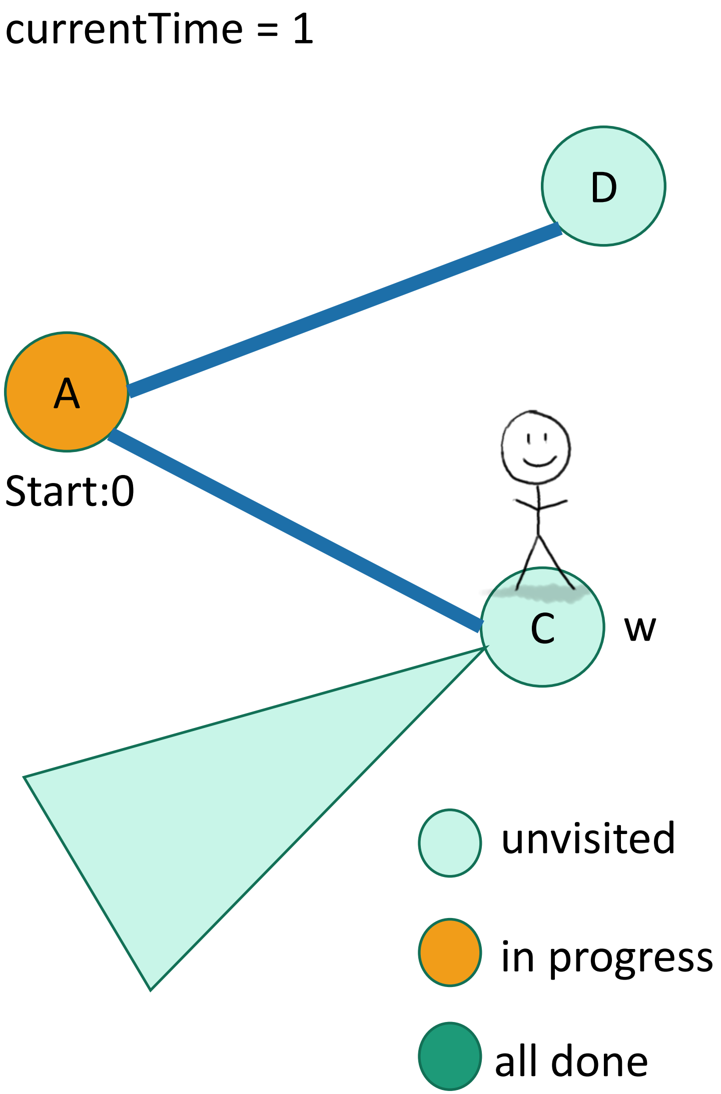
Depth First Search

currentTime = 1



```
• 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
```

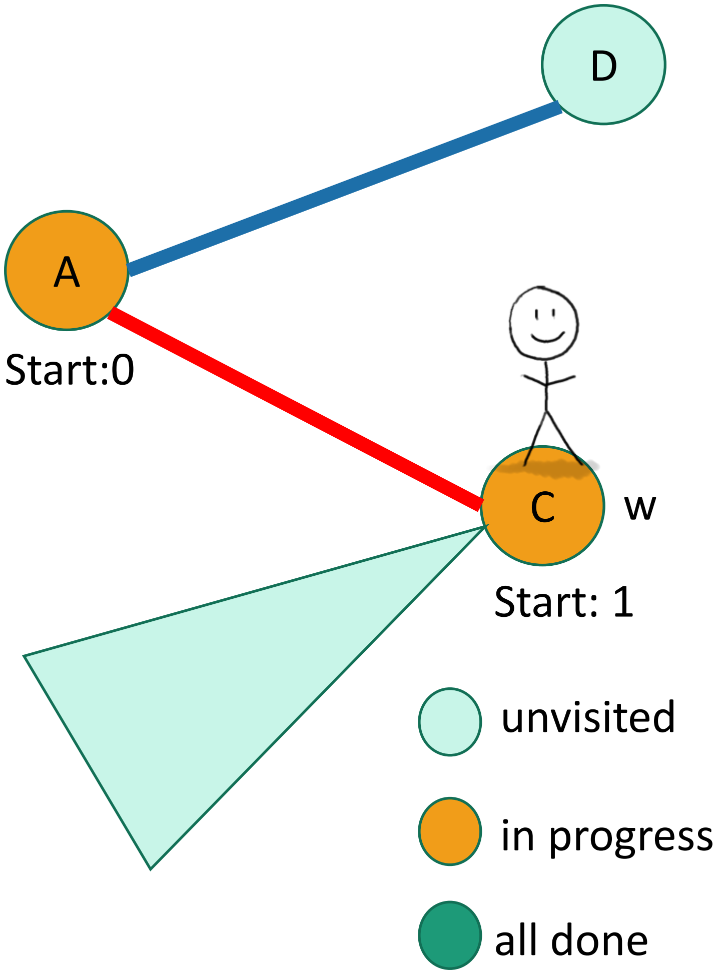
Depth First Search



```
• 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

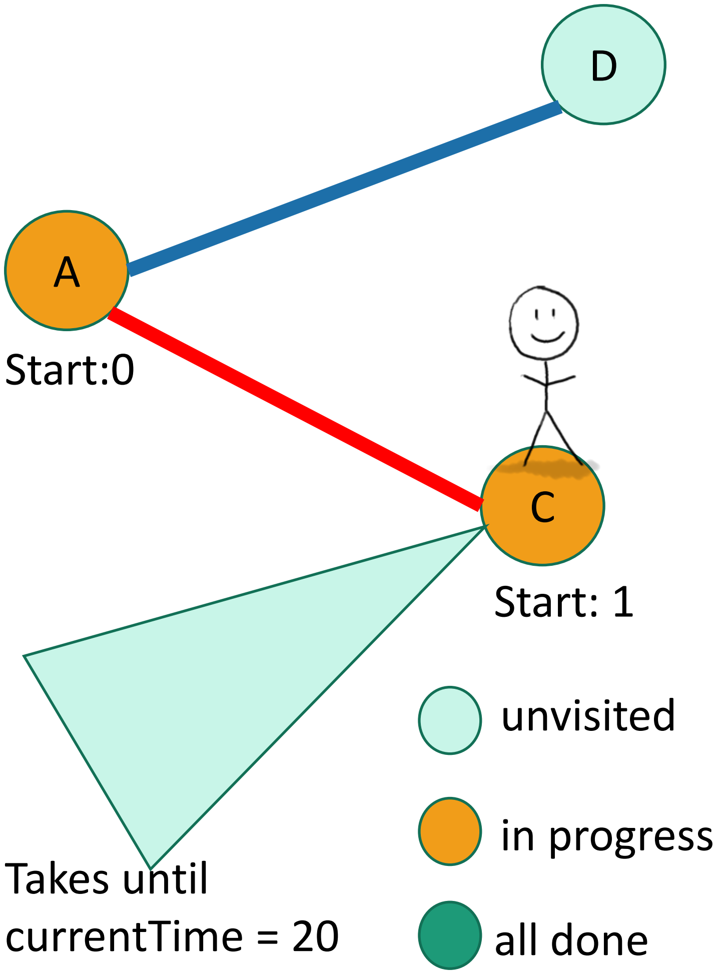
currentTime = 2



```
• 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

currentTime = 20



Start:0

Start: 1

Takes until currentTime = 20

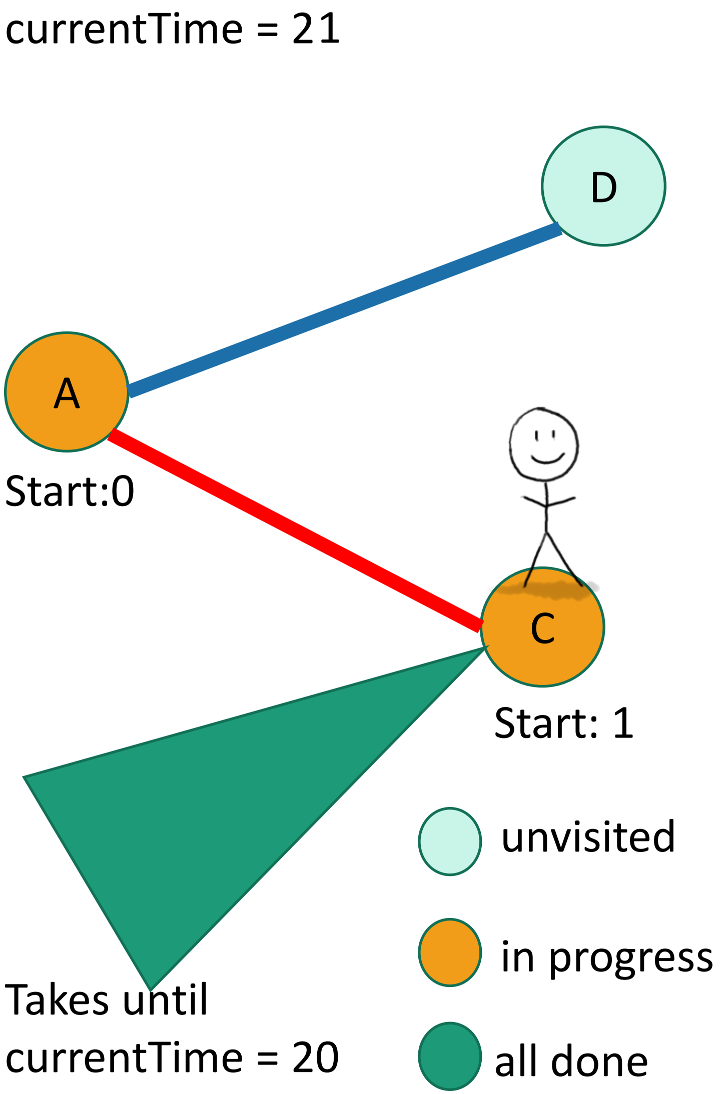
○ unvisited

● in progress

● all done

```
• 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



currentTime = 21

Takes until currentTime = 20

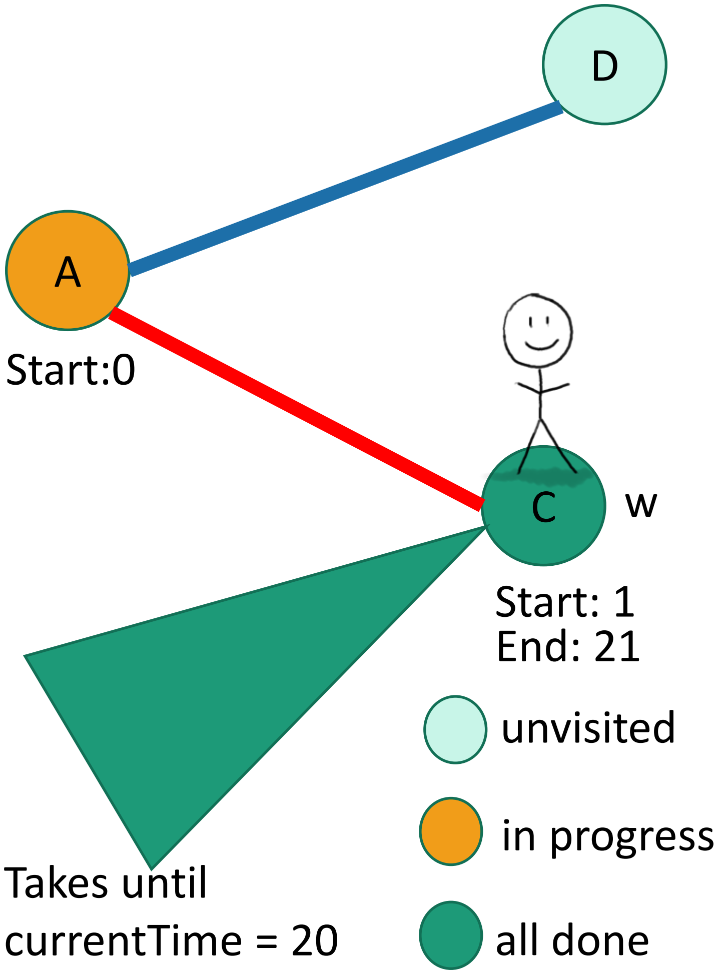
○ unvisited

● in progress

● all done

