CS 170
Efficient Algorithms and Intractable Problems

Lecture 7:
Strongly Connected Components (aka DFS is awesome!)

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

We have released discussion solutions a bit earlier this week
→ You can refer to them to help with HW 3.

Please fill out HW party feedback form:
→ https://tinyurl.com/cs170-hwp
→ We'd like to know what's working and what needs to be improved.

Recap of last lecture

Exploring graphs

- $explore(u)$ visited exactly the set of edges reachable from u.
- DFS: repeatedly calling explore. Runs in time $O(n+m)$.

→ Found connected components of undirected graphs.

→ Edge types: tree edge, back edge, forward edge, cross edge (directed graphs only)

$explore(G, u)$

  $visited[u] = true$

  $pre[u] = clock; clock{+}{+}$

  For v such that $\{u, v\} \in E$ //alphabetic order

    If $visited[v] = false$ then $explore(G, v)$

  $post[u] = clock; clock{+}{+}$

dfs(G)

boolean array $visited(n)$

  // initialize to all false.

$clock = 1$

int array $pre(n), post(n)$

For $v \in V$

  If $visited[v] = false$ then $explore(G, v)$

DFS Tree/Forest for Directed Graphs

Edge (u, v):

- Tree edge

  Recursive explore calls

  $pre[u] < pre[v] < post[v] < post[u]$

- Forward edge:

  From ancestor to non-child descendent

  $pre[u] < pre[v] < post[v] < post[u]$

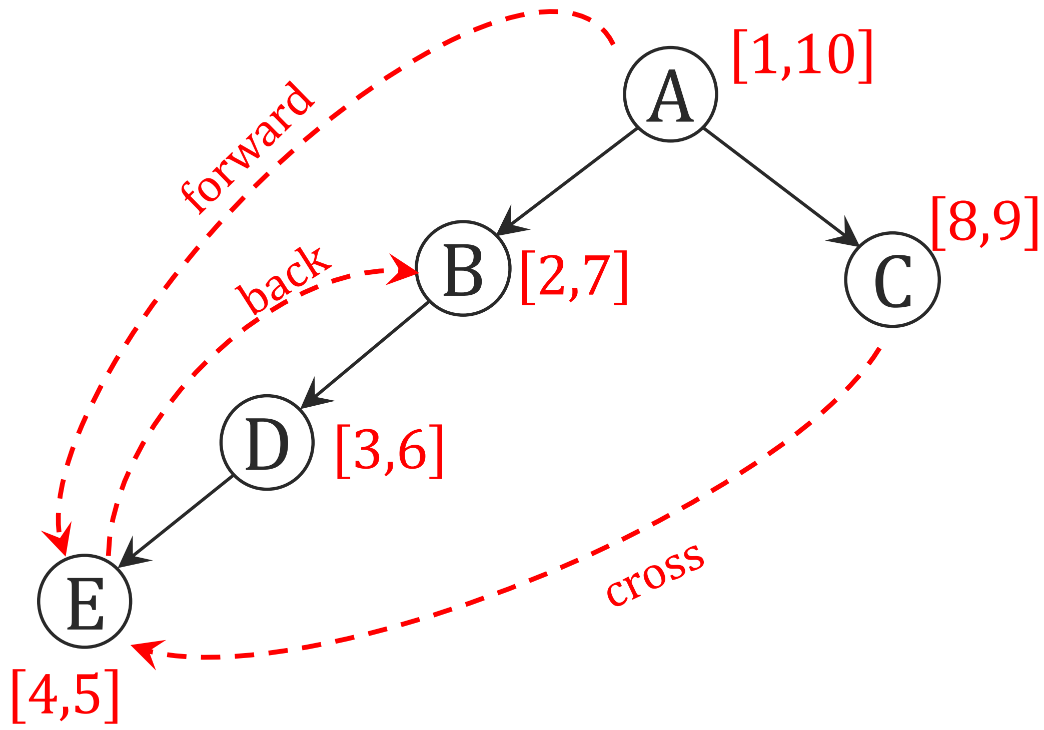
- Back edge:

  From descendent to ancestor

  $pre[v] < pre[u] < post[u] < post[v]$

- Cross edge:

  Between neither descendent or ancestor.



Cross Edge

Imagine $(u, v) \in E$ is a cross edge.

What is the relationship between $[pre[v], post[v]]$ and $[pre[u], post[u]]$?

$$pre[v] < post[v] < pre[u] < post[u]$$

Edges Types and Intervals Summary

Edge $(u, v) \in E$	
Tree / Forward edge	$pre[u] < pre[v] < post[v] < post[u]$
Back edge:	$pre[v] < pre[u] < post[u] < post[v]$
Cross edge:	$pre[v] < post[v] < pre[u] < post[u]$

All other relationships between intervals are impossible!

This lecture

Using DFS and pre/post times in other algorithm design problems.

→Topological sort

→Strongly Connected components

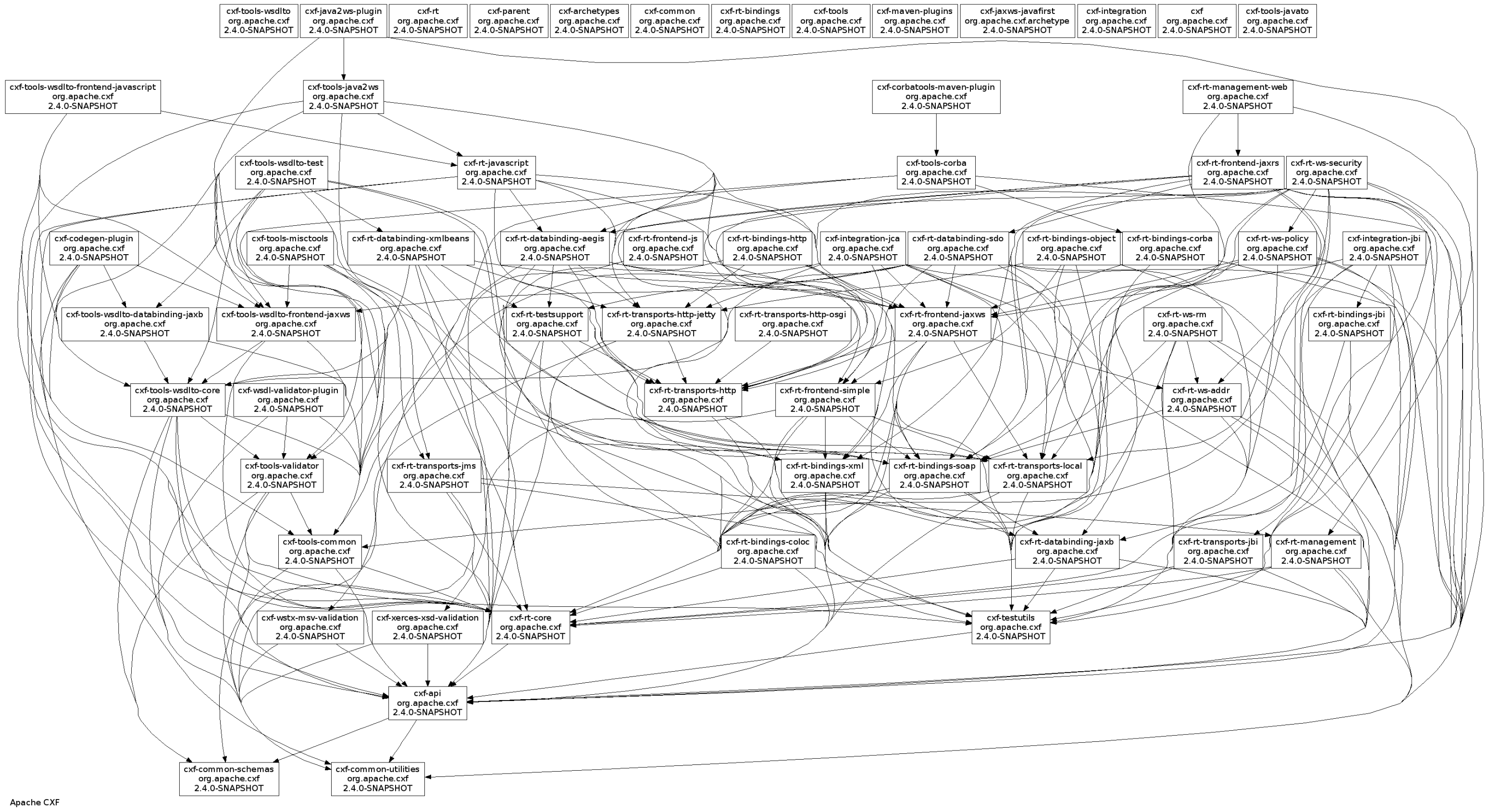
Topological Sort

Topological Sort

Find an ordering of vertices so that no edges go backward.

→ i.e., If u comes before v in the ordering, there is no edge (v, u).

E.g., software package dependency

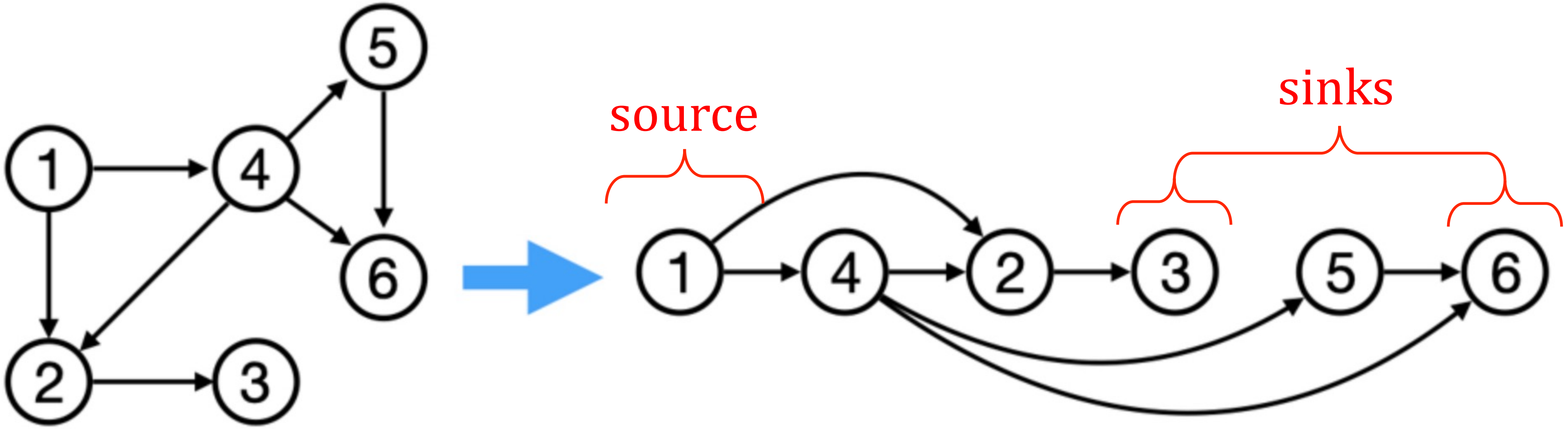


Topological Sort

Find an ordering of vertices so that no edges go backward.

→ i.e., If u comes before v in the ordering, there is no edge (v, u).

E.g., software package dependency



Source: Any node that has no incoming edges and has only outgoing edges.
Sink: Any node that has no outgoing coming edges and has only incoming edges.

Topological Sort, DAGS, and Back edges

Definition: a directed acyclic graph (DAG) is a graph with no directed cycles.

Claim: Suppose we run a DFS on on G. G is a DAG if and only if it has no back edges!

Back Edges and Post-times are special!

Edge $(u, v) \in E$	
Tree / Forward edge	$pre[u] < pre[v] < post[v] < post[u]$
Back edge:	$pre[v] < pre[u] < \boxed{post[u] < post[v]}$
Cross edge:	$pre[v] < post[v] < pre[u] < post[u]$

An edge $(u, v) \in$ E is a back edge if and only if $post[u] < post[v]$.

Corollary: In a DAG, every edge $(u, v) \in E$ has the property that $post[v] < post[u]$.

Topological Sort Algorithm

How should we use the DFS to find a topological sort for a DAG?

3 min break!
Please close the doors to the auditorium!

Strongly Connected Components

Connected Components in directed Graphs

In undirected graphs, connected components can be found via DFS

Questions for directed graphs:

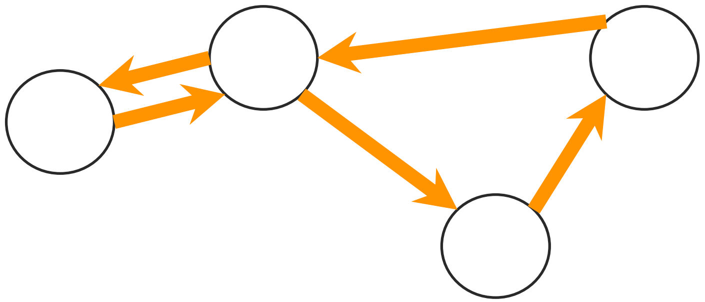
- How should we define connected components in a directed graph?
- How do we compute them, fast?

Strongly Connected Graphs

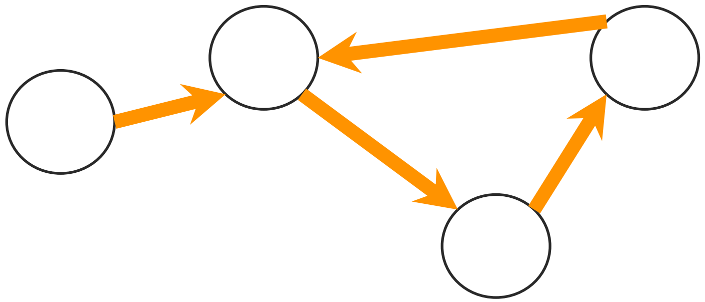
Vertices u, v are **strongly connected** if

- there is a path **from u to v**, and
- there is a path **from v to u.**

A directed graph G = (V,E) is **strongly connected** if all of its vertices are strongly connected.



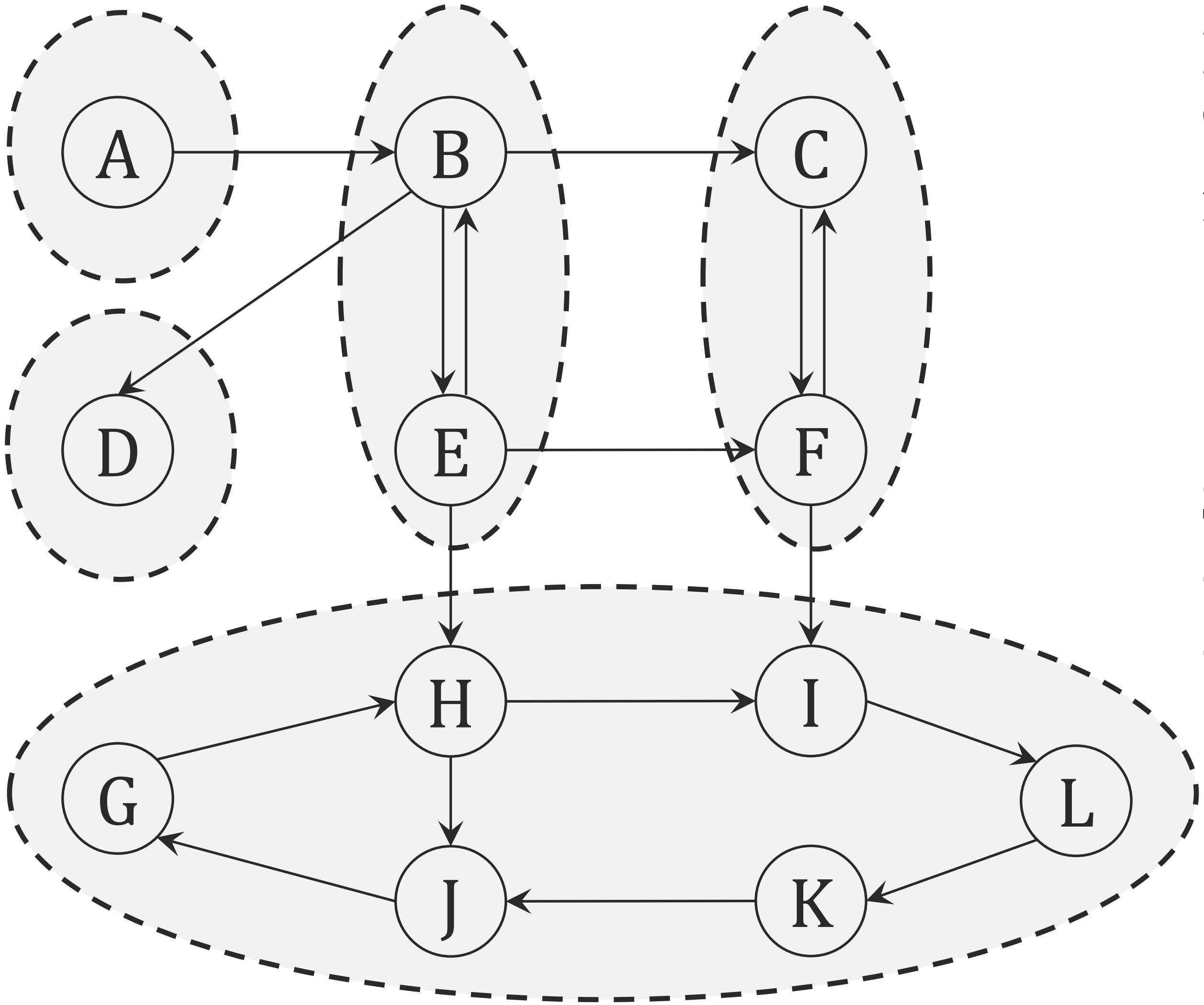
strongly connected



not strongly connected

Strongly Connected Components

We can partition a graph into **strongly connected components** (SCCs).