```
• 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



currentTime = 21

Start:0

Start: 1
End: 21

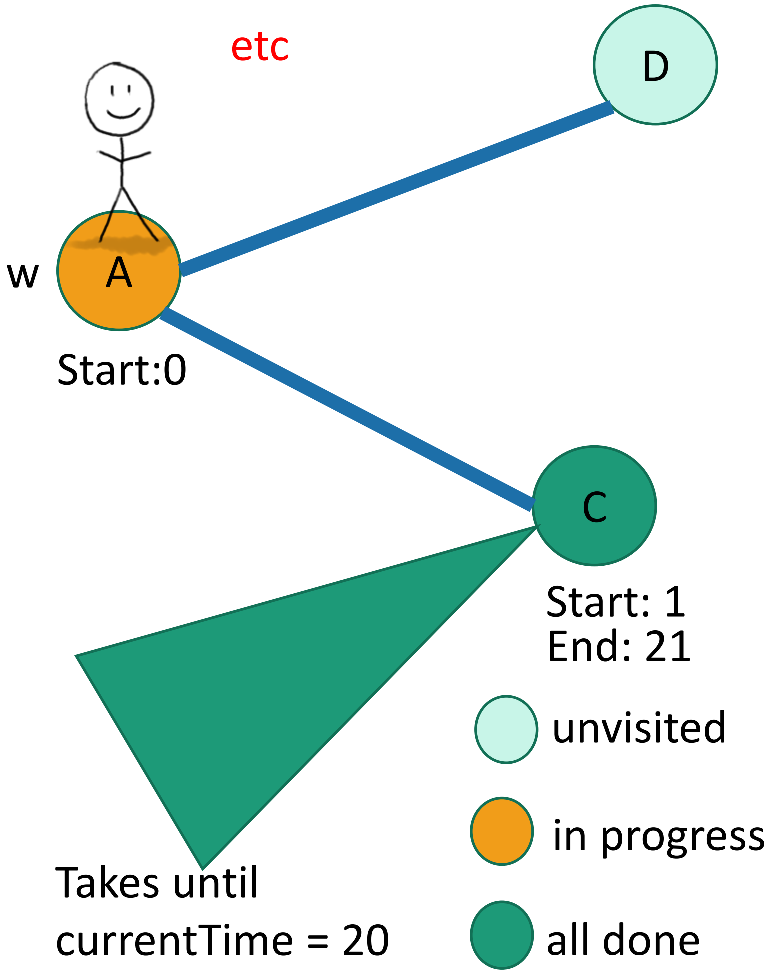
Takes until
currentTime = 20

- unvisited
- in progress
- all done

```
• 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

currentTime = 22

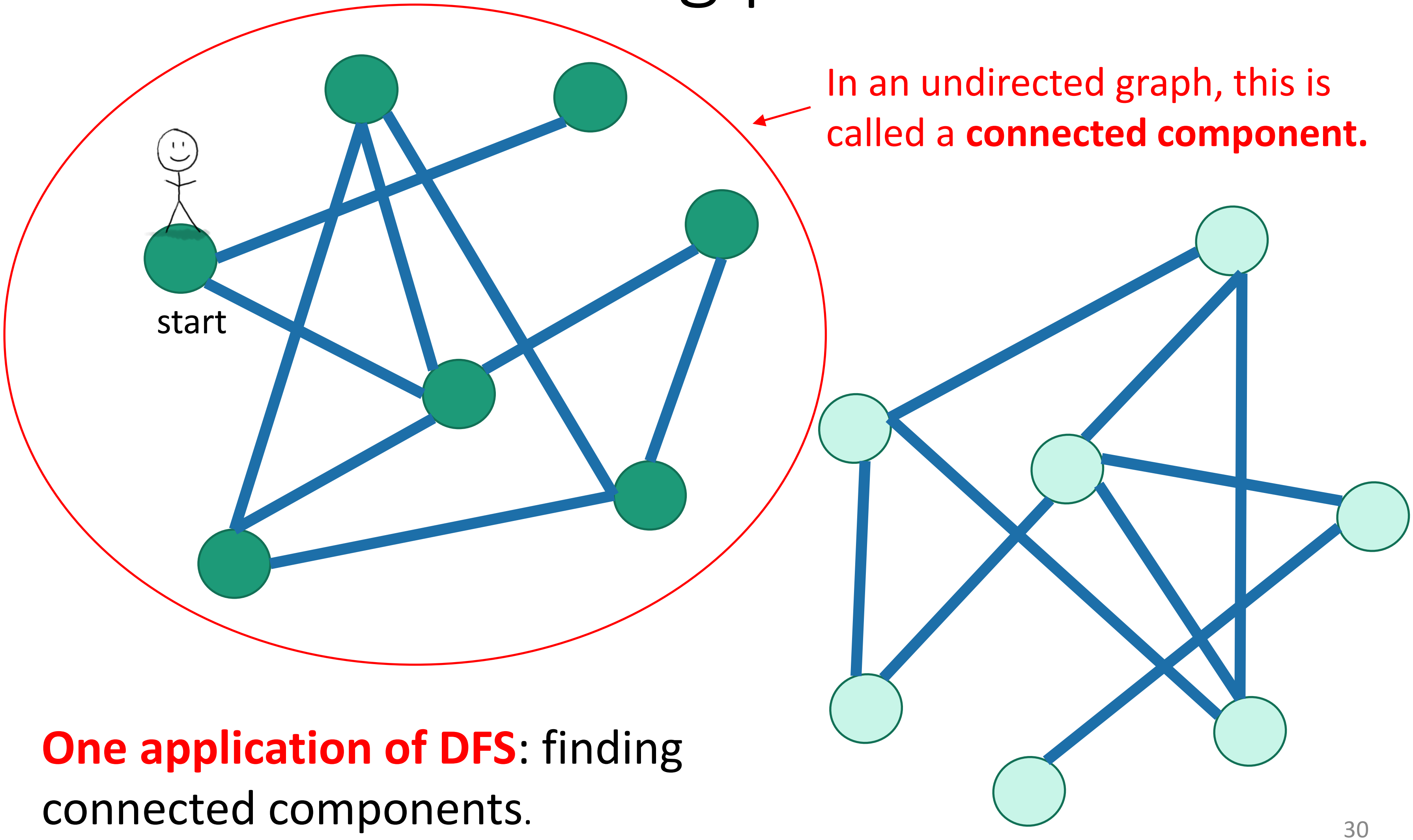


```
• DFS(w, currentTime):
    • w.startTime = currentTime
    • currentTime ++
    • Mark w as in progress.
    • for v in w.neighbors:
        • if v is unvisited:
            • currentTime
                  = DFS(v, currentTime)
            • currentTime ++
    • w.finishTime = currentTime
    • Mark 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

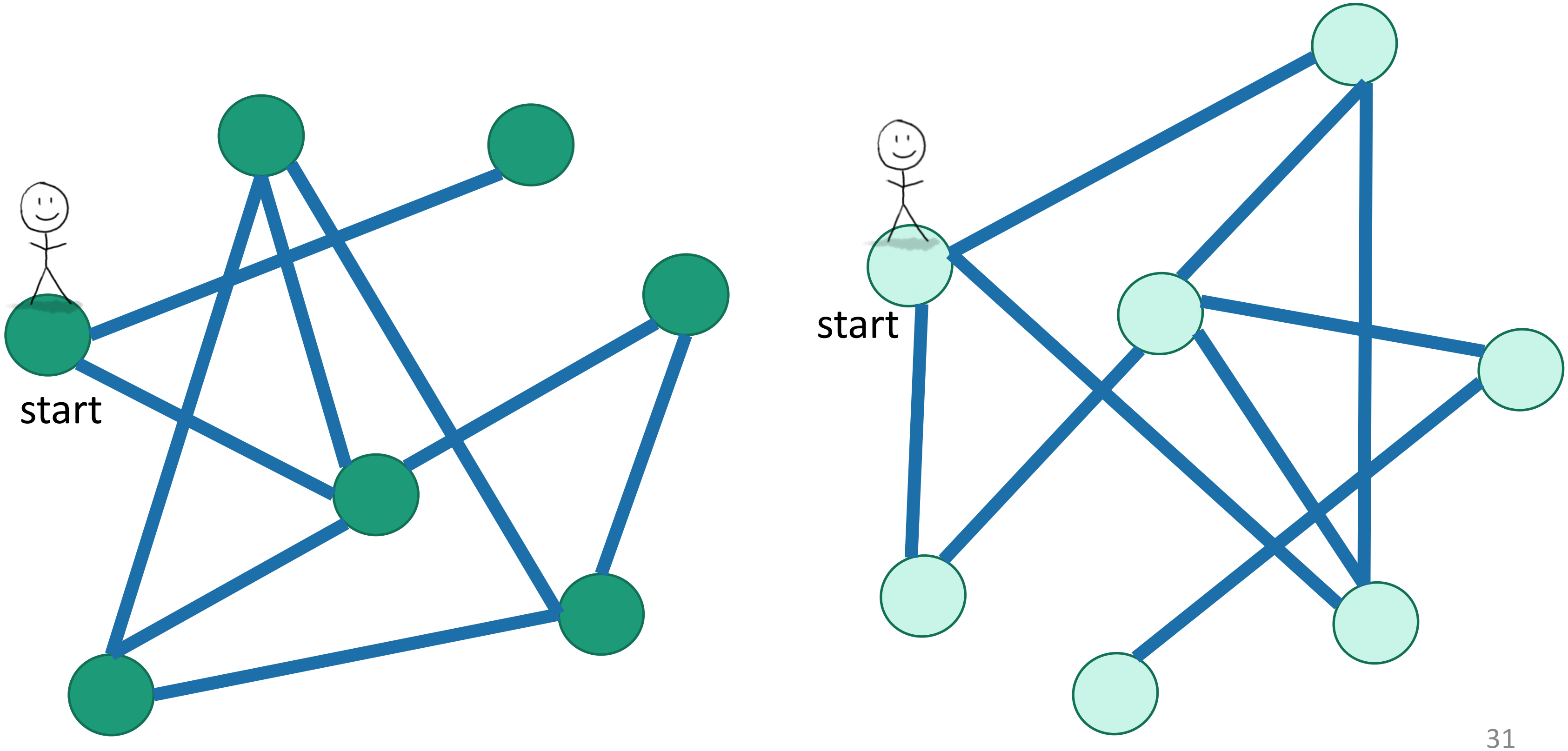


In an undirected graph, this is called a **connected component.**

One application of DFS: finding connected components.

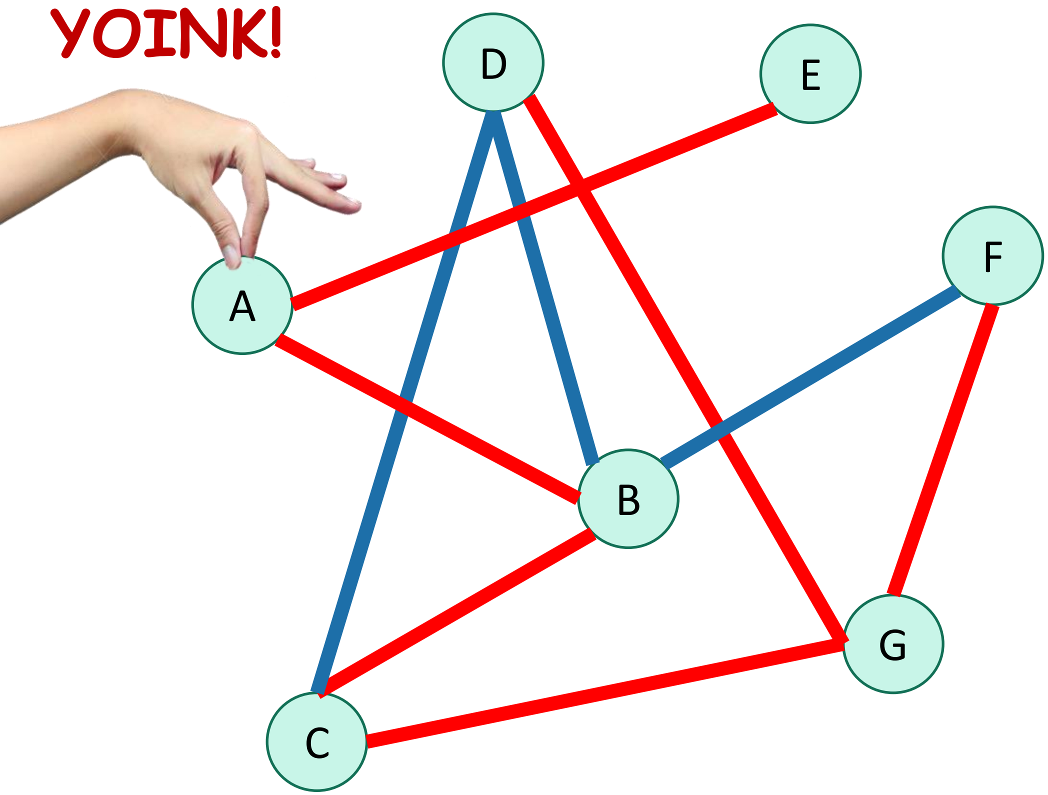
To explore the whole graph

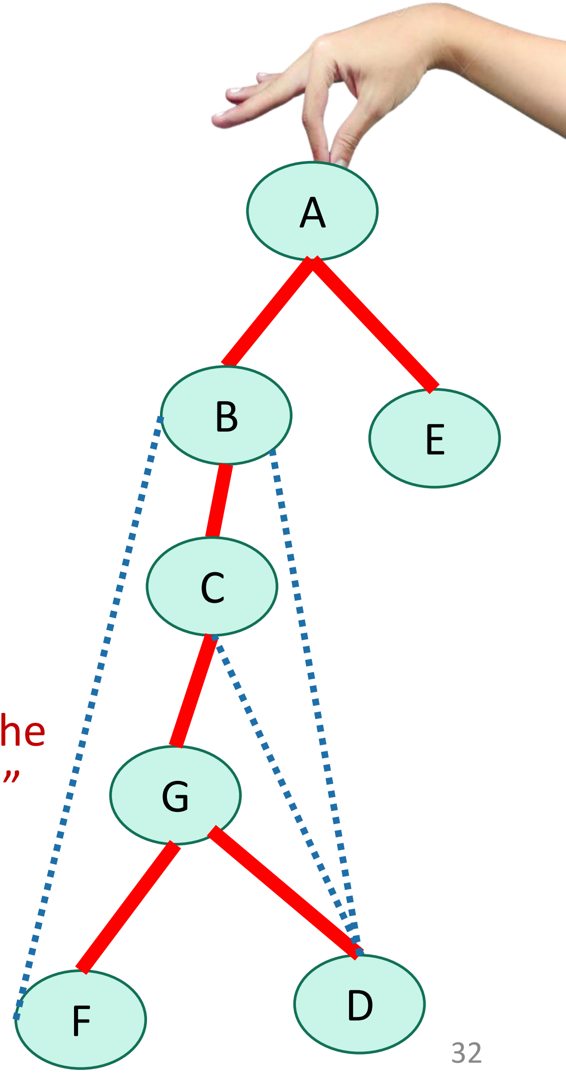
- Do it repeatedly!



Why is it called depth-first?

- We are implicitly building a tree:





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…

$$O(m)$$

Running time

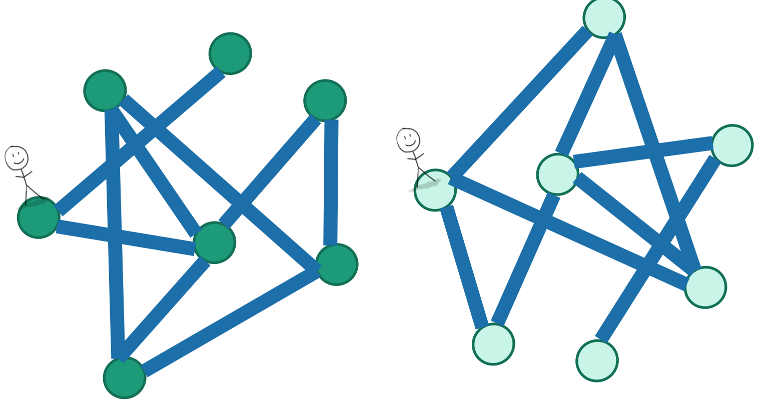
To explore just the connected component we started in

- 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"
- 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, $|V'| \leq |E'| + 1$.

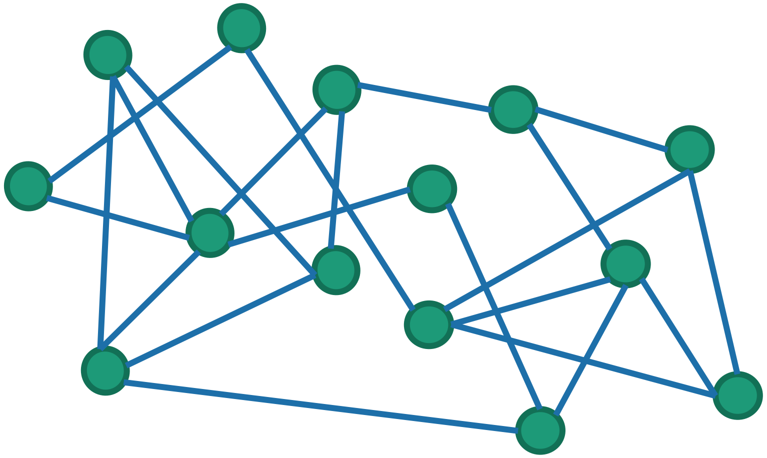
Running time

To explore the whole graph



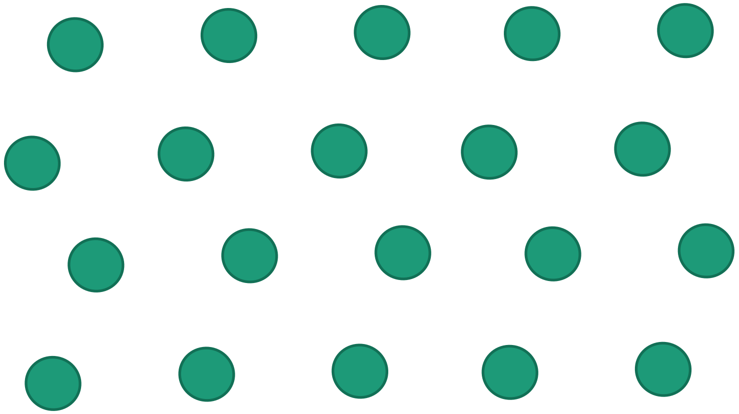
- Explore the connected components one-by-one.
- This takes time O(n + m)
  - Same computation as before:

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



Here the running time is O(m) like before

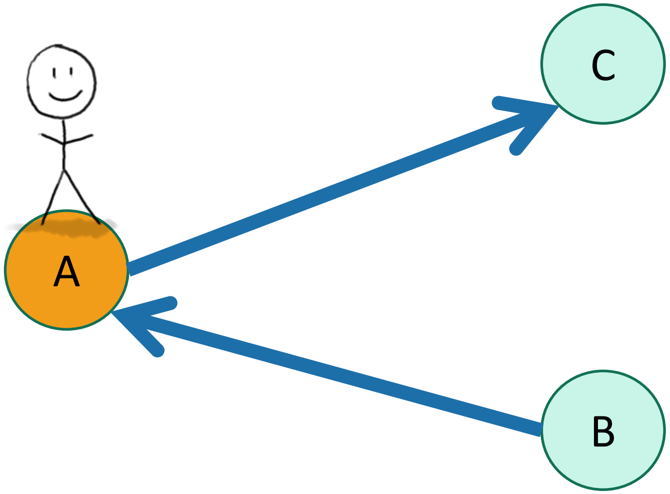
or



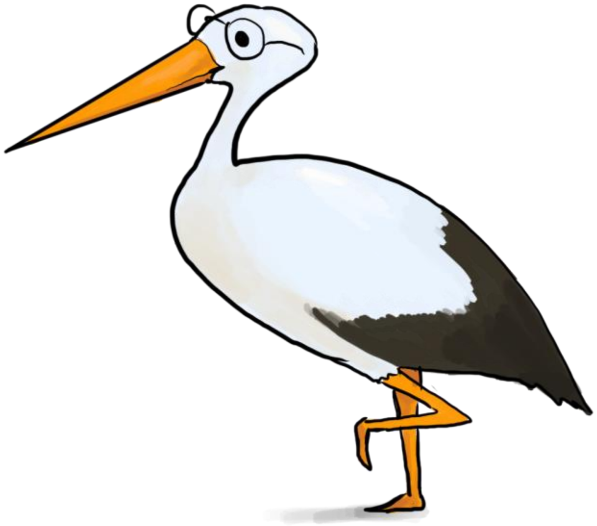
Here m=0 but it still takes time O(n) to explore the graph.

You check:

DFS works fine on directed graphs too!