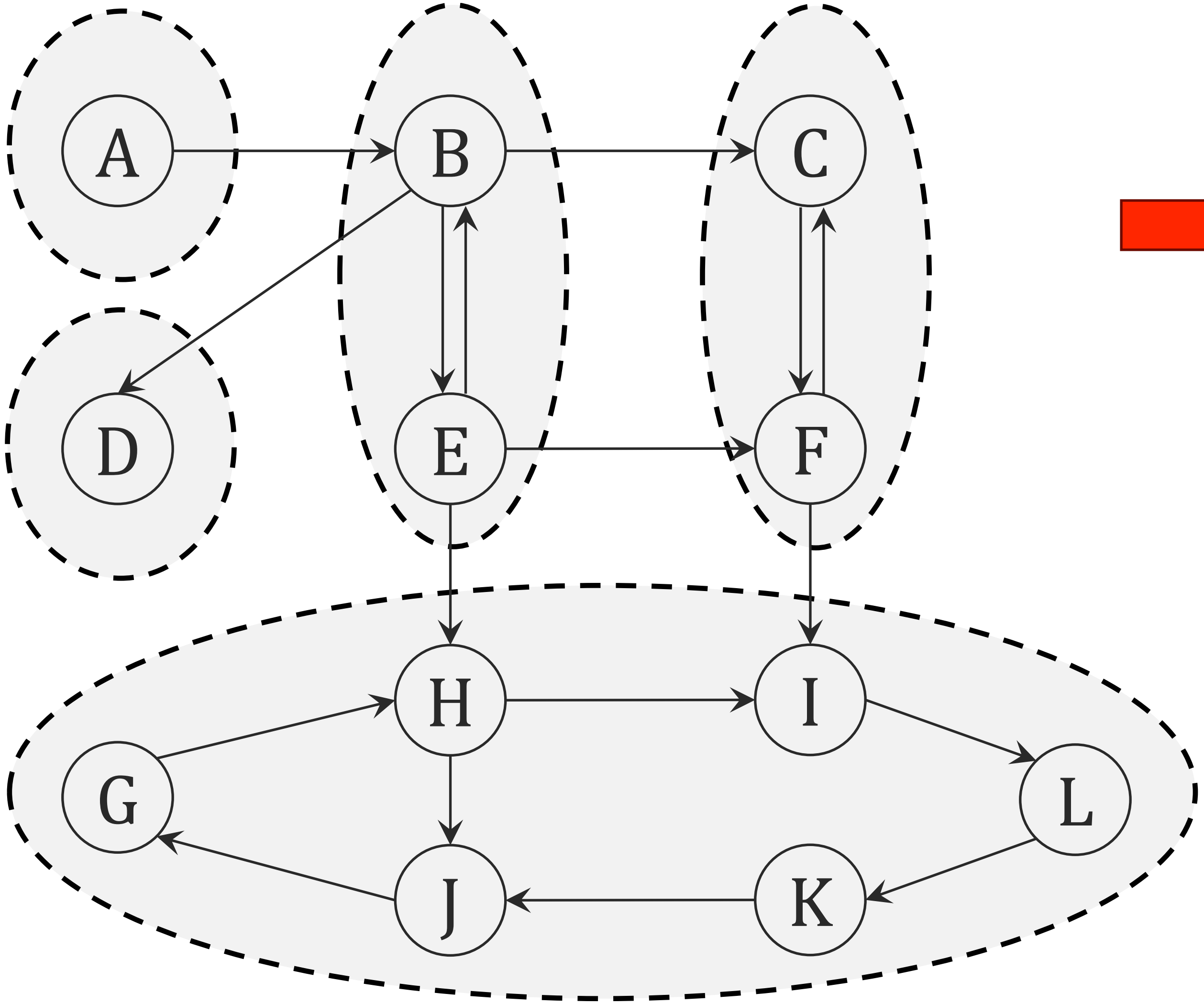
Formal definition:
"strong connectivity" is an equivalence relationship between vertices.
→ Reflexive, symmetric, and transitive!

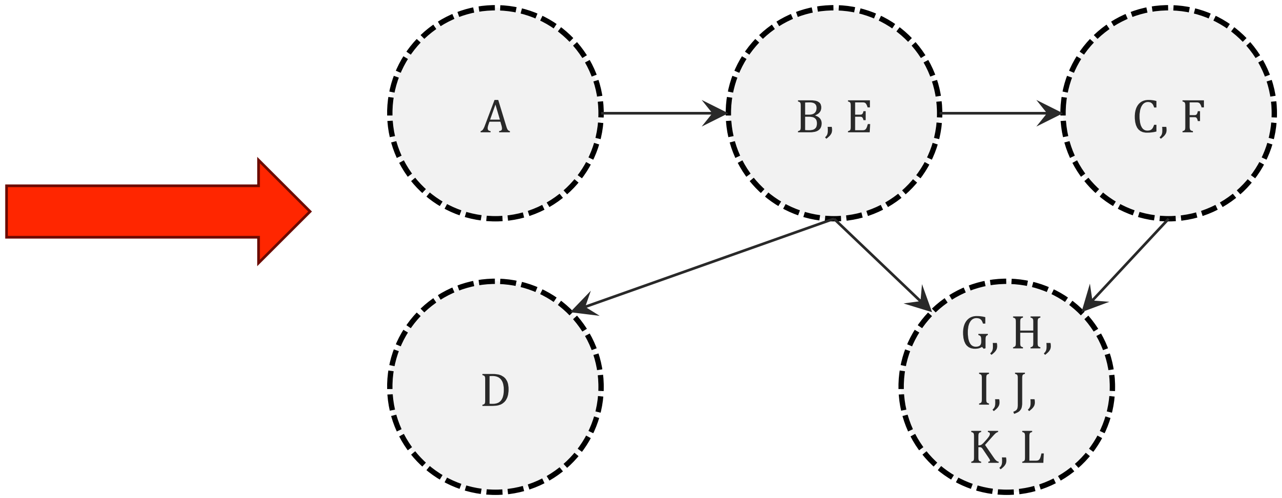
Strongly connected components are the equivalence classes under "strong connectivity" relationship.

Observation: If we reverse all edges of G, the SCCs don't change!

Meta Graph for SCC

We can partition a graph into **strongly connected components** (SCCs).





Discuss

Which one is true?

1. The meta graph can have a directed cycle
2. The meta graph is always a DAG.

Why would we care about SCC?

Useful broadly in science and engineering.

- Structure of graphs more generally: model checking, advertising, etc.
- SCCs tell you about communities of people, objects, similarities.
- Many graph algorithms only make sense on a SCC

→ Used a preprocessing step a lot.

How to find SCCs?

We won't go into the details here, these are bad algorithms anyway!

Slow attempt 1:

- Consider all possible decompositions and check.

Slow attempt 2: something like

- Run an *explore* for every node to get the set of reachable nodes.
- For every node u, and any $v \in explore(u)$, run $explore(v)$ and add v to the same component as u if u is visited in explored list of v ....

Runs in time $\Omega(n^2)$ at the very least.

How to find SCCs?

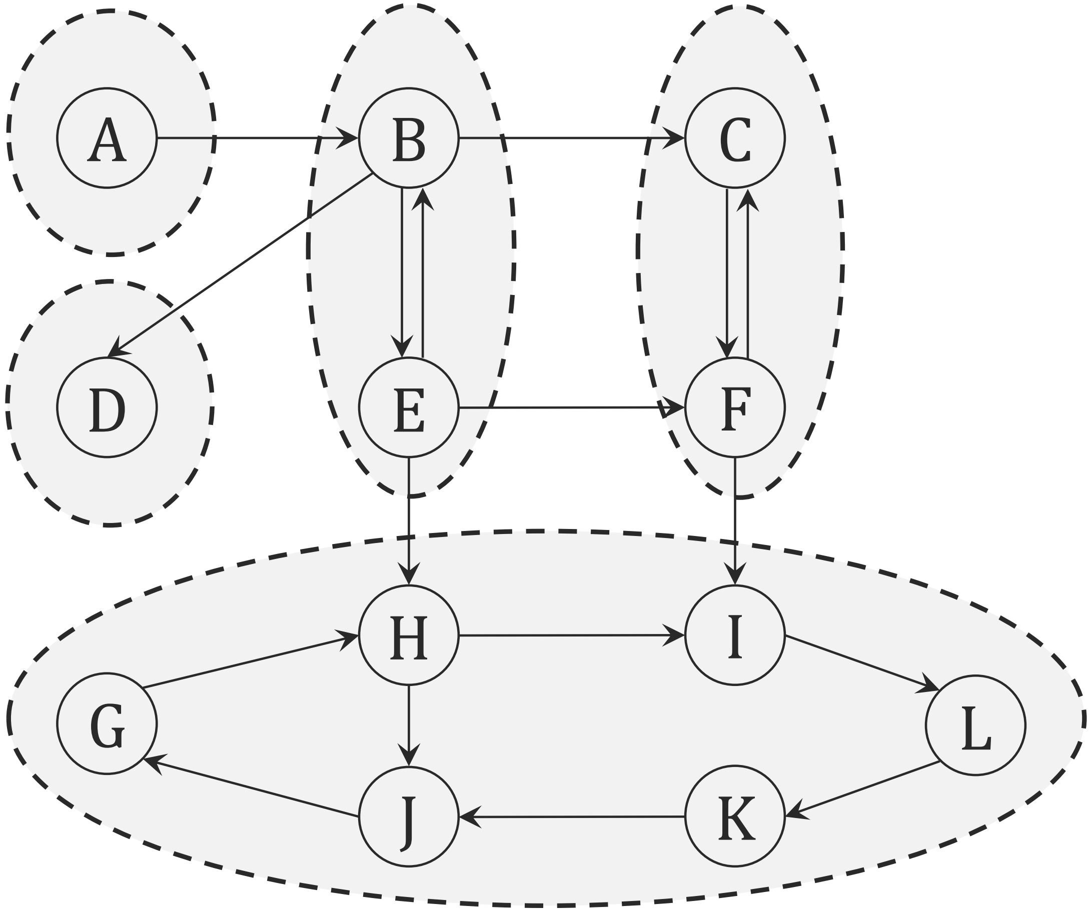
There is an algorithm for finding all SCCs that runs in time $O(n + m)$!

→ Just as efficient as one (or $O(1)$) rounds of DFS!

We will use just two calls to DFS!

How to find SCCs?

- What do you get when you run DFS from A?
- What about from G?
- What about D?



- Suggested algorithm: run DFS from the "right" place to identify SCCs.

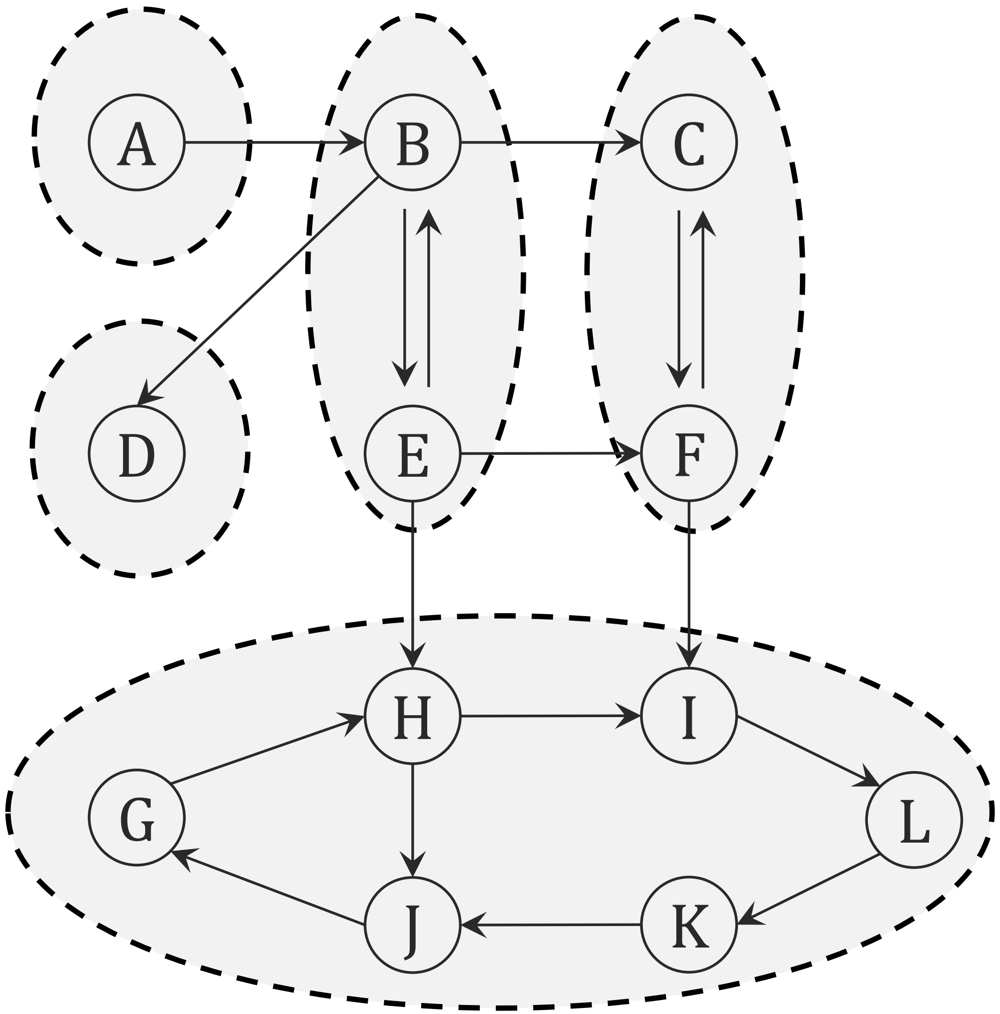
SCCs and *sinks* of the meta graph

Where is the “right” place to start DFS?

Start from any node in a sink of the meta graph!

Say, start from node G:

- *explore*(node G) visits the SCC of the sink.



SCCs and *sinks* of the meta graph

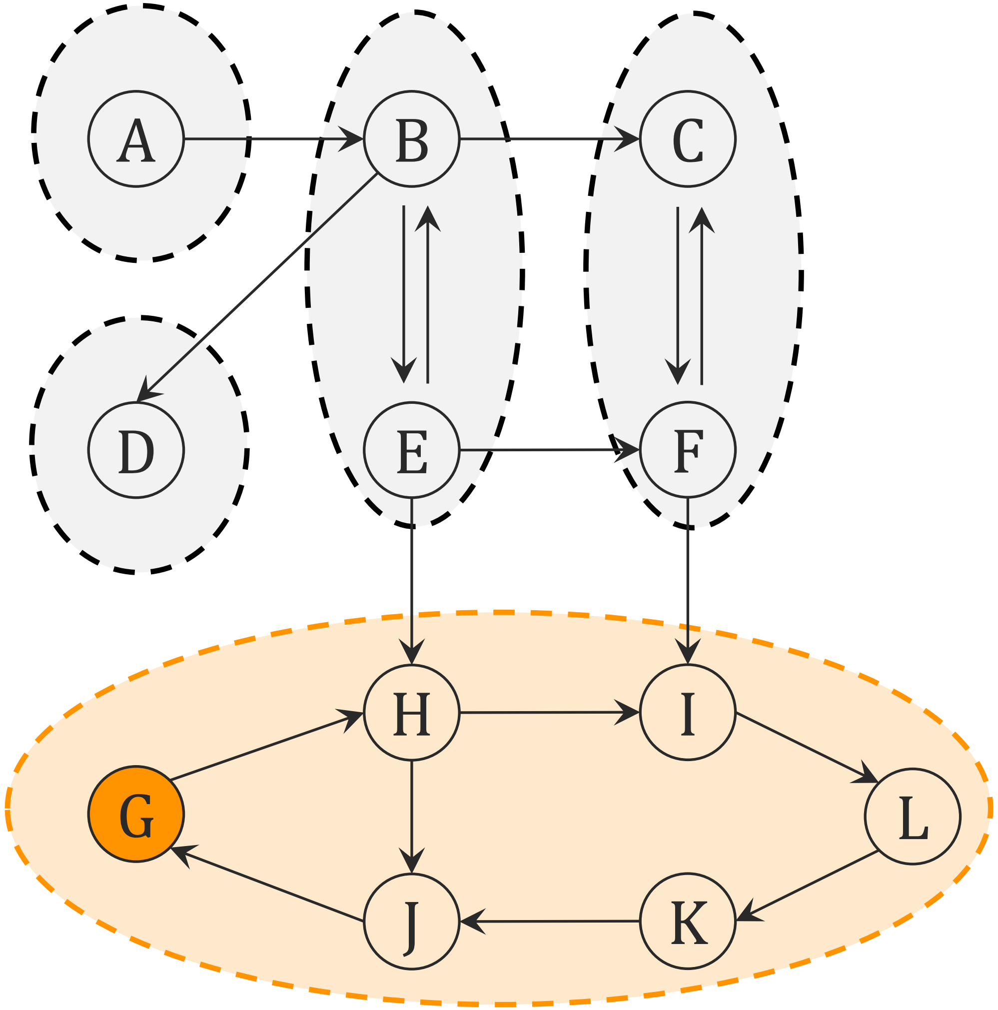
Where is the “right” place to start DFS?