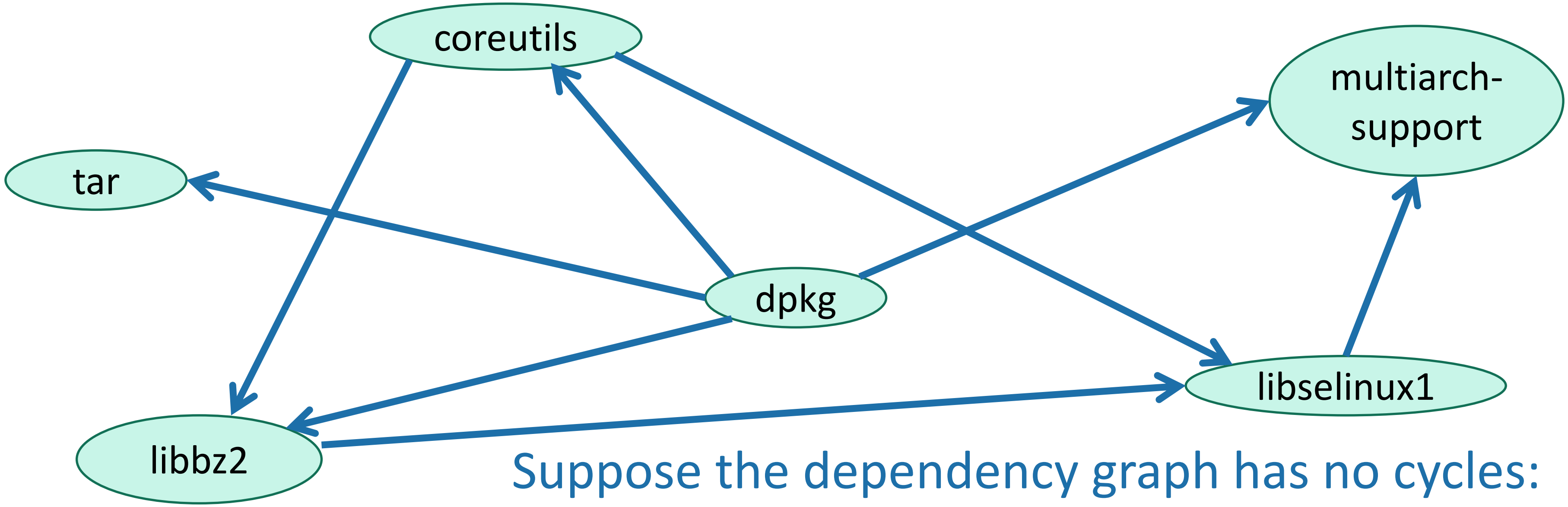


Only walk to C, not to B.



Application of DFS: topological sorting

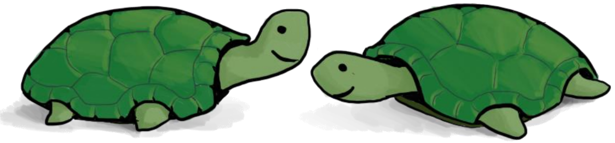
- Find an ordering of vertices so that all of the dependency requirements are met.
  - Aka, if v comes before w in the ordering, there is not an edge from w to v.

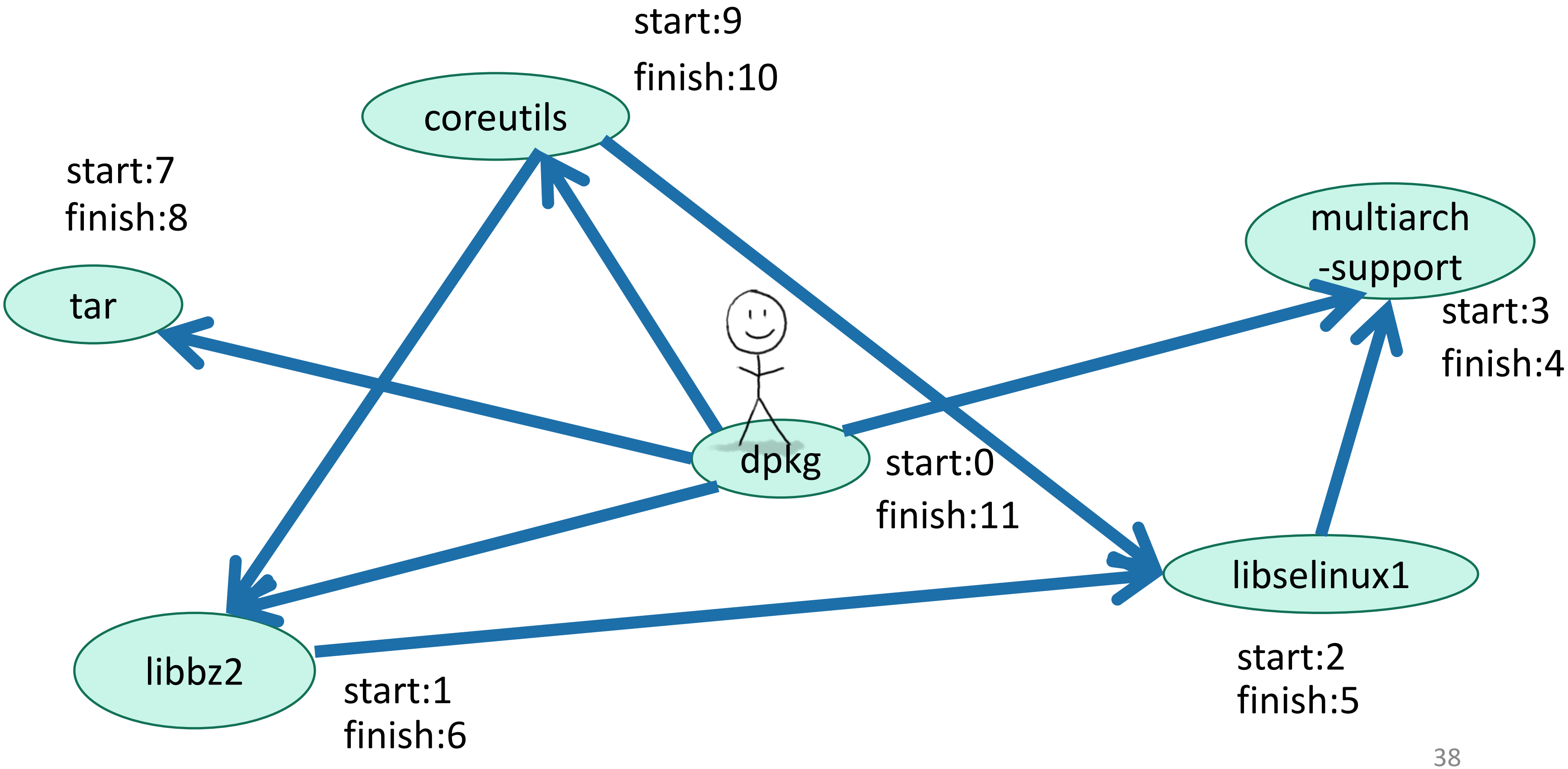


Suppose the dependency graph has no cycles: it is a **Directed Acyclic Graph (DAG)**

Let’s do DFS

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





start:9
finish:10

coreutils

start:7
finish:8

tar

multiarch -support

start:3
finish:4

dpkg

start:0
finish:11

libselinux1

libbz2

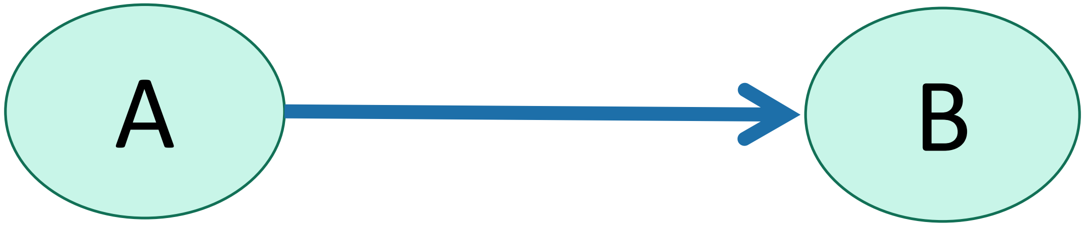
start:2
finish:5

start:1
finish:6

Suppose the underlying graph has no cycles

Finish times seem useful

Claim: In general, we'll always have:



finish: [larger]

finish: [smaller]

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

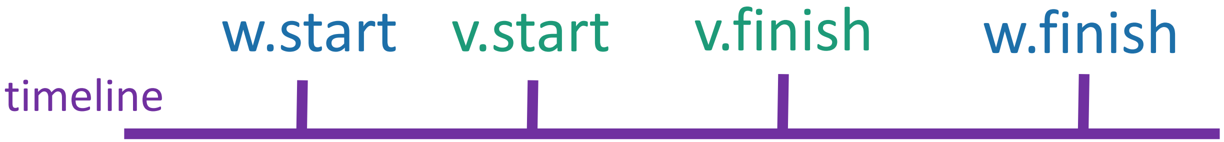
A more general statement

(this holds even if there are cycles)

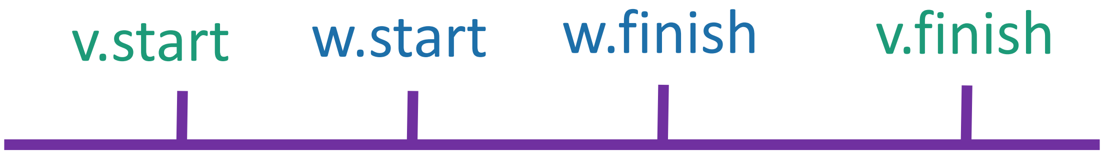
This is called the "parentheses theorem"

(check this statement carefully!)