Start from any node in a sink of the meta graph!

Say, start from node G:

- *explore*(node G) visits the SCC of the sink.
- No edge coming out of the sink component

→ It does not explore any node outside of SCC!



SCCs and *sinks* of the meta graph

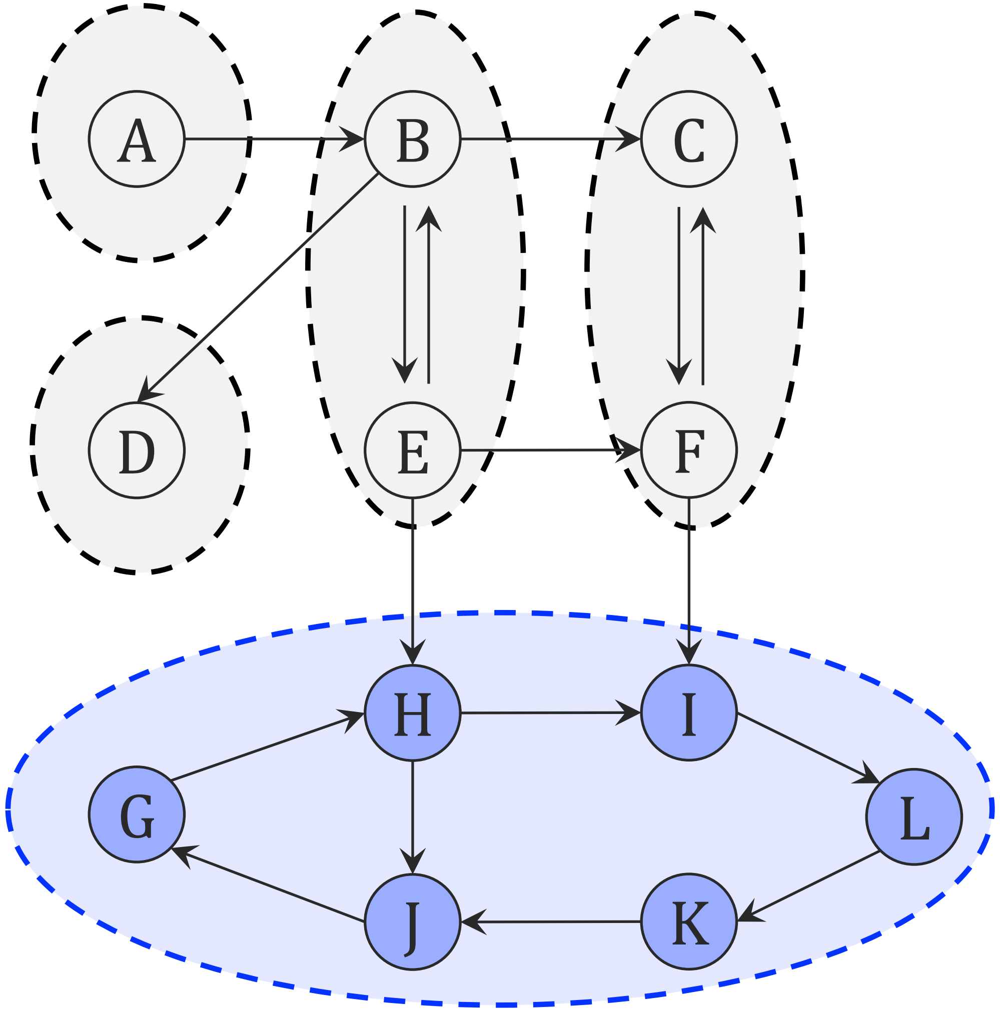
Where is the "right" place to start DFS?

Start from any node in a sink of the meta graph!

Say, start from node G:

- *explore*(node G) visits the SCC of the sink.
- No edge coming out of the sink component

→ It does not explore any node outside of SCC!



SCCs and *sinks* of the meta graph

Where is the "right" place to start DFS?

Start from any node in a sink of the meta graph!

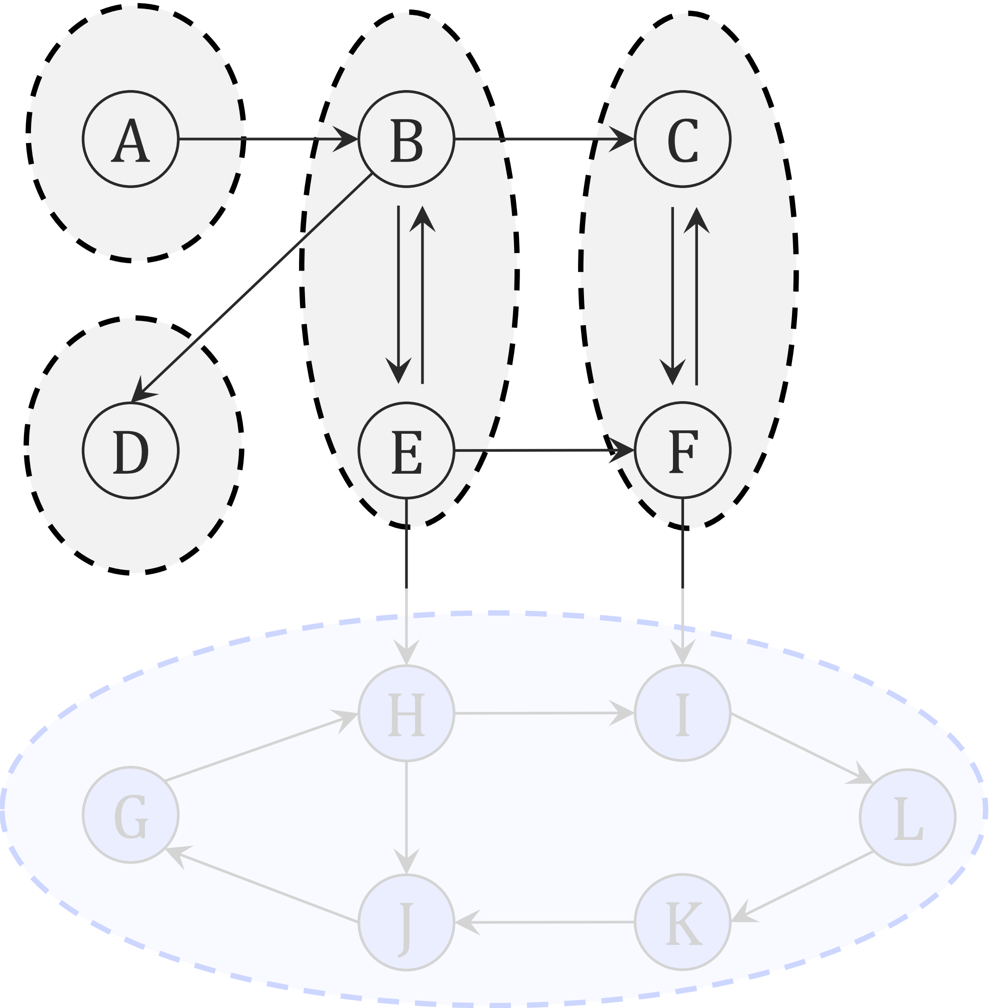
Say, start from node G:

- *explore*(node G) visits the SCC of the sink.
- No edge coming out of the sink component

→ It does not explore any node outside of SCC!

Remove this component, and continue

- *explore* any node in the sink of the remaining meta graph.



Sinks, Sources, and DFS post numbers

Claim: Suppose we run DFS on graph G. Let C and C’ be two SCCs such that $C \rightarrow C'$ in the meta graph. Then

$$\text{highest } post[v] \text{ for } v \in C > \text{highest } post[u] \text{ for } u \in C'$$

Sinks, Sources, and DFS post numbers

Corollary: Suppose we run DFS on graph G. The highest $post[v]$ belongs to a node v that is in the source SCC of the meta graph!

Reverse all the edges of G, first. Then, run the DFS!

The Reverse Graph Properties

Claim [we saw at the beginning of class]: G and G^R (reverse of G) have the same connected components! Also, edges in the meta graph of G^R are the reverse of edges in the meta graph of G.

Corollary: Suppose we run DFS on graph G^R. The highest $post[v]$ belongs to a node v that is in the sink SCC of the meta graph!

Algorithm for finding the SCCs

Compute the reverse of G, call it G^R.
Run DFS (using alphabetic order or any arbitrary order) on G^R.
→Store the post numbers of this DFS, say in an array called post-r.
Run DFS on G. This time, explore any unvisited node in the decreasing order post-r.
→Each new call from DFS to explore, finds a new SCC.

$explore(G, u)$

  $visited[u] = true$

  $sccnum[u] = count$

  For v such that $(u, v) \in E$

    If $visited[v] = false$ then $explore(G, v)$

Find SCCs(G)

boolean array $visited(n)$ // init all false.

[pre-r, post-r] ← dfs(G^R)

count = 1

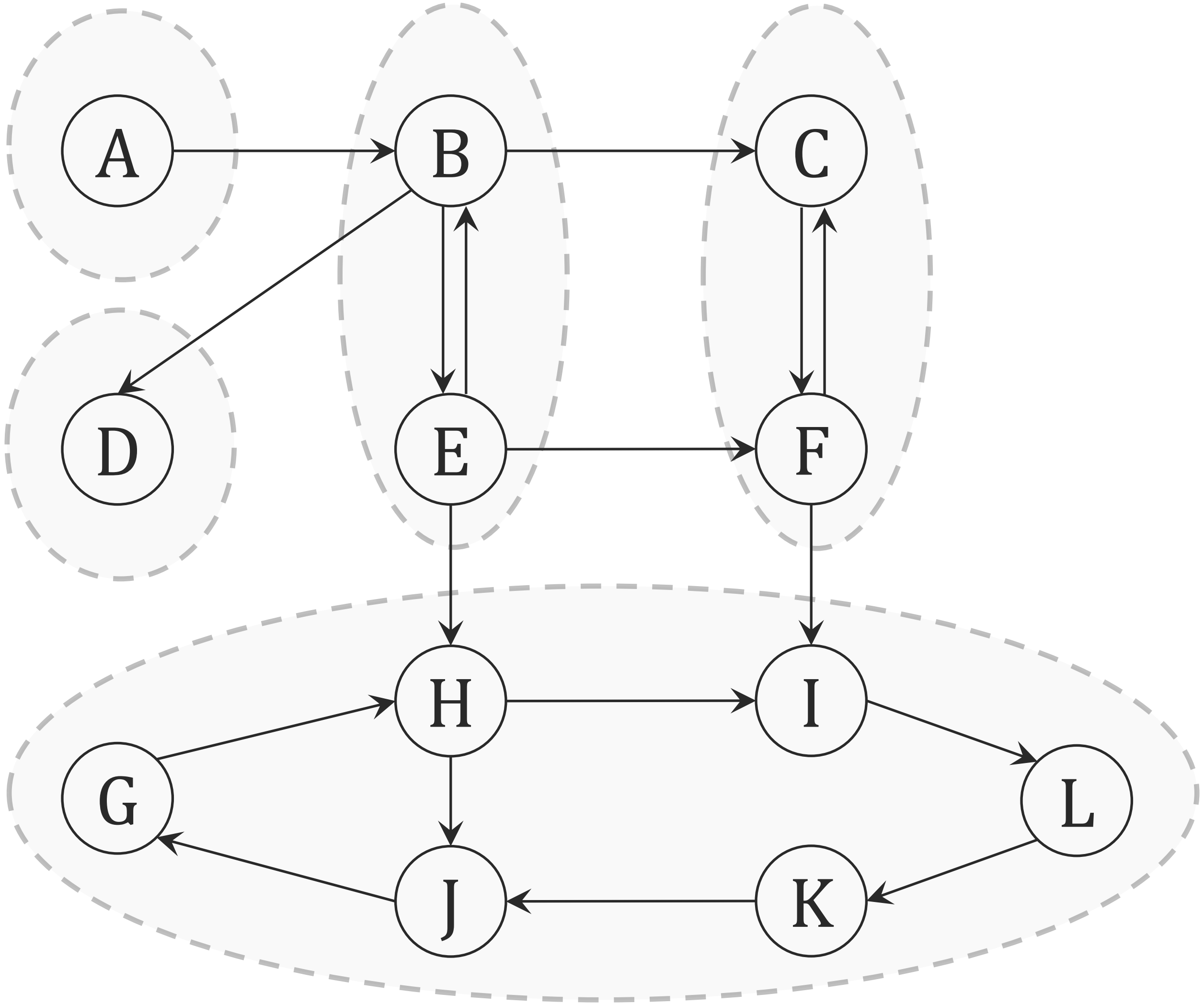
For $v \in V$ from highest to lowest post-r[v]

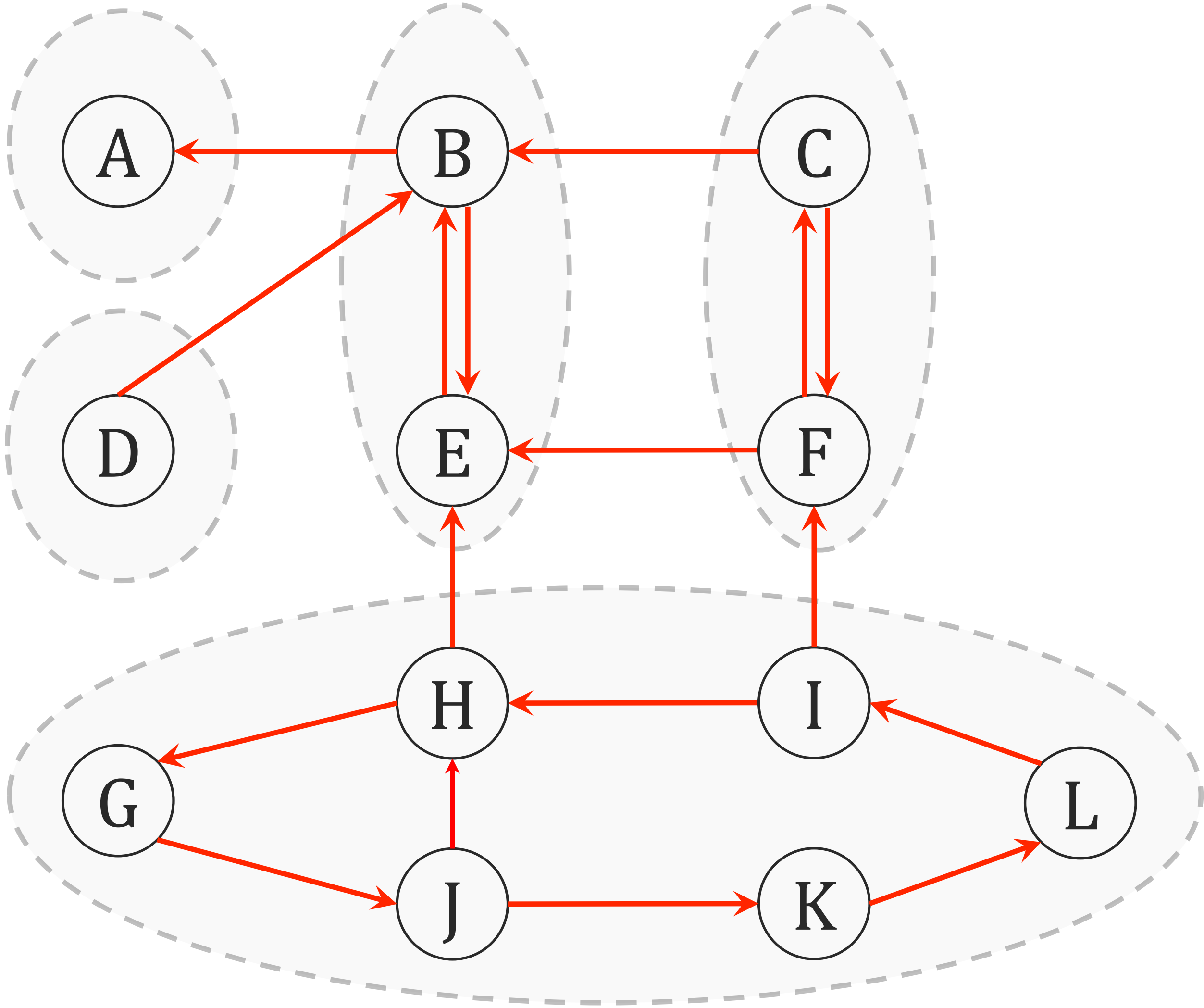
  If $visited[v] = false$ then

    $explore(G, v)$

    count ++

Algorithm step 1: Reverse G



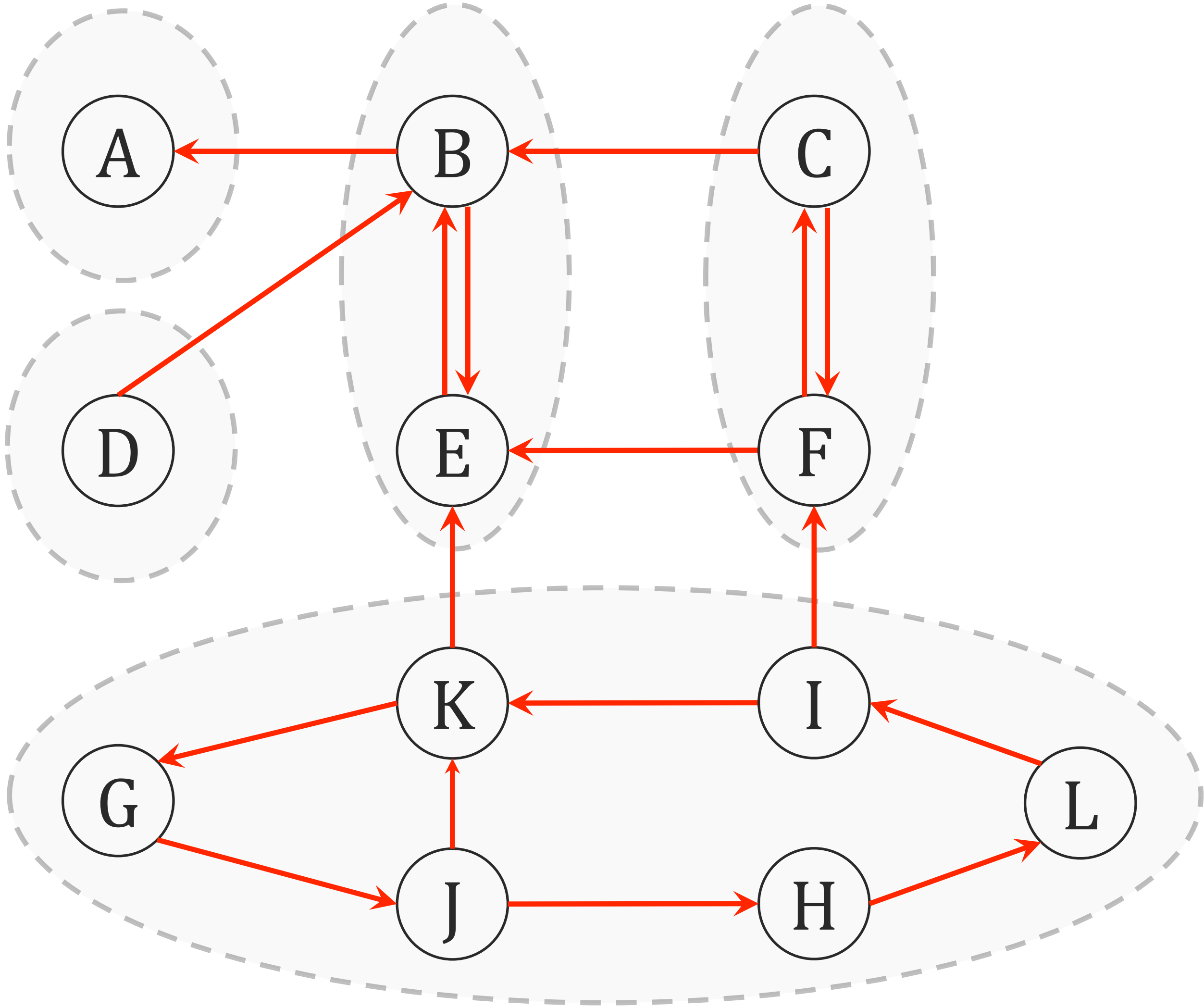


Alg step 2: Run DFS on the reverse of G

Not been there yet

Been there, still exploring.

Finished exploring



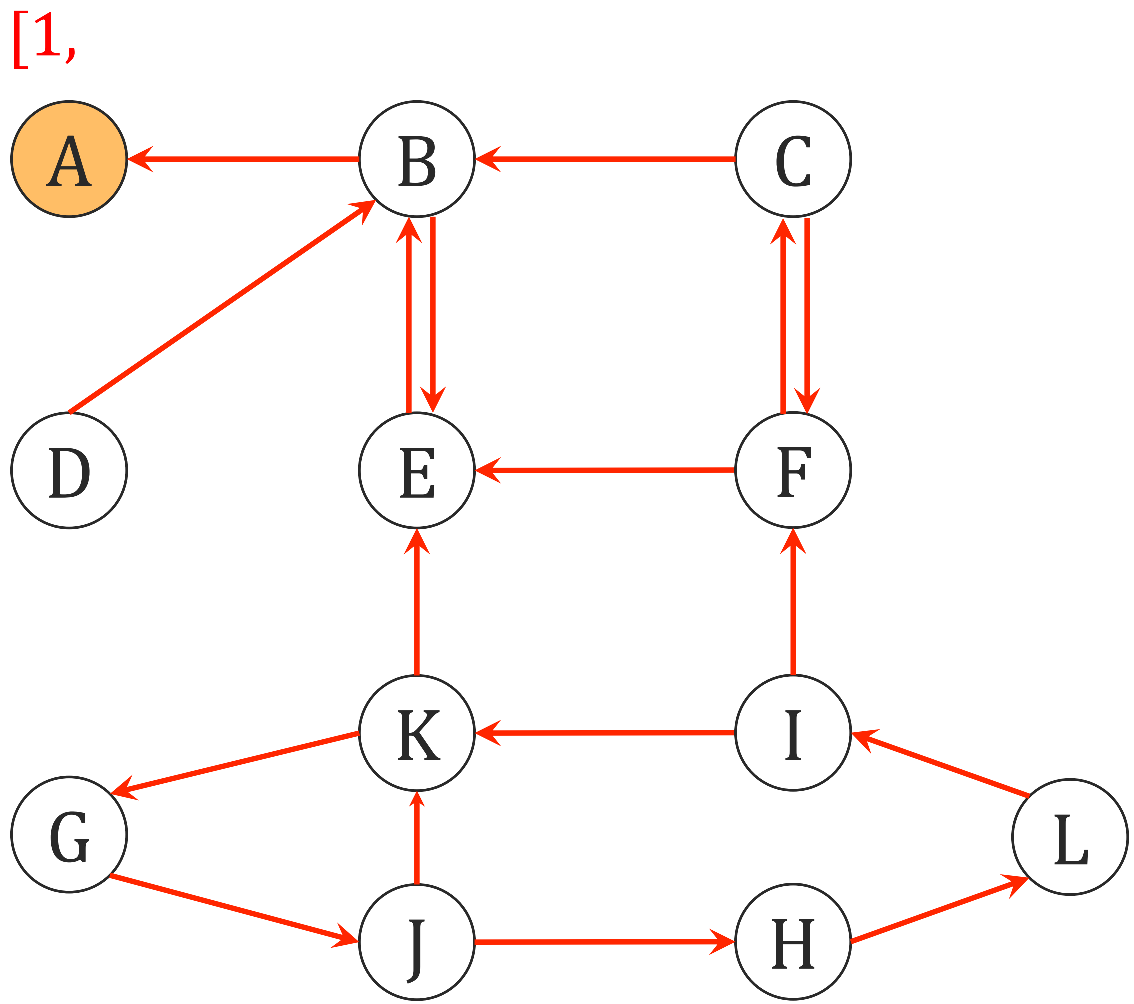
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

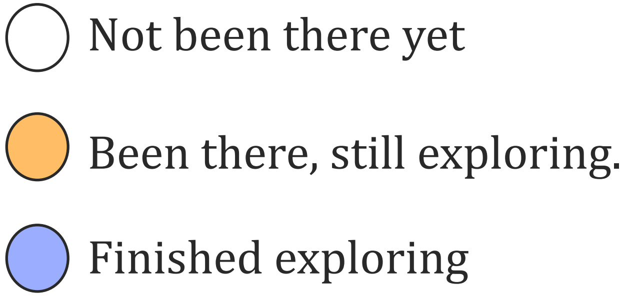
Been there, still exploring.

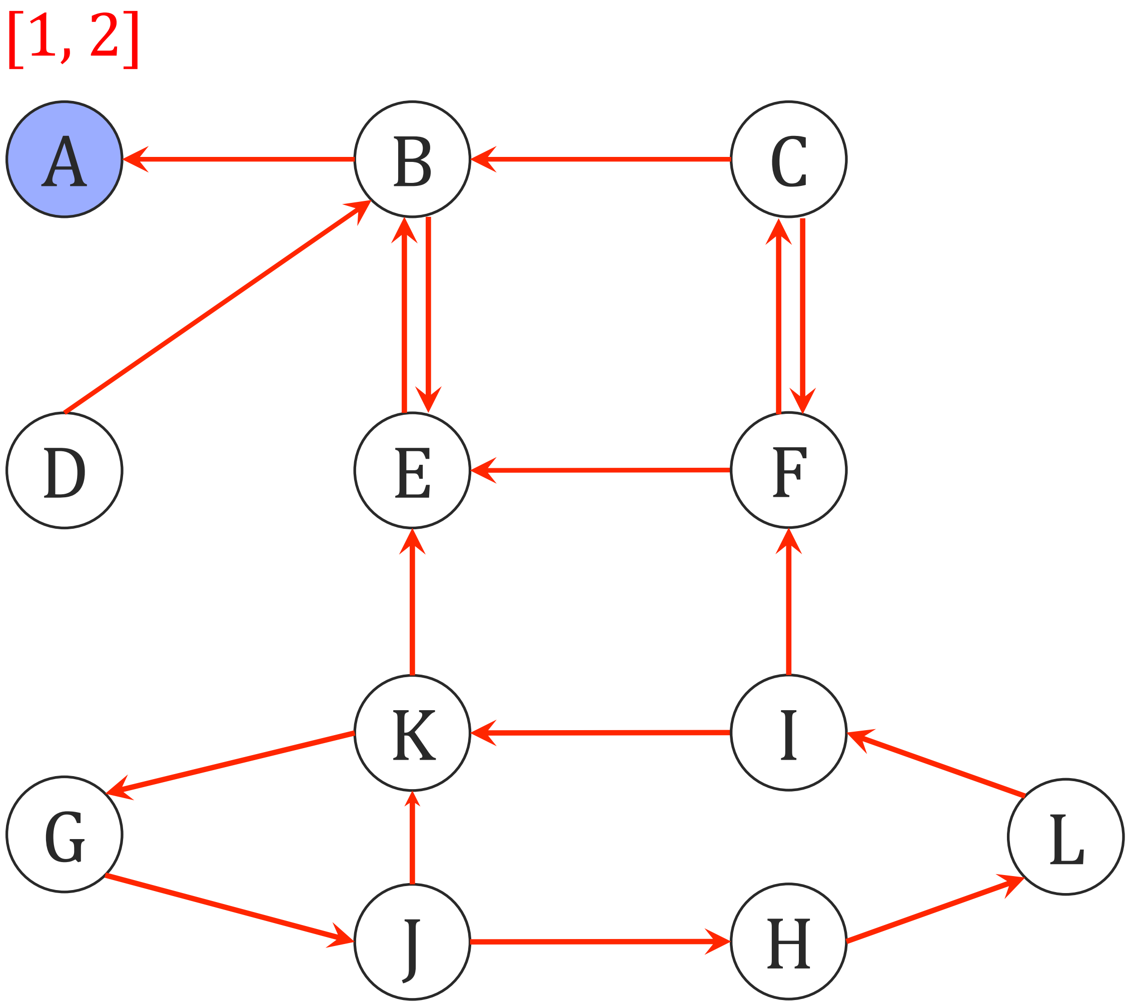
Finished exploring



In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G





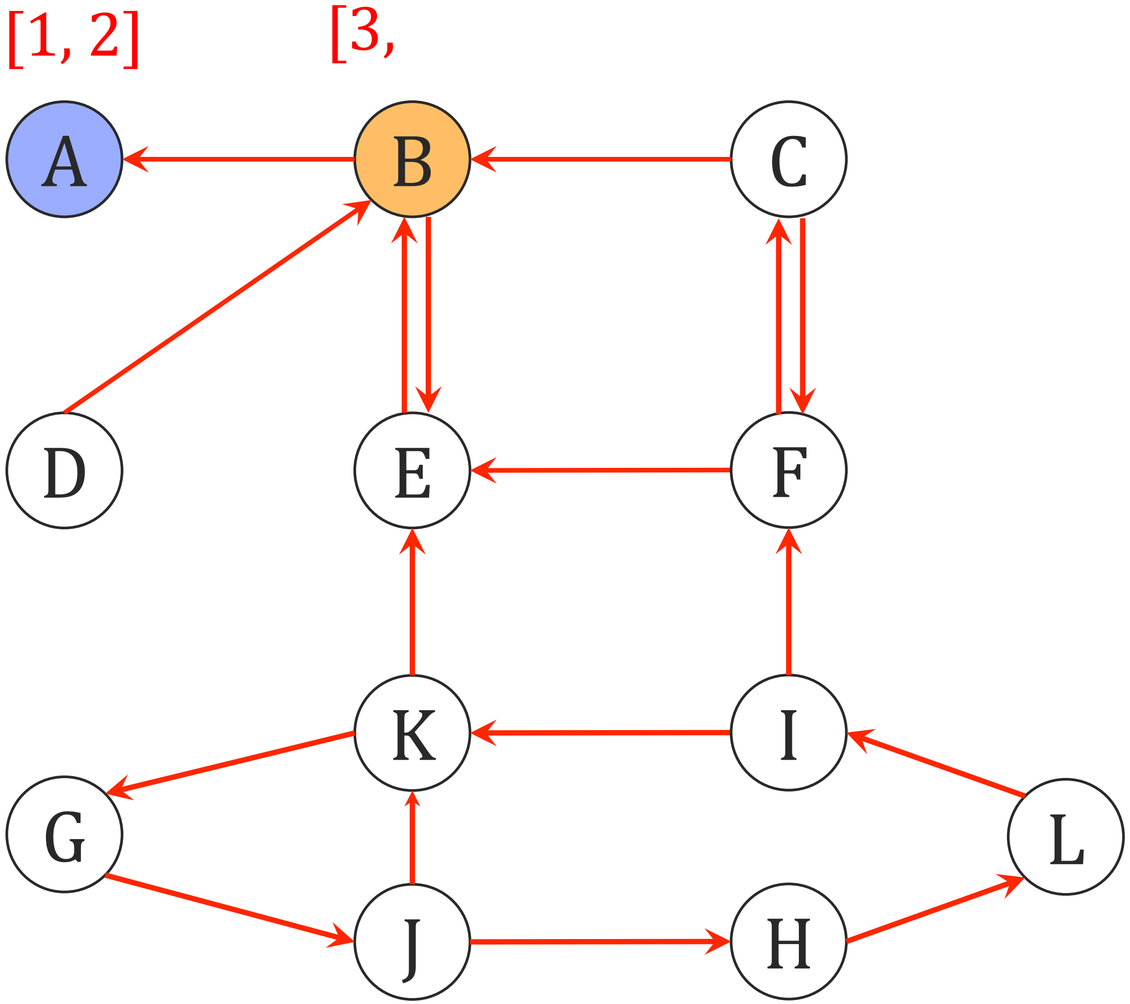
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

Been there, still exploring.

Finished exploring



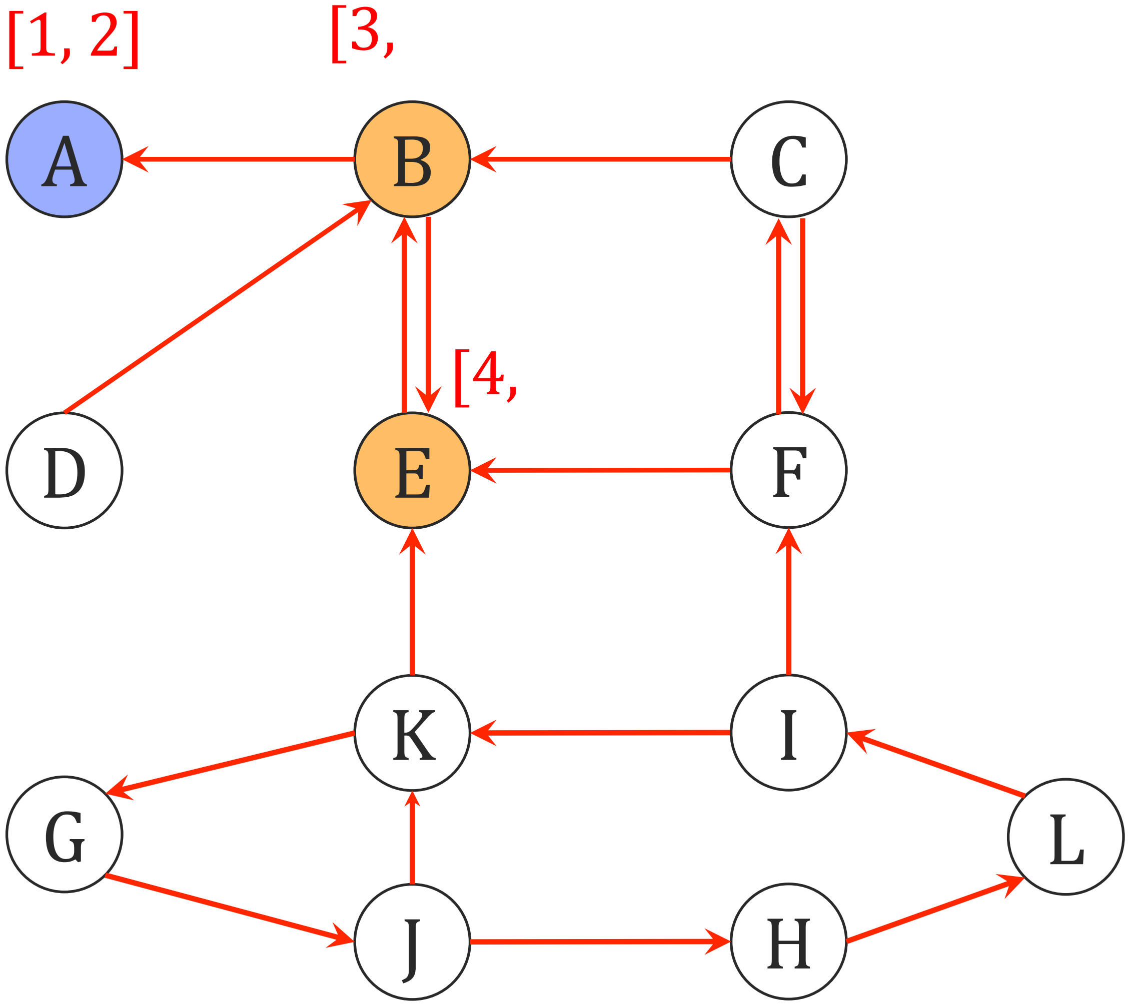
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

Been there, still exploring.