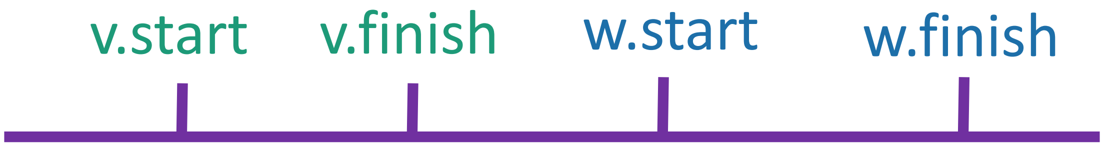
- If v is a descendant of w in this tree:



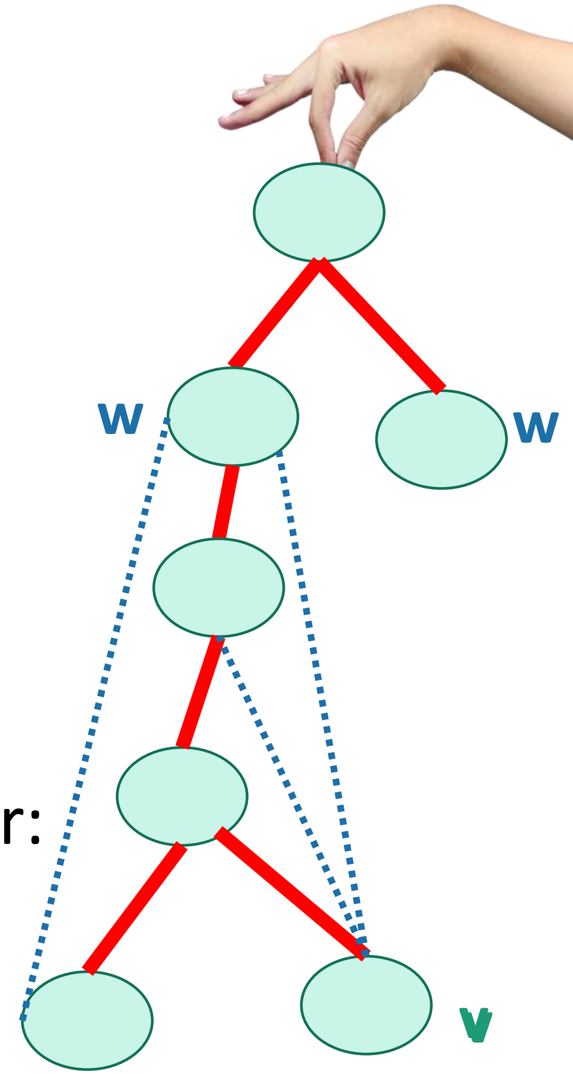
- If w is a descendant of v in this tree:



- If neither are descendants of each other:

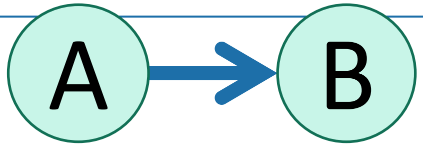


(or the other way around)



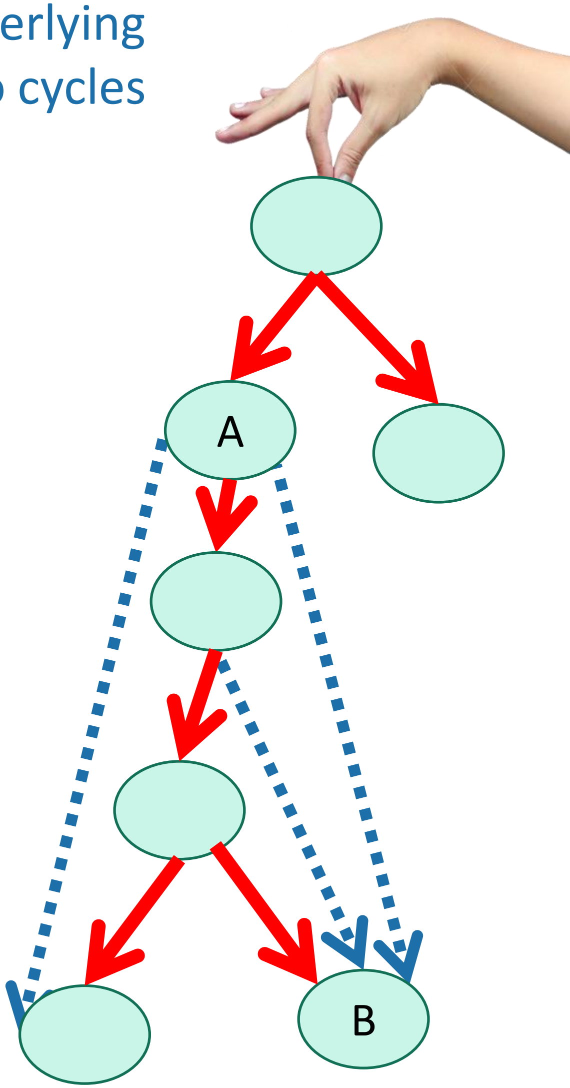
So to prove this →

If



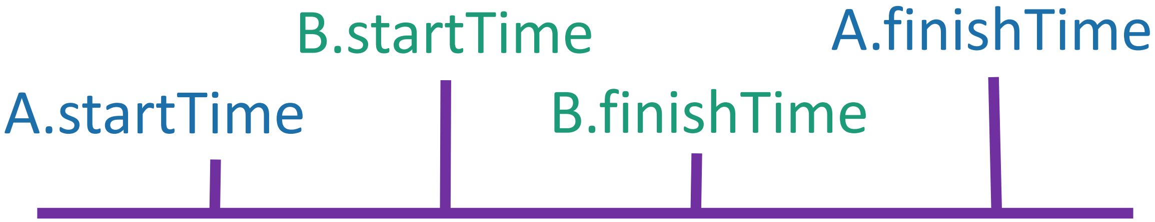
Then B.finishTime < A.finishTime

Suppose the underlying graph has no cycles



- **Case 1**: B is a descendant of A in the DFS tree.

- Then



A.startTime

B.startTime

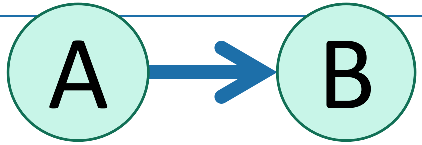
B.finishTime

A.finishTime

- aka, B.finishTime < A.finishTime.

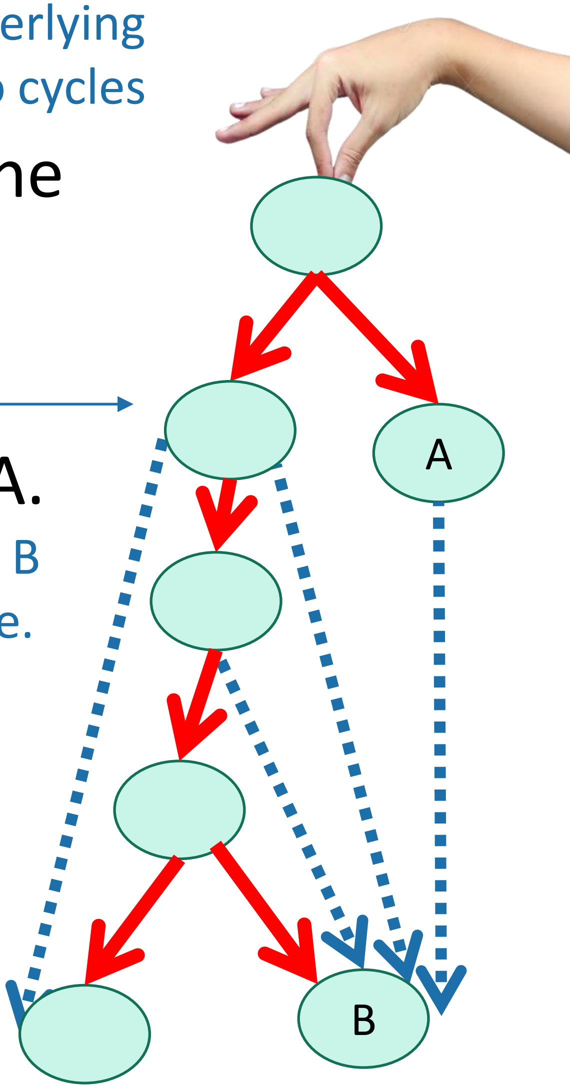
So to prove this →

If

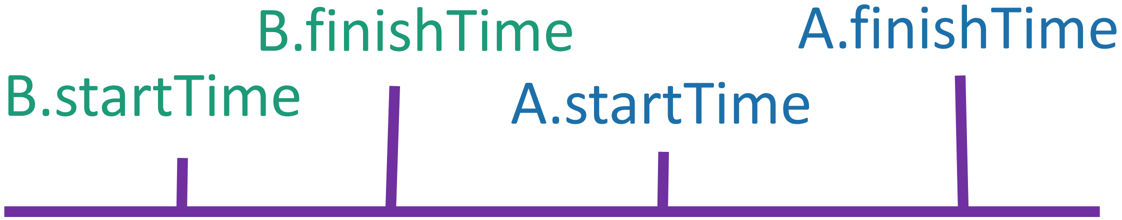


Then B.finishTime < A.finishTime

Suppose the underlying graph has no cycles



- **Case 2**: B is a NOT descendant of A in the DFS tree.
  - Notice that A can’t be a descendant of B or else there’d be a cycle; so it looks like this →
- Then we must have explored B before A.
  - Otherwise we would have gotten to B from A, and B would have been a descendant of A in the DFS tree.
- Then

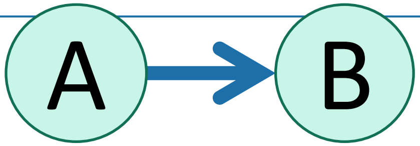


- aka, B.finishTime < A.finishTime.

Theorem

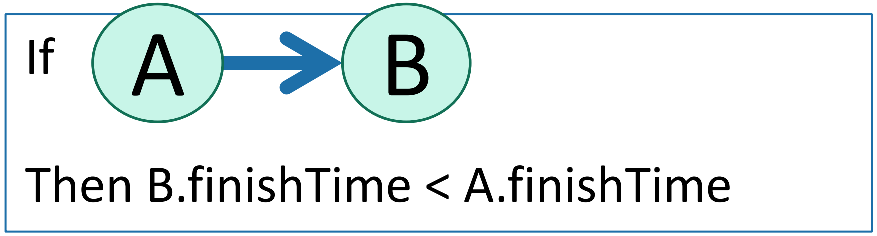
- If we run DFS on a directed acyclic graph,