Finished exploring



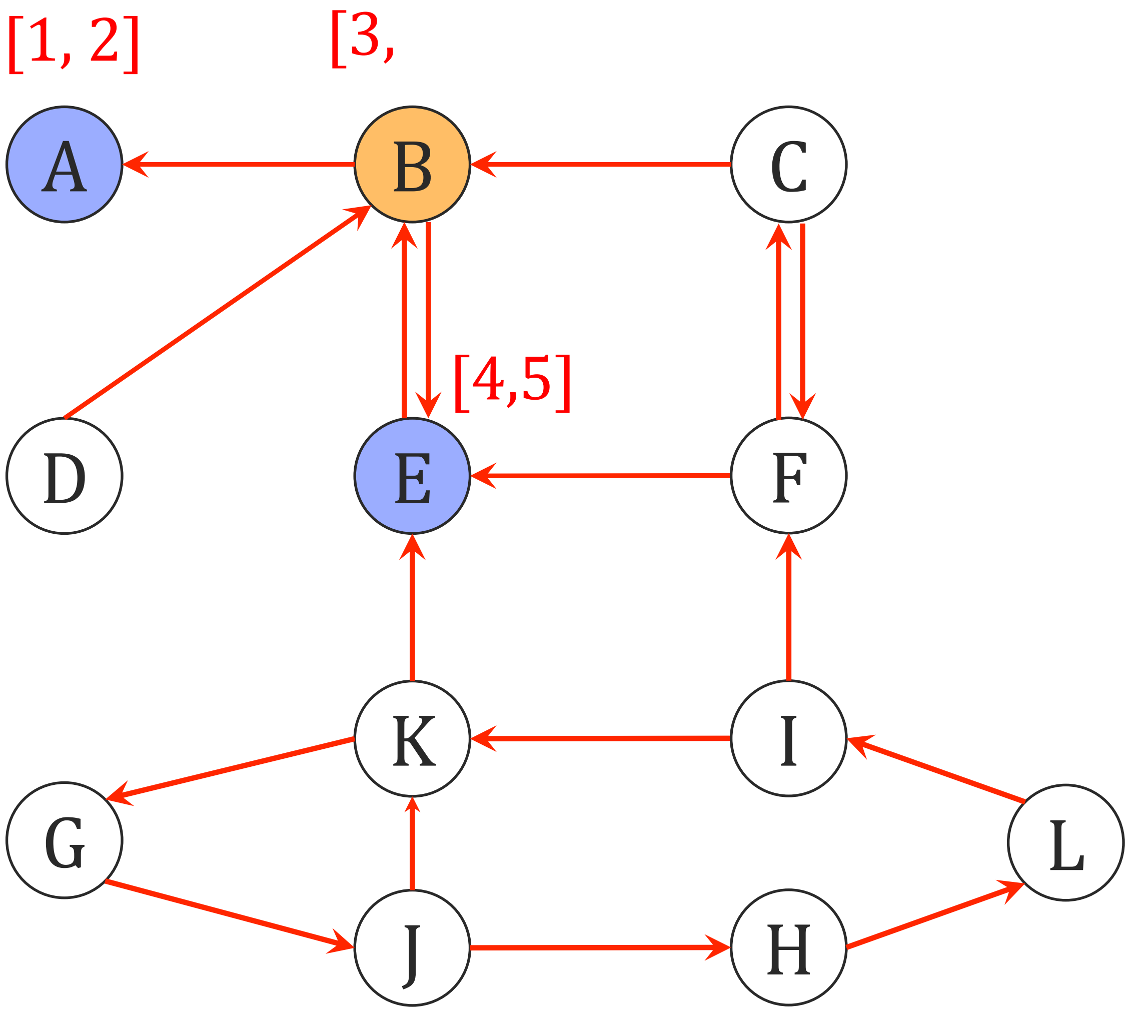
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

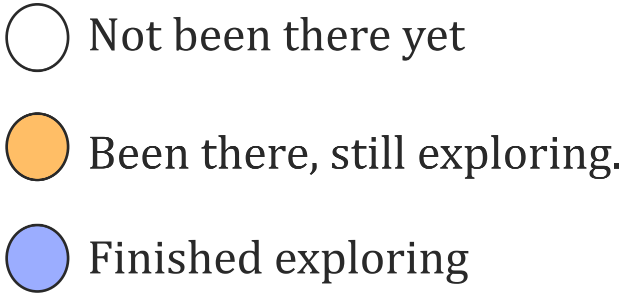
Been there, still exploring.

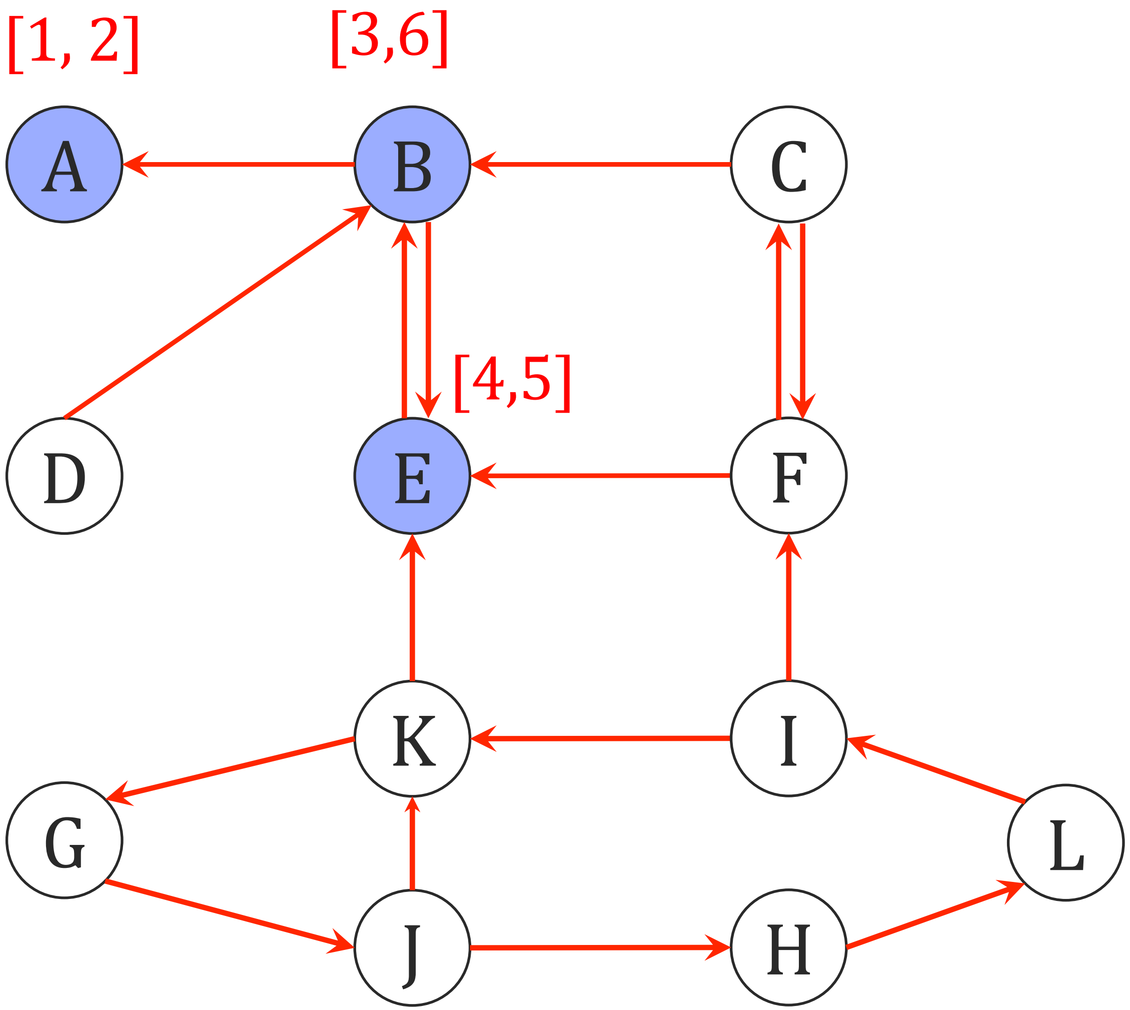
Finished exploring



In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

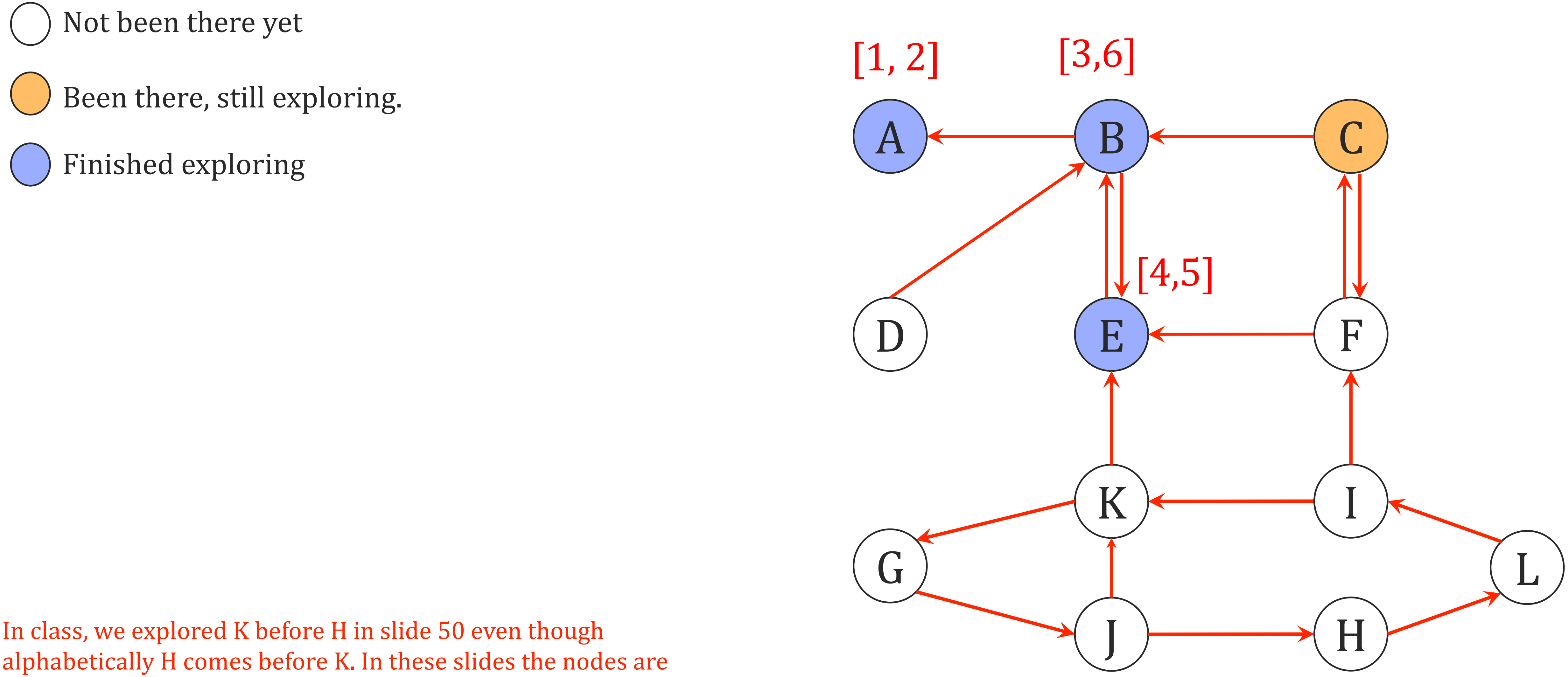
Alg step 2: Run DFS on the reverse of G





In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G



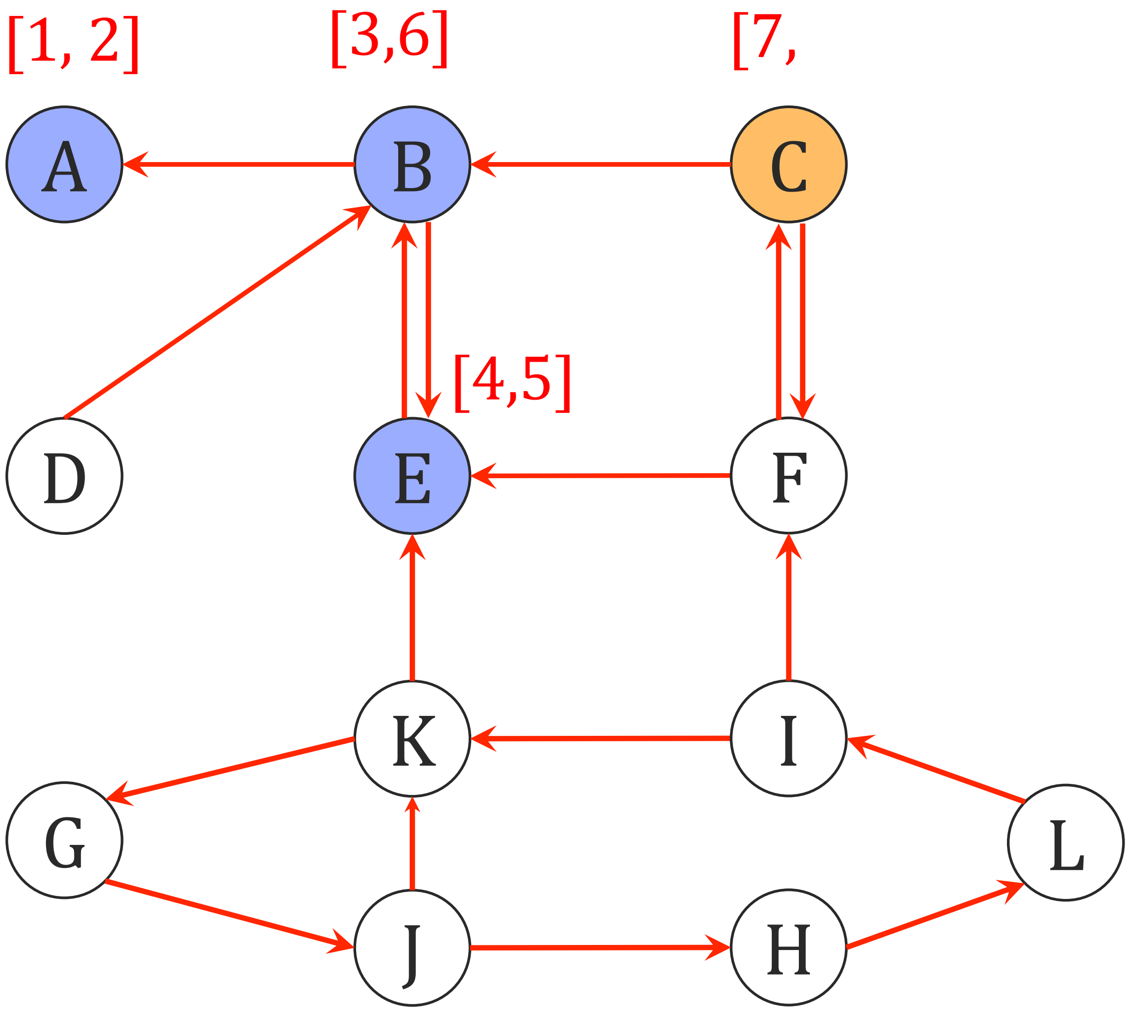
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

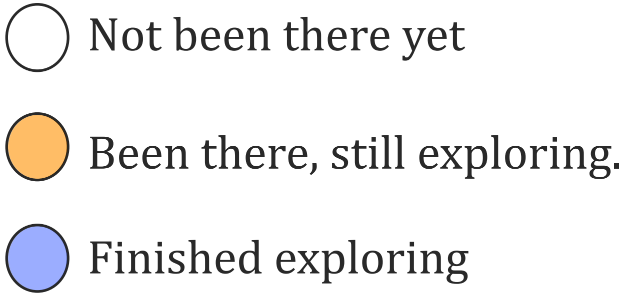
Been there, still exploring.

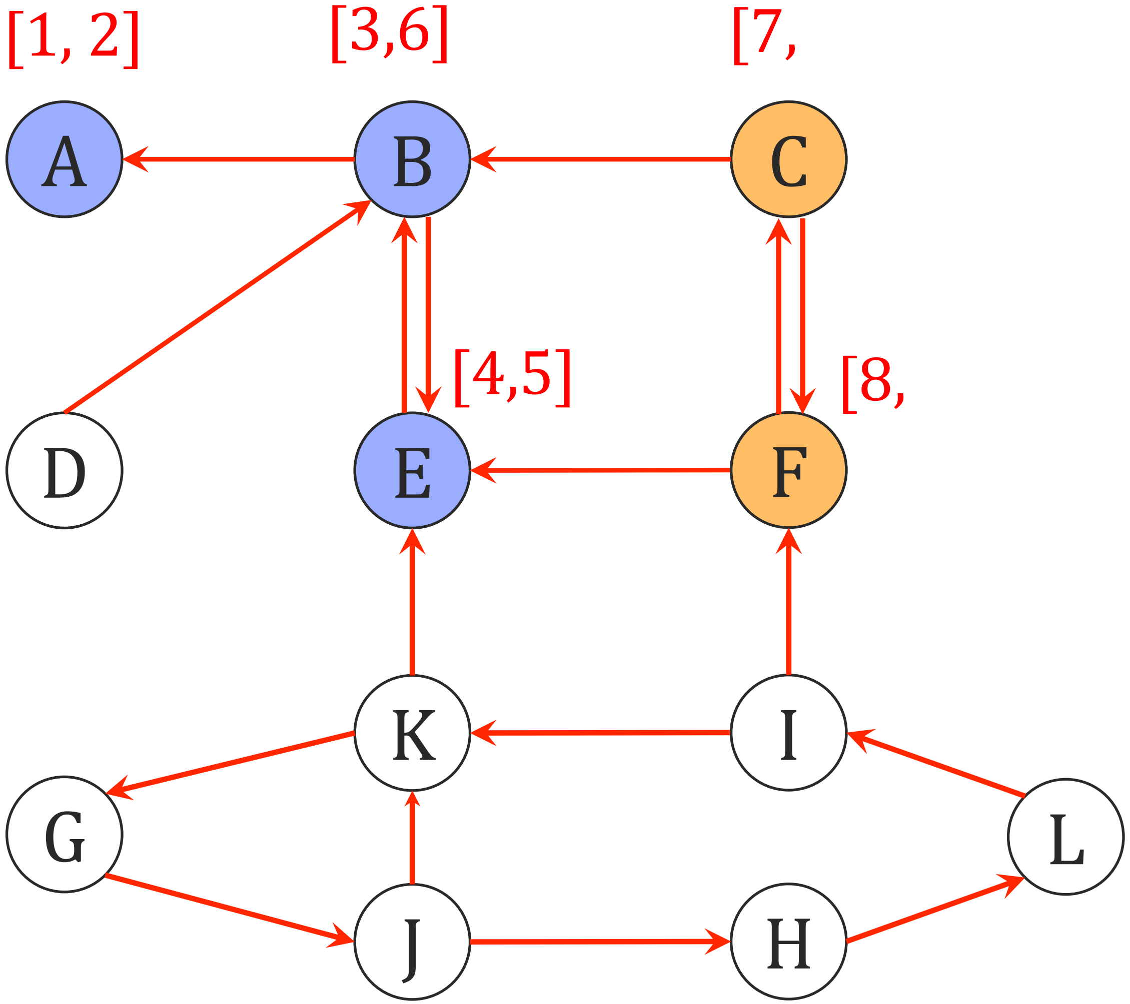
Finished exploring



In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G





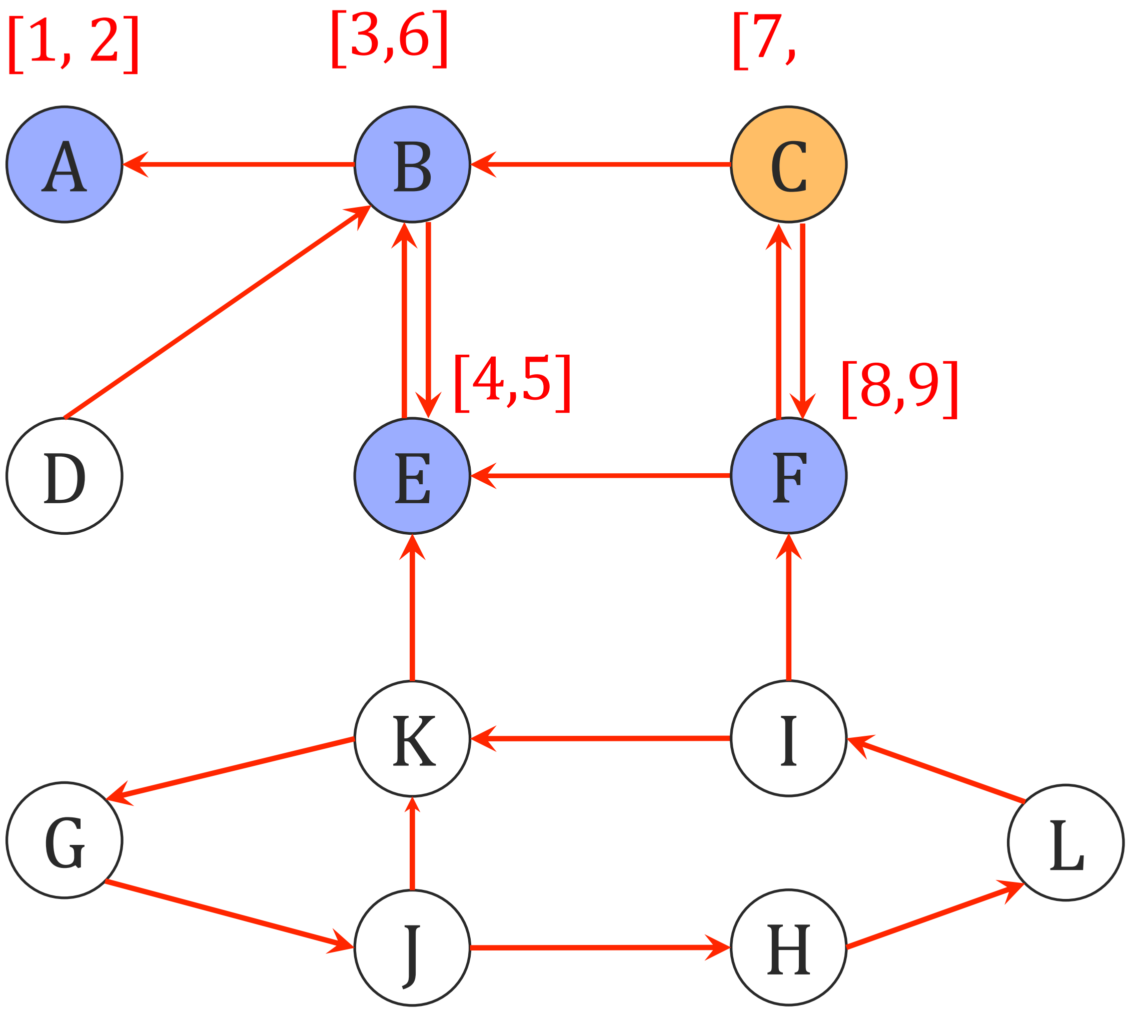
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

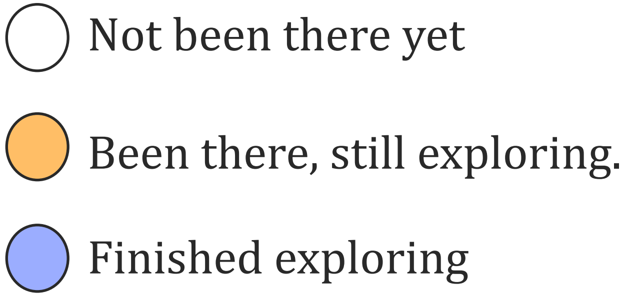
Been there, still exploring.