If



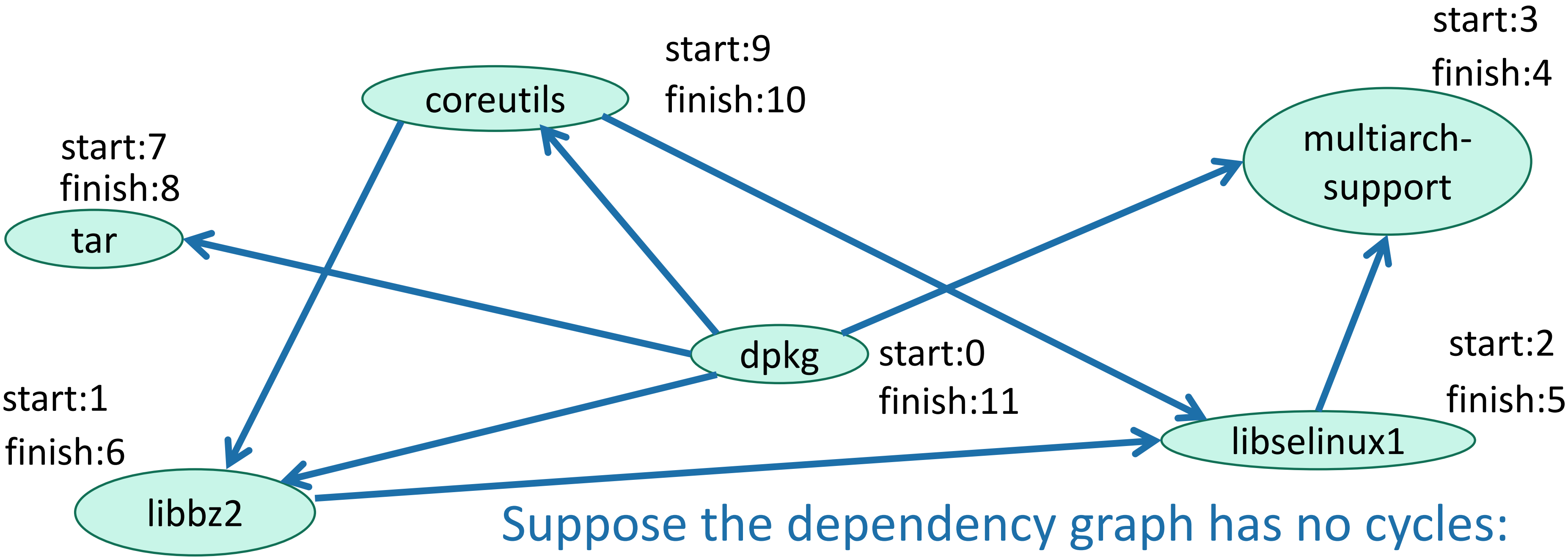
Then B.finishTime < A.finishTime

Back to topological sorting



Then B.finishTime < A.finishTime

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

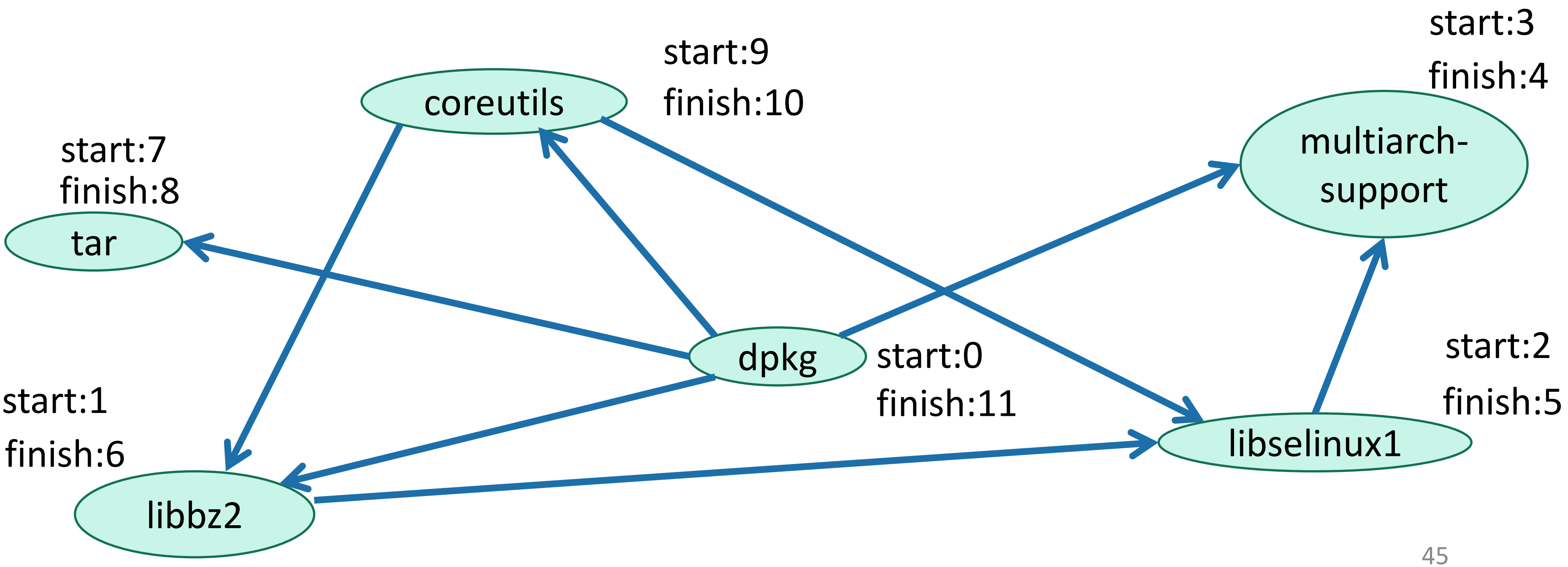


Suppose the dependency graph has no cycles: it is a **Directed Acyclic Graph (DAG)**

Topological Sorting (on a DAG)

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

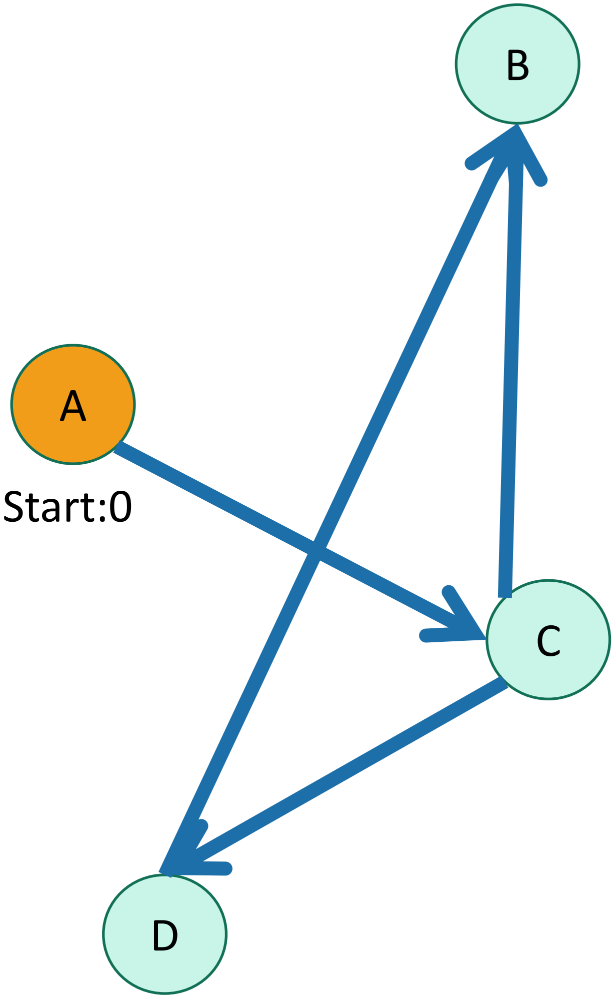
- dpkg
- coreutils
- tar
- libbz2
- libselinux1
- multiarch_support



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:

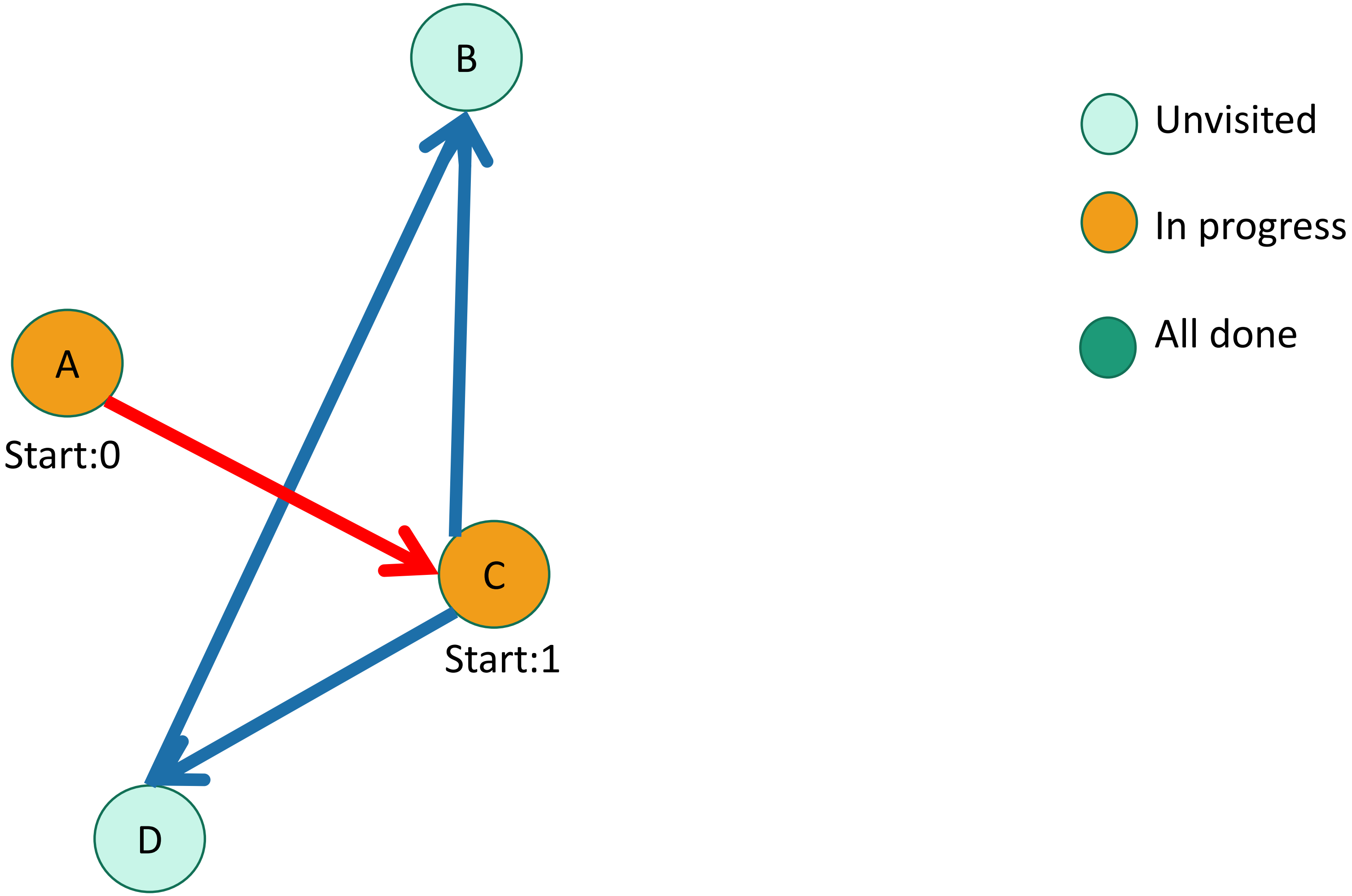


○ Unvisited

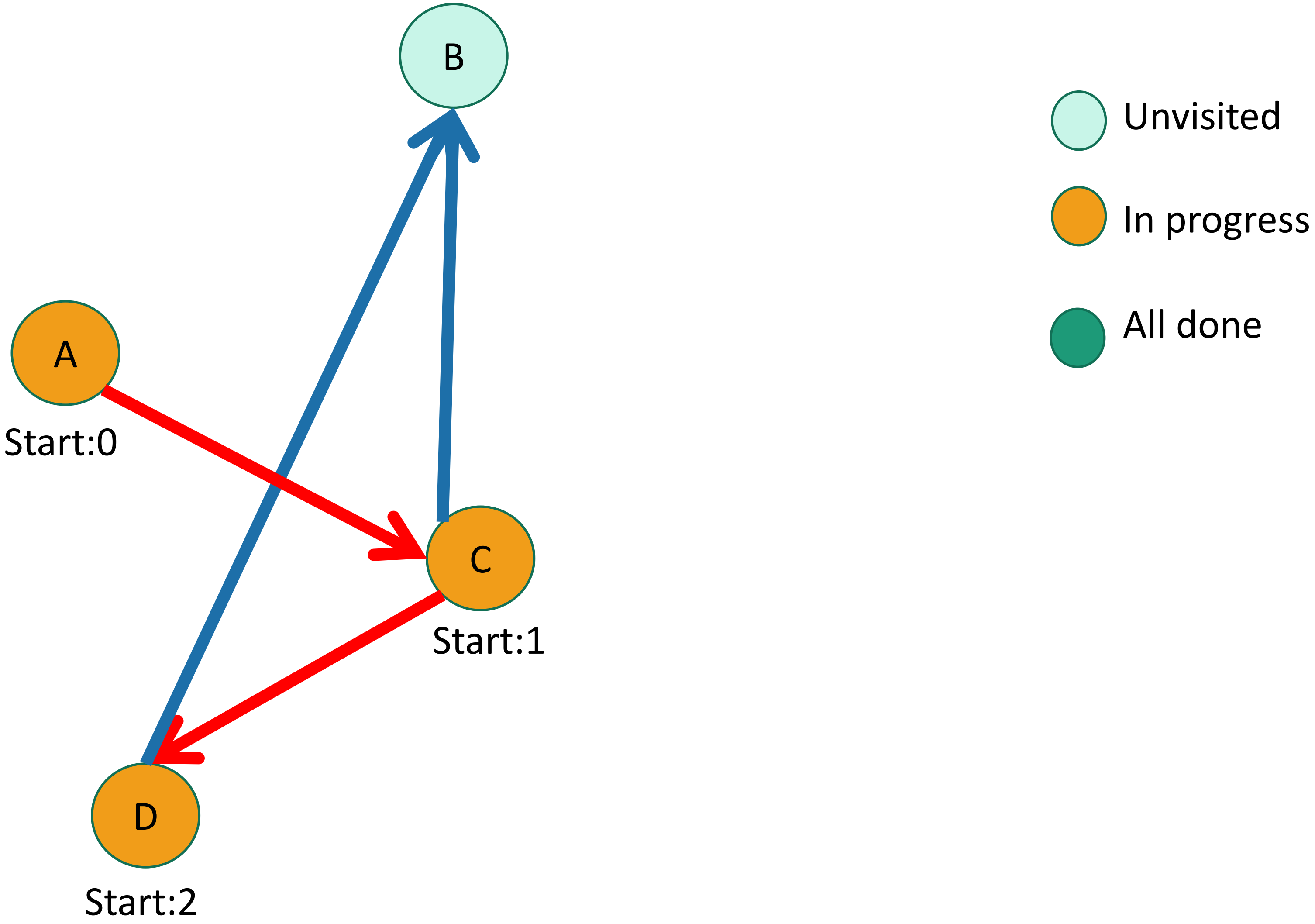
🟠 In progress

🟢 All done

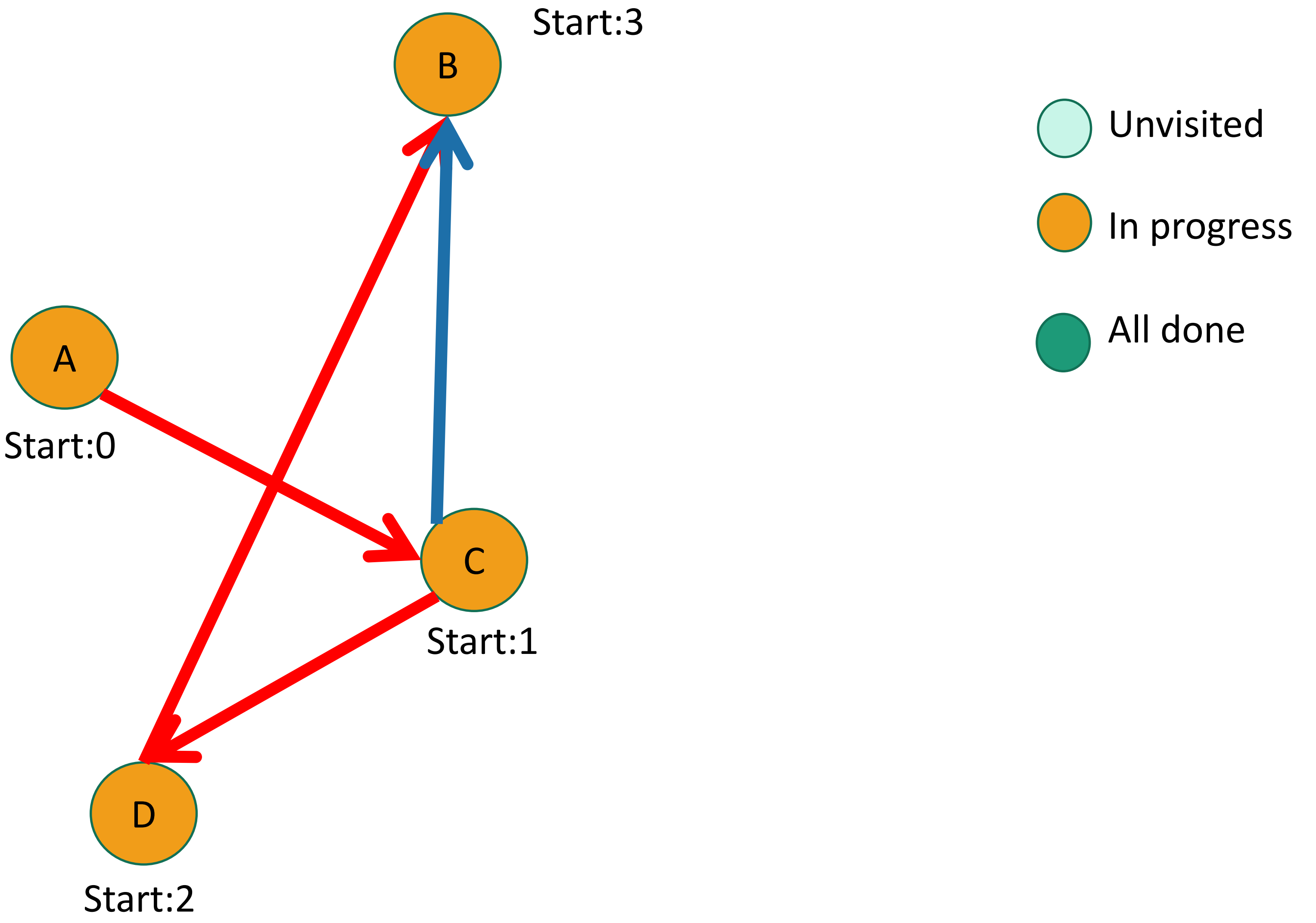
Example



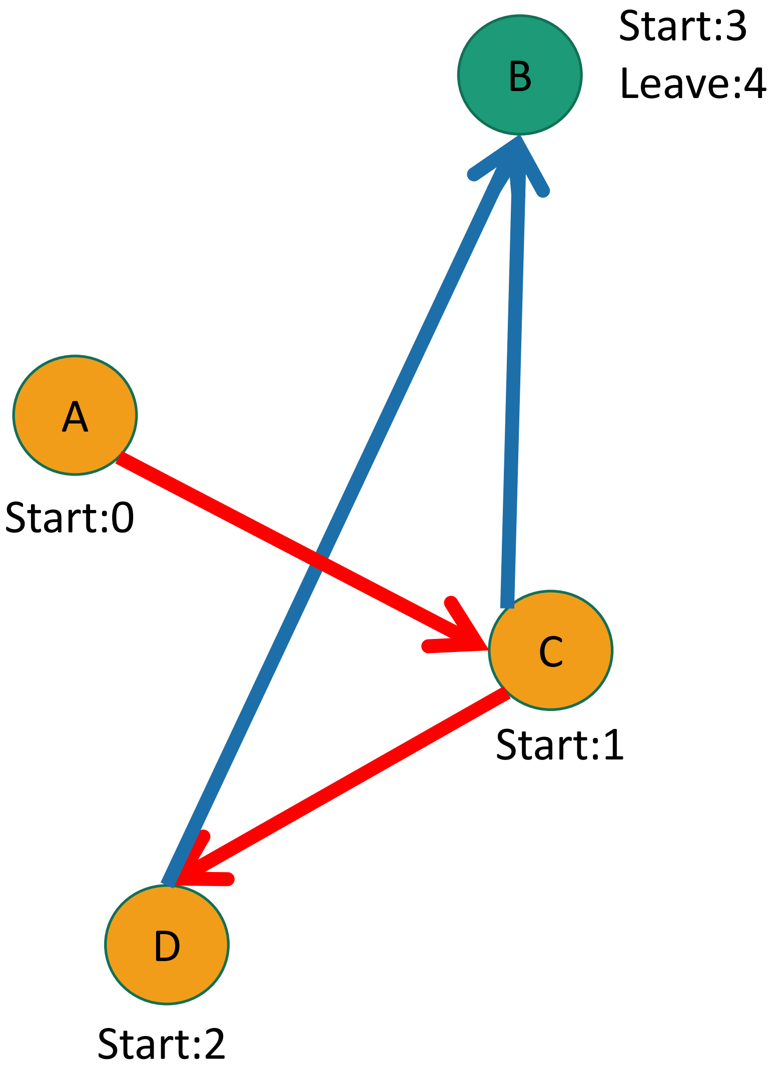
Example



Example


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

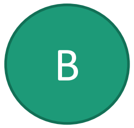
Example



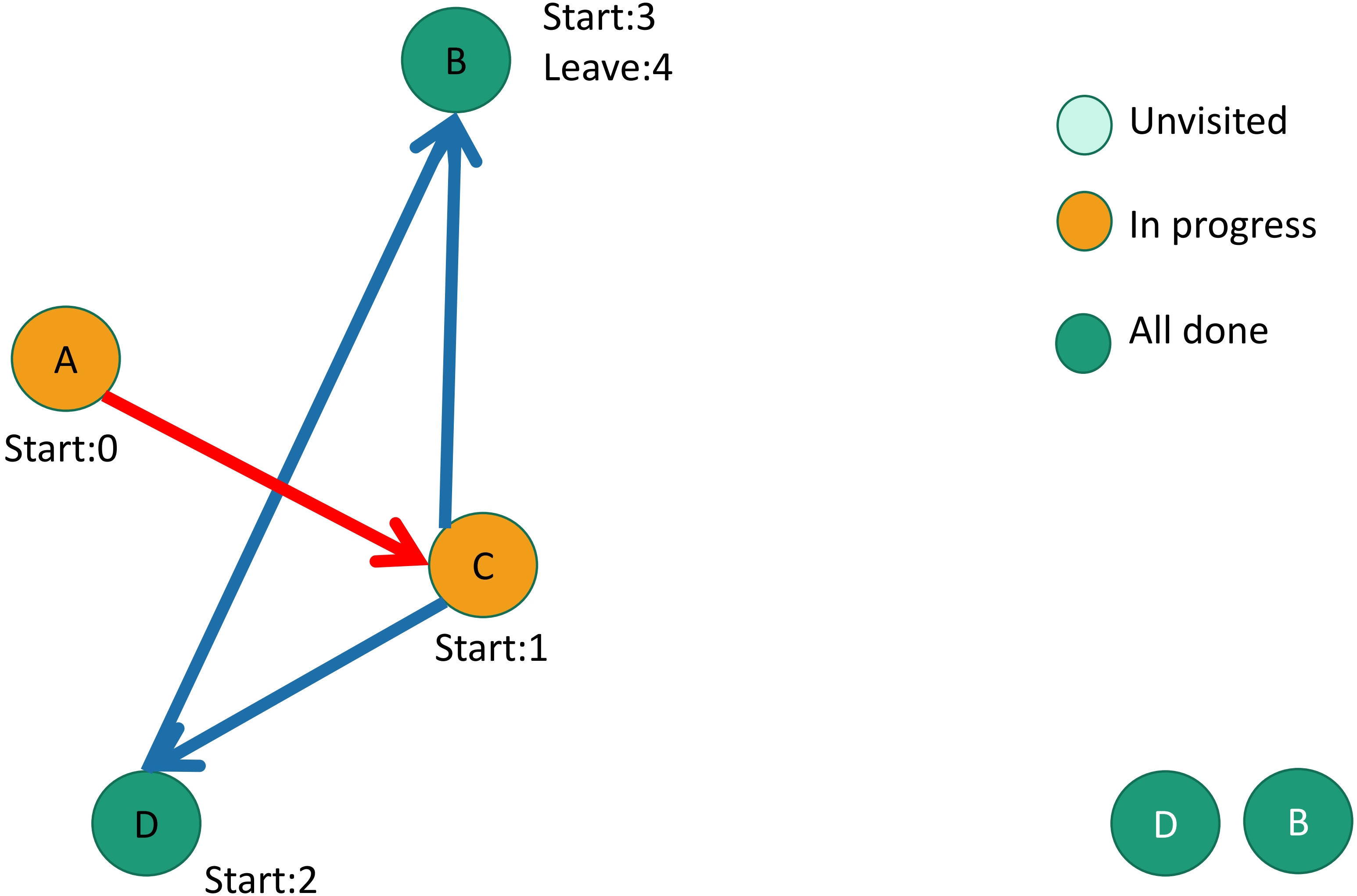
○ Unvisited

● In progress