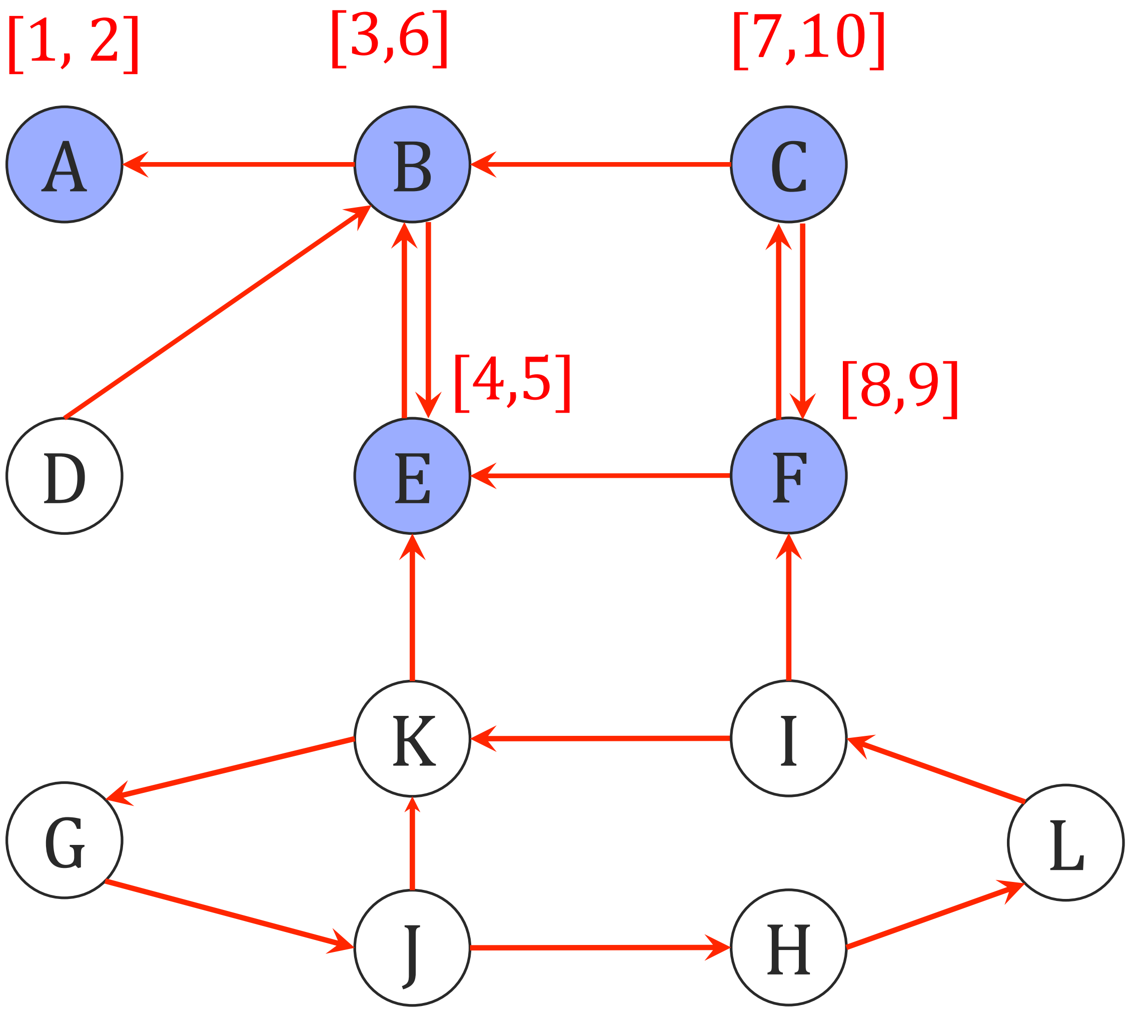
Finished exploring



In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

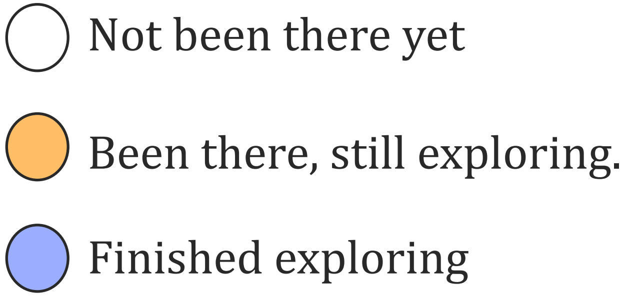
Alg step 2: Run DFS on the reverse of G

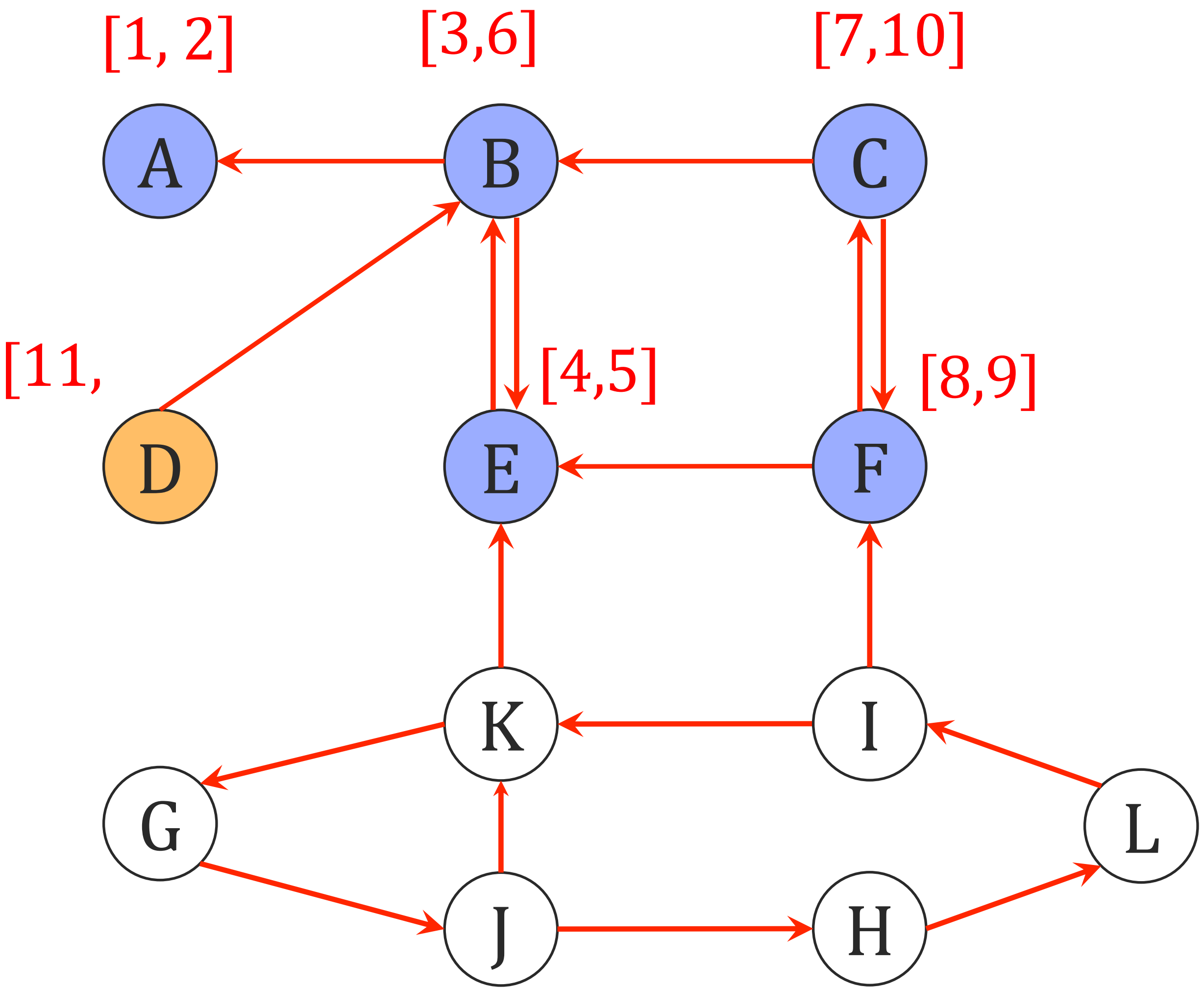




In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

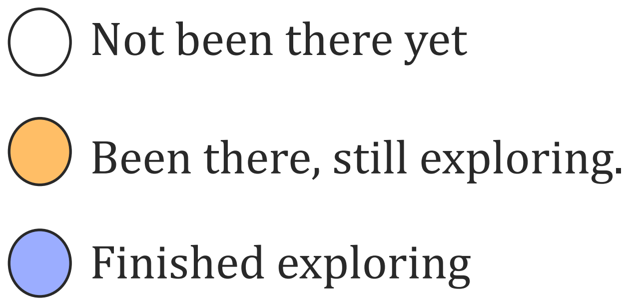
Alg step 2: Run DFS on the reverse of G

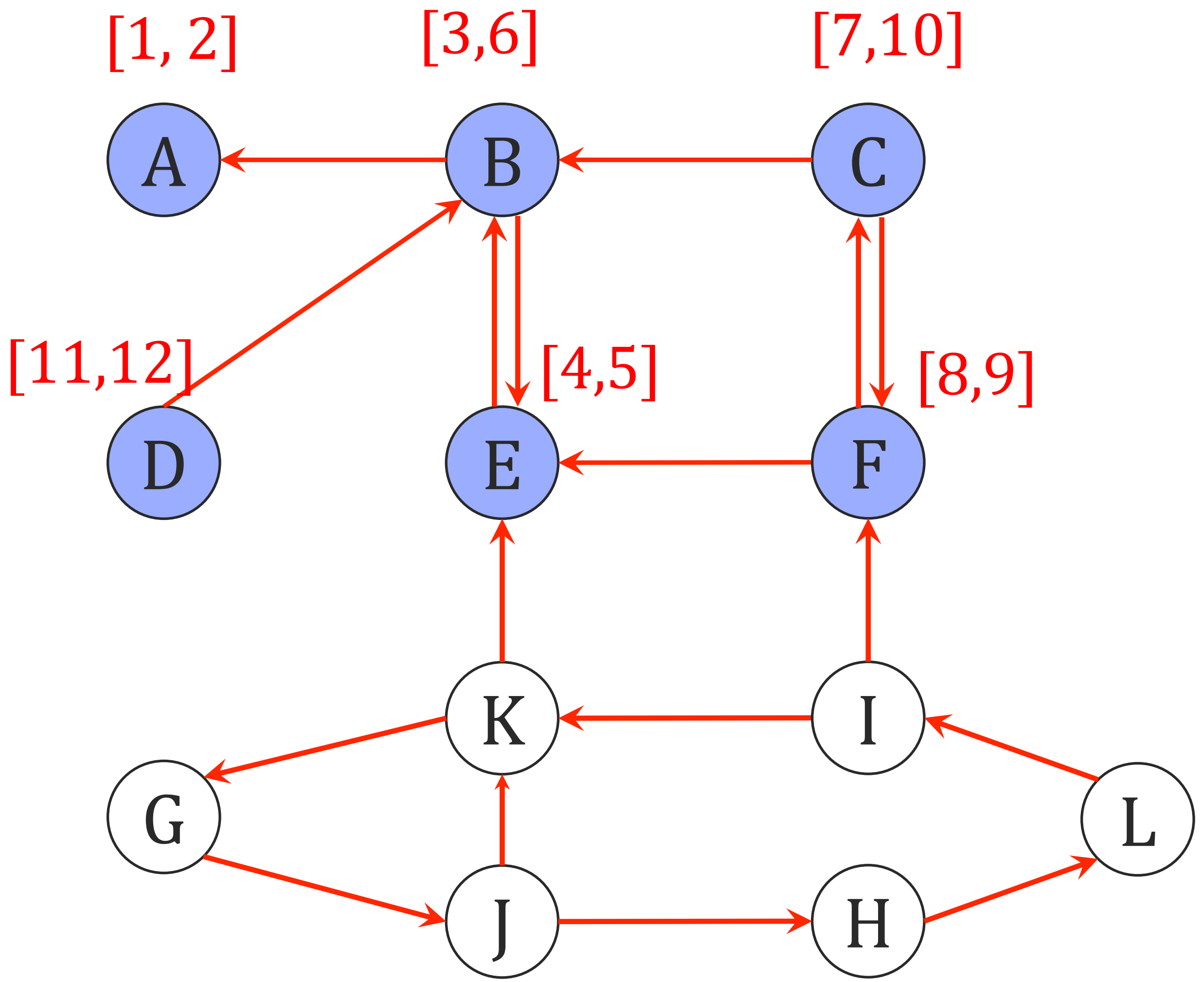




In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

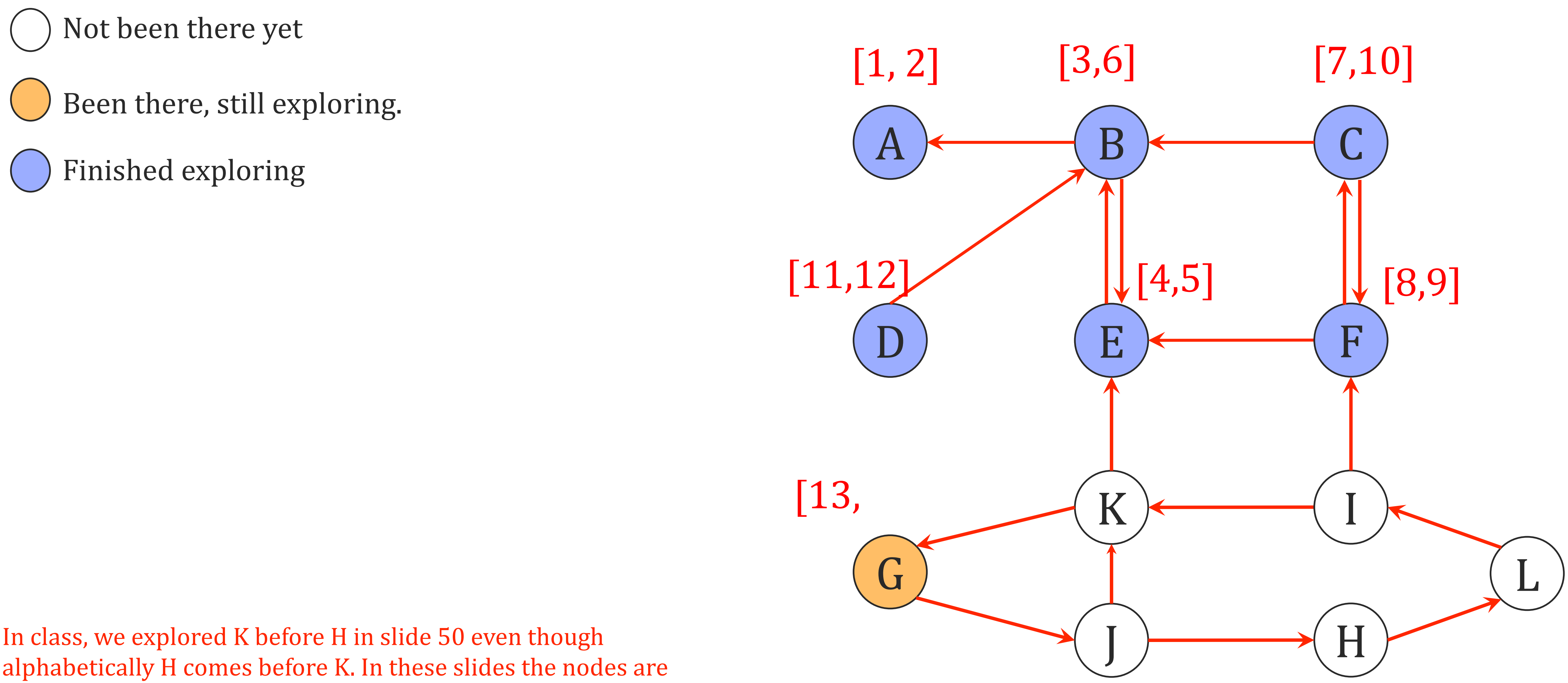
Alg step 2: Run DFS on the reverse of G





In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G



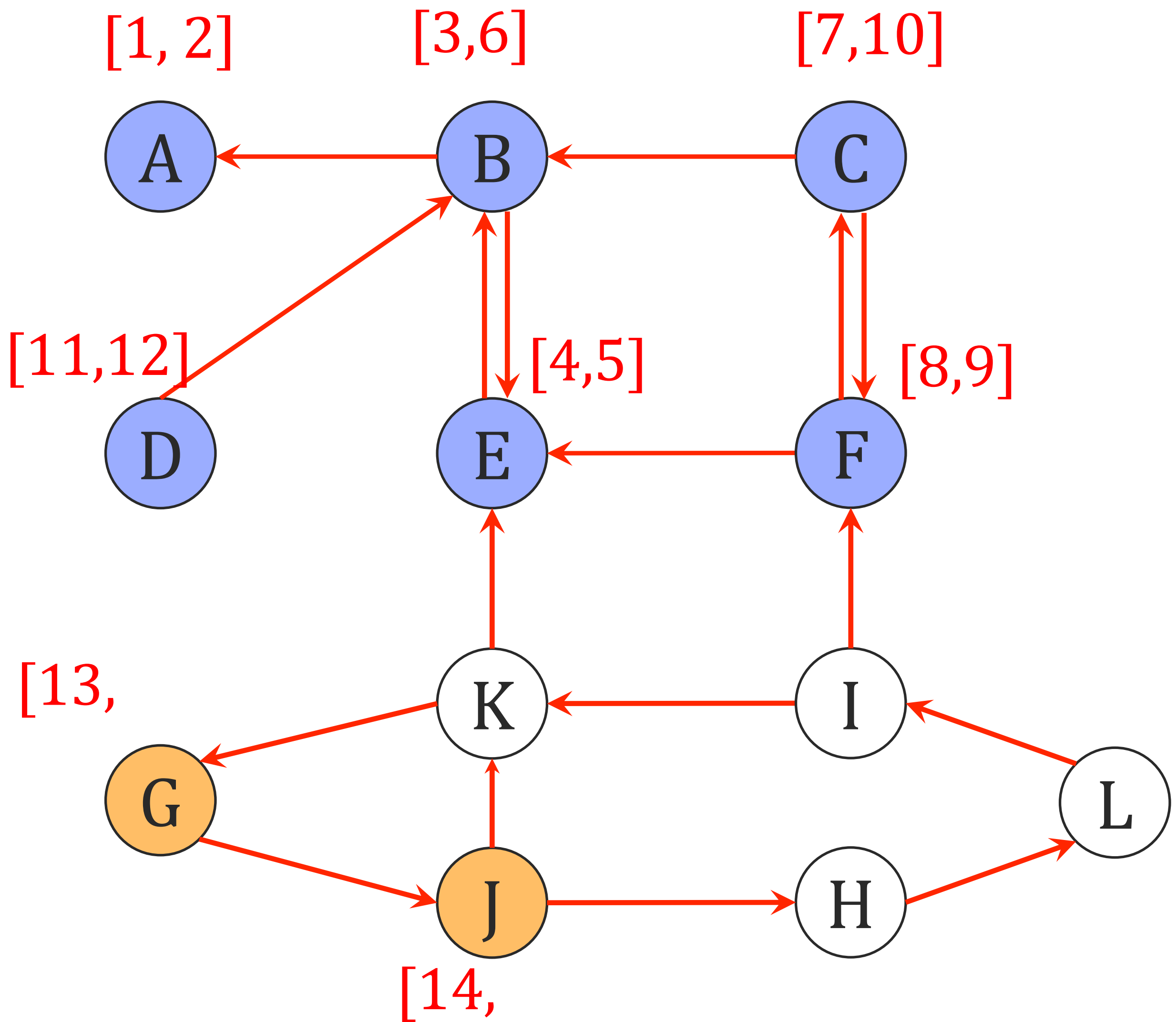
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

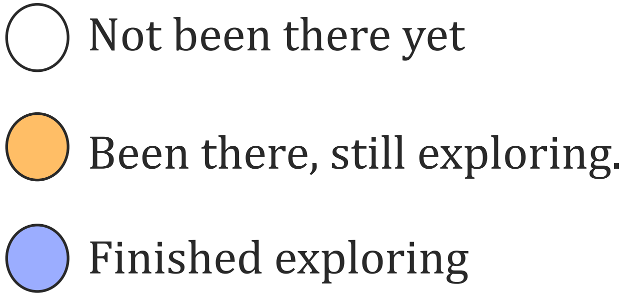
Been there, still exploring.