● All done



Example



B

Start:3

Leave:4

A

Start:0

C

Start:1

D

Start:2

Leave:5

○ Unvisited

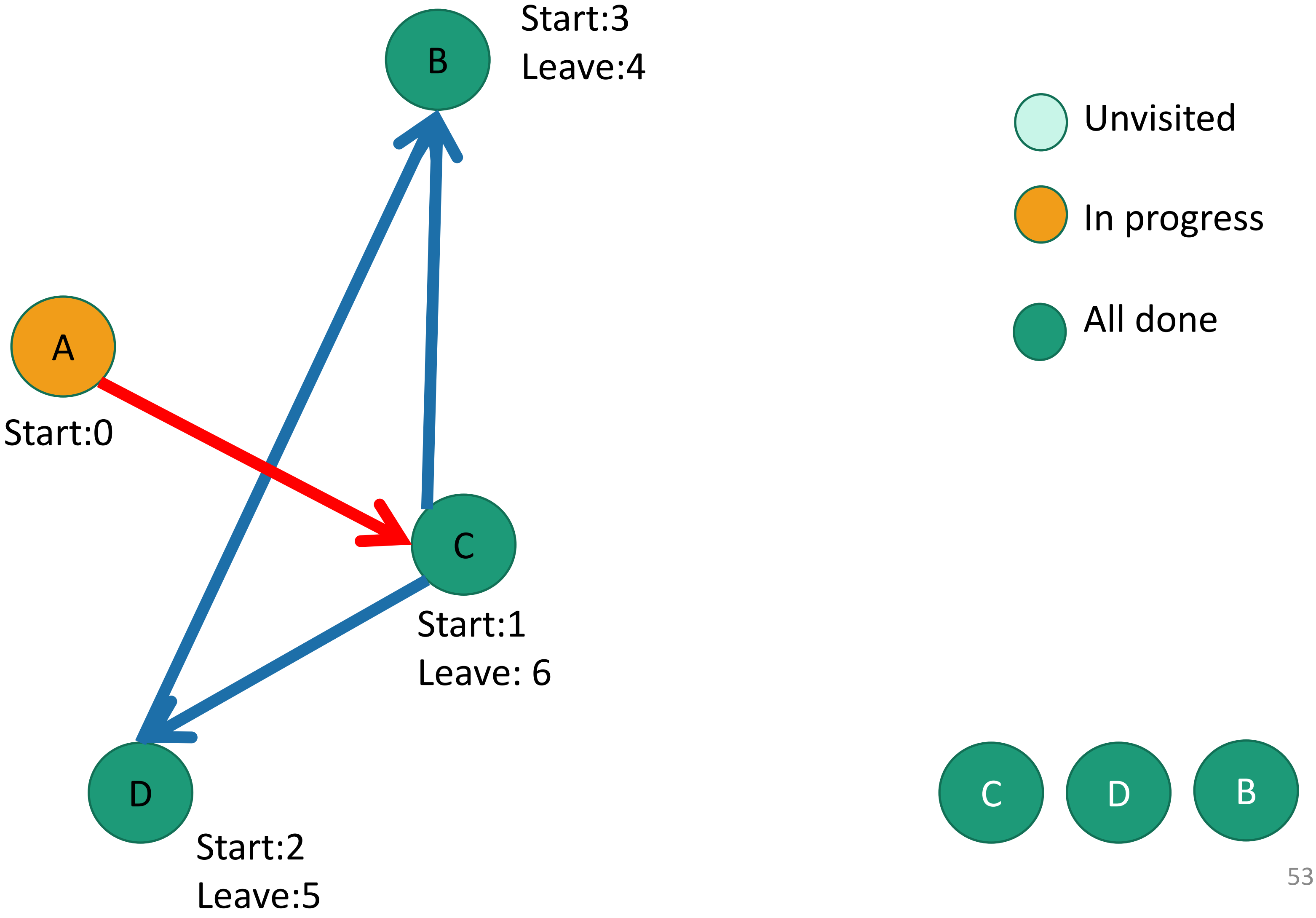
● In progress

● All done

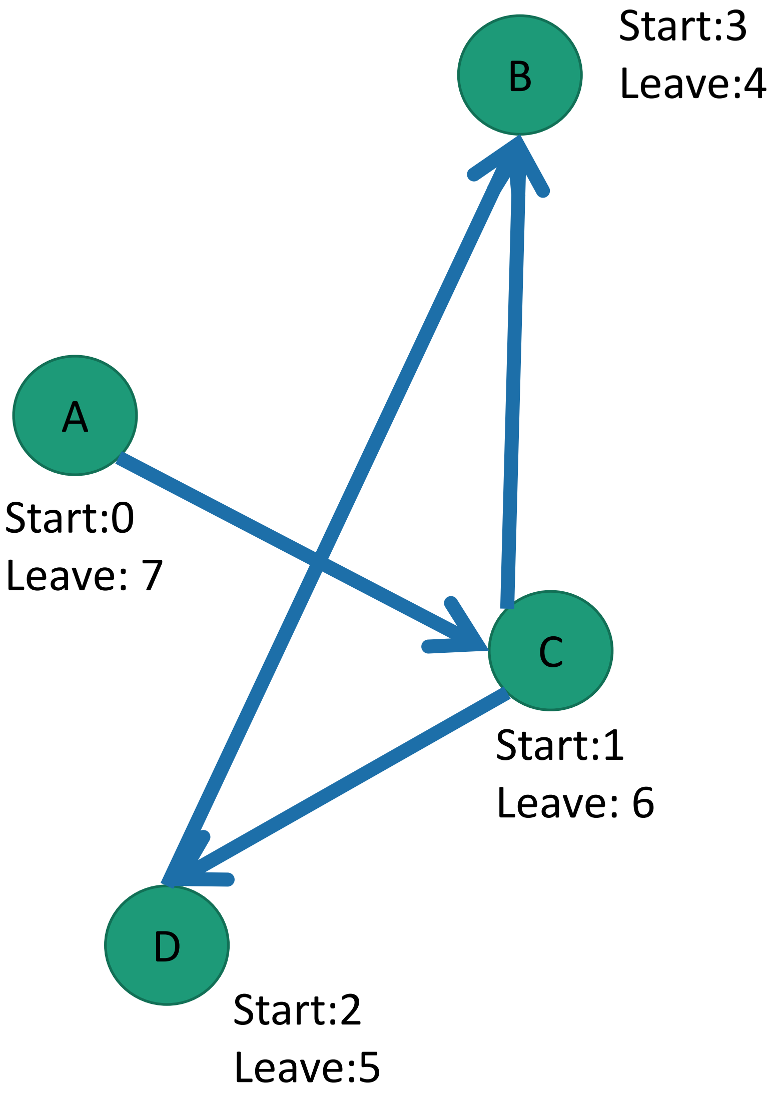
D

B

Example


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

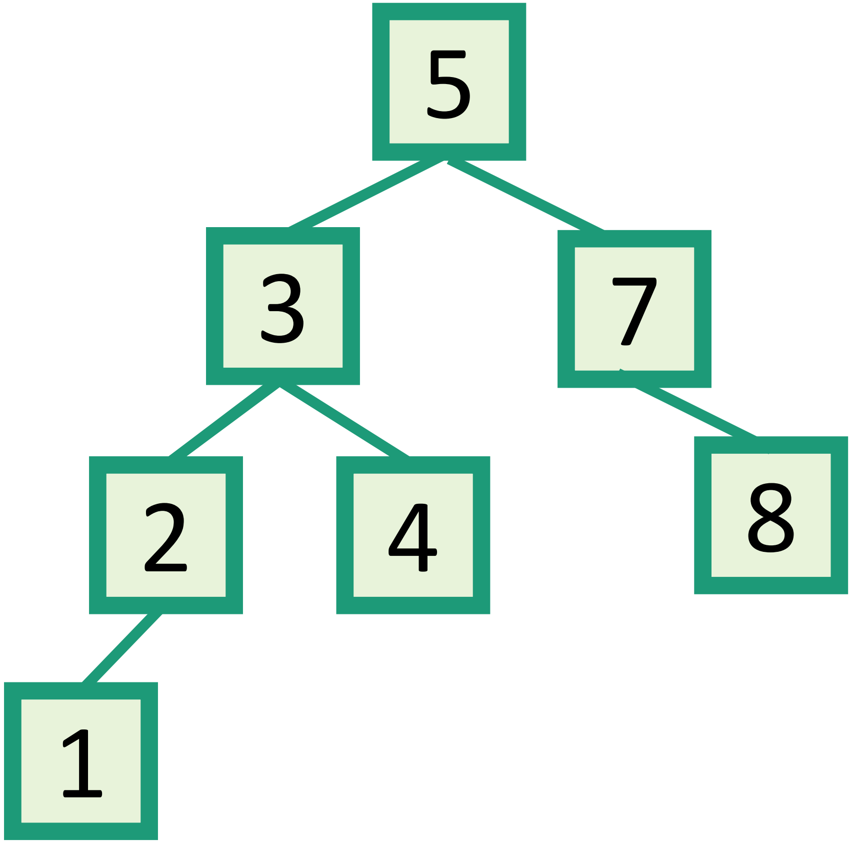
● All done

Do them in this order:

A C D B

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.