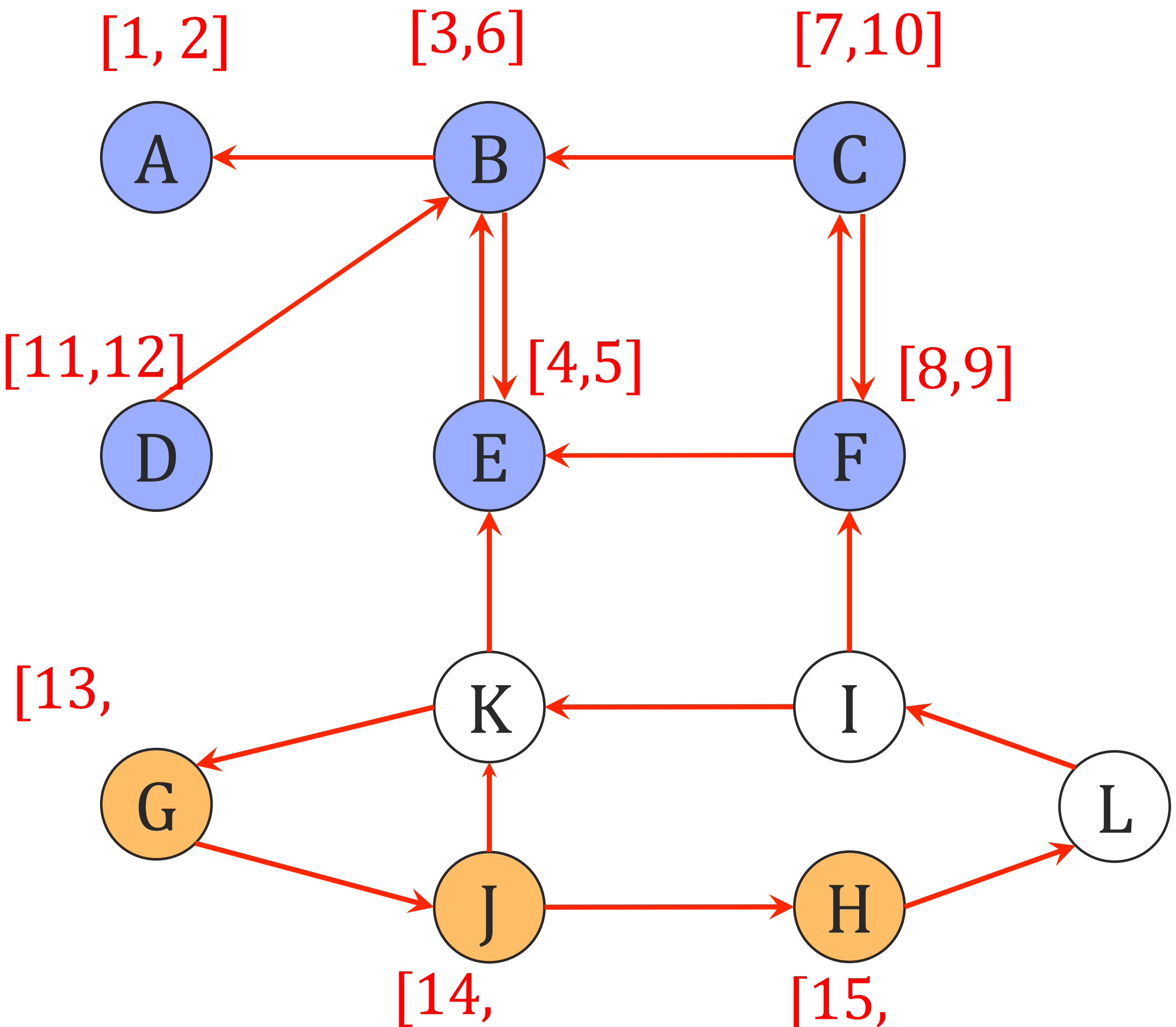
Finished exploring



In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

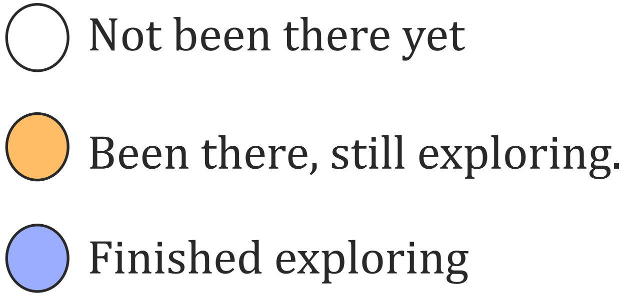
Alg step 2: Run DFS on the reverse of G

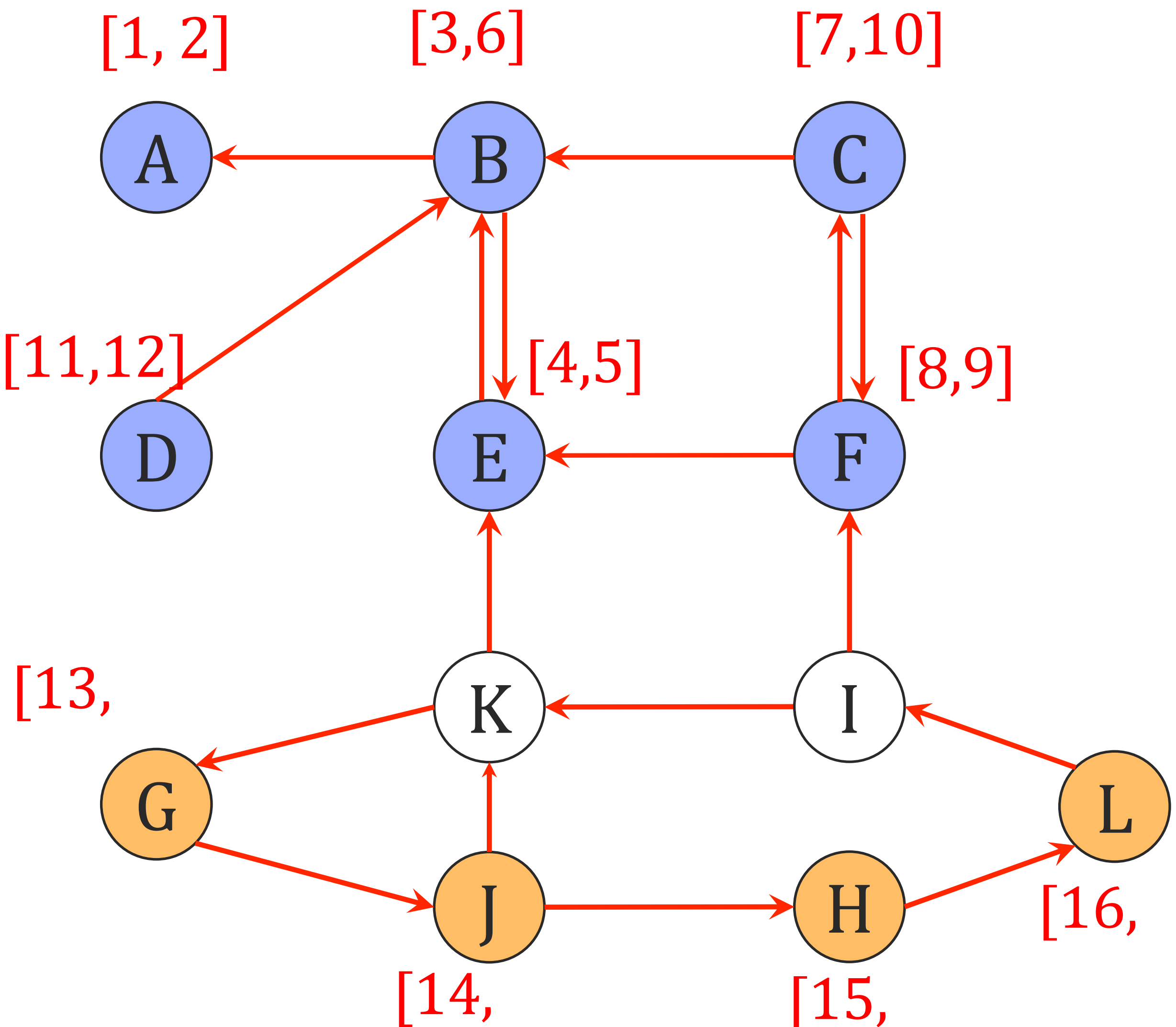




In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G





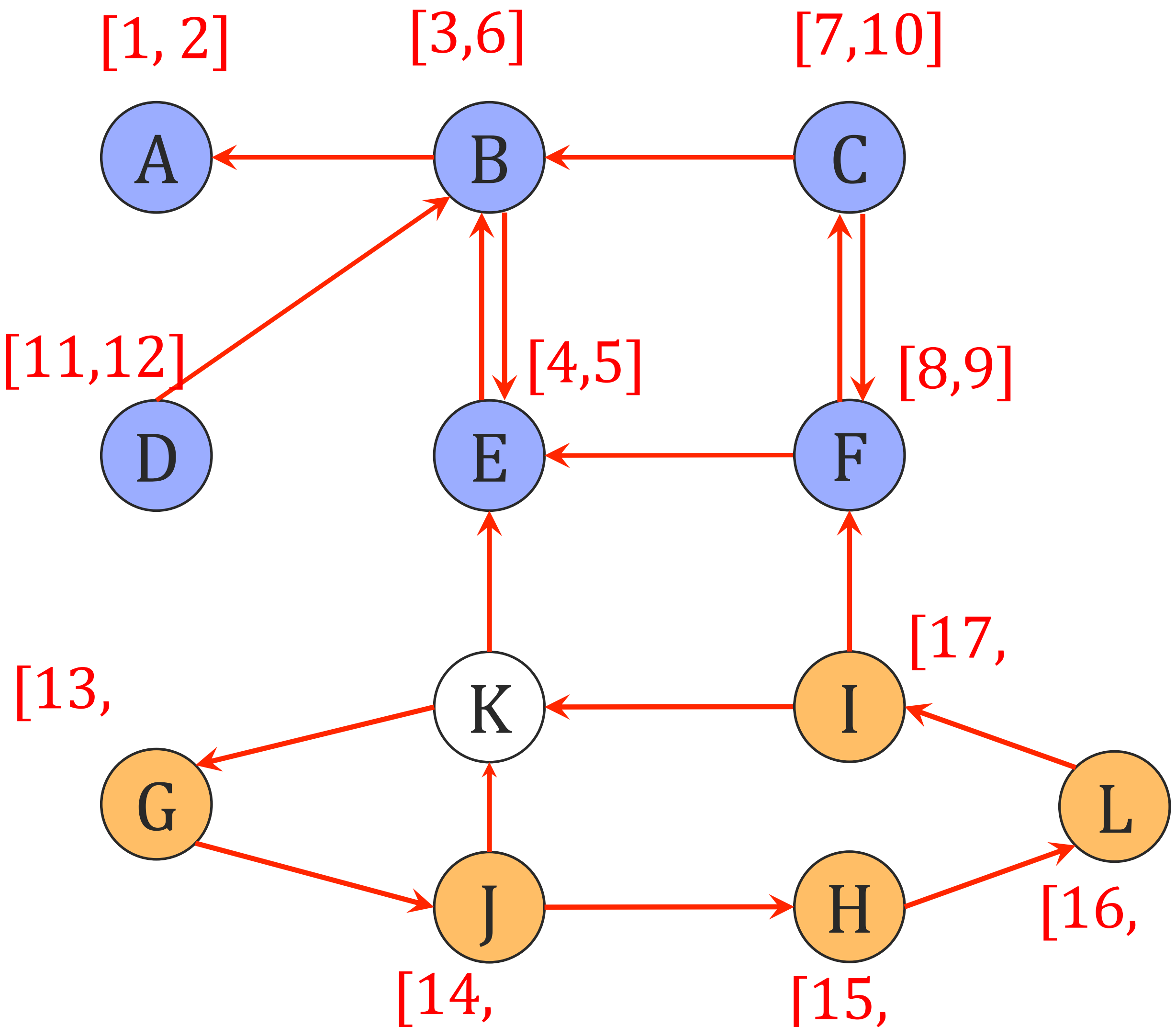
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

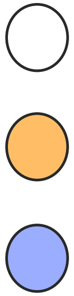
Been there, still exploring.

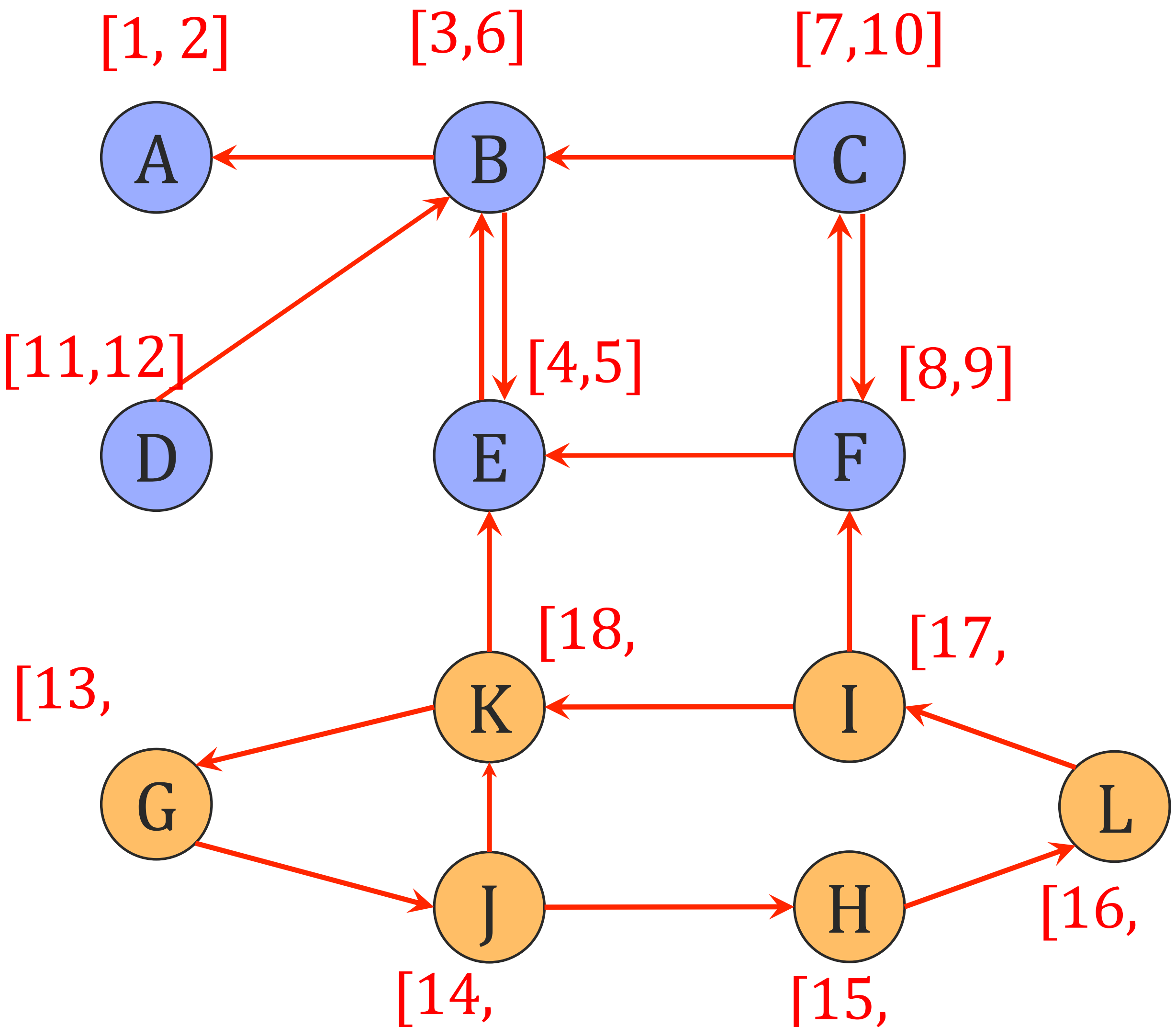
Finished exploring



In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

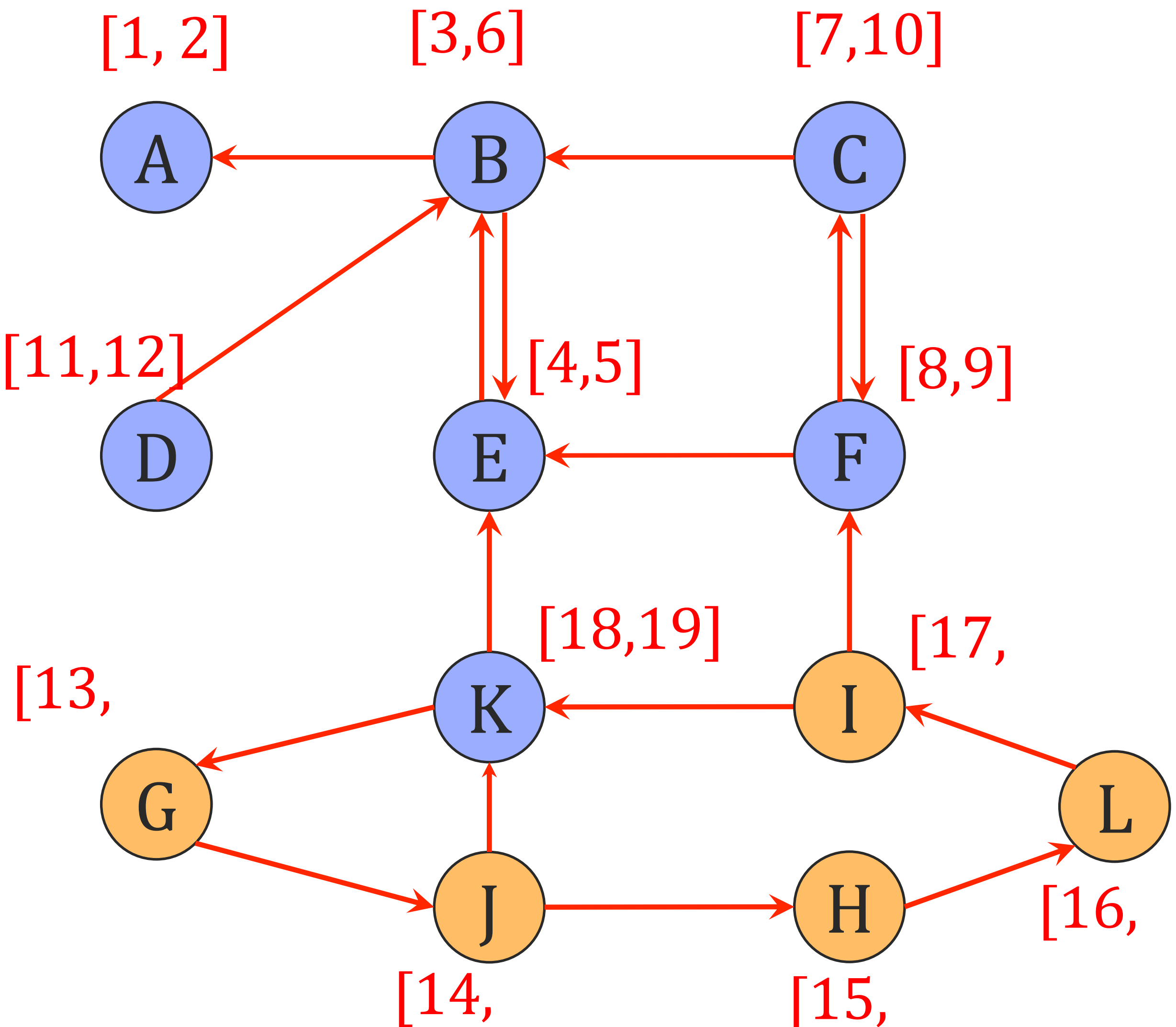




In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

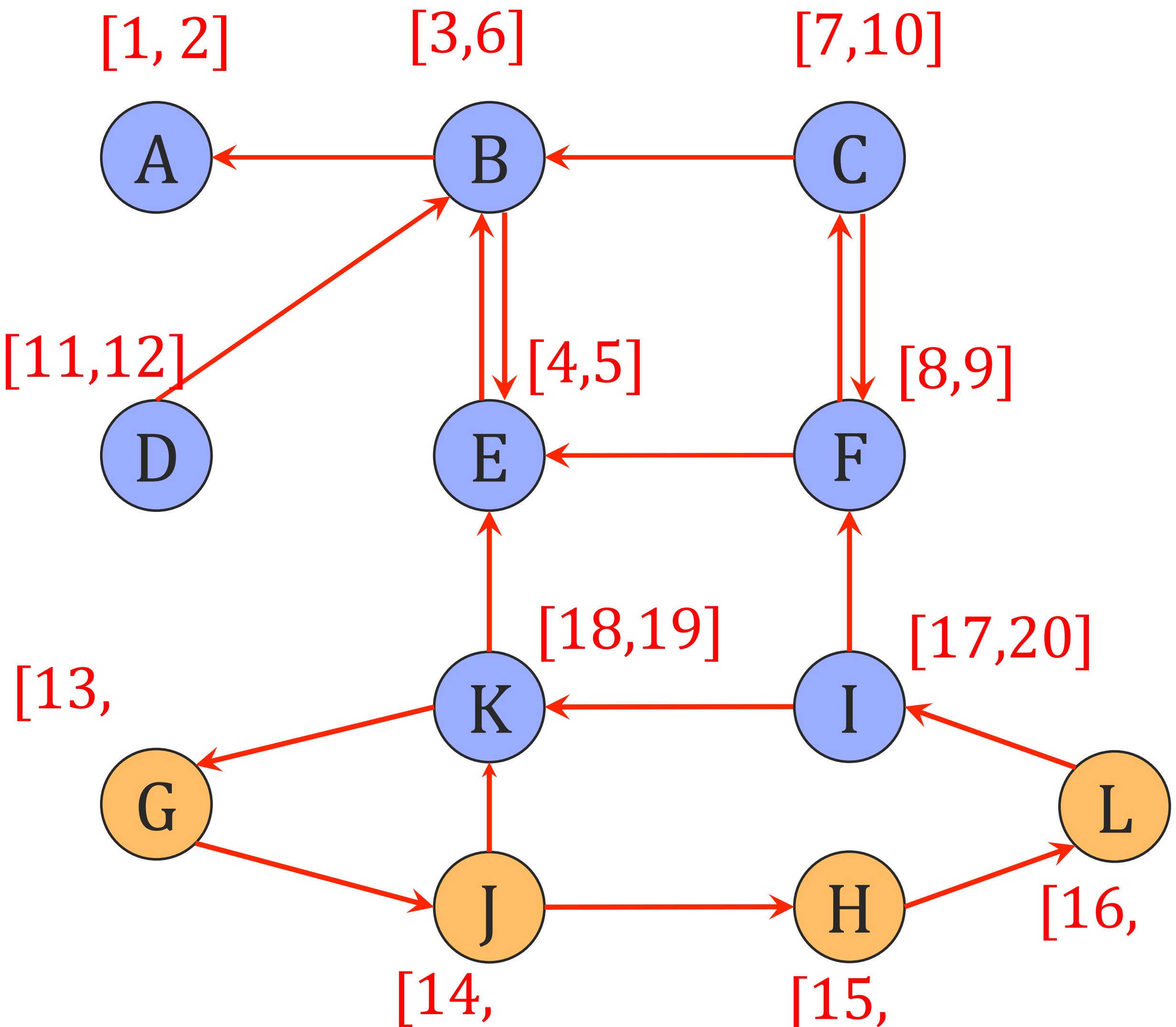
- Not been there yet
- Been there, still exploring.
- Finished exploring



In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

- Not been there yet
- Been there, still exploring.
- Finished exploring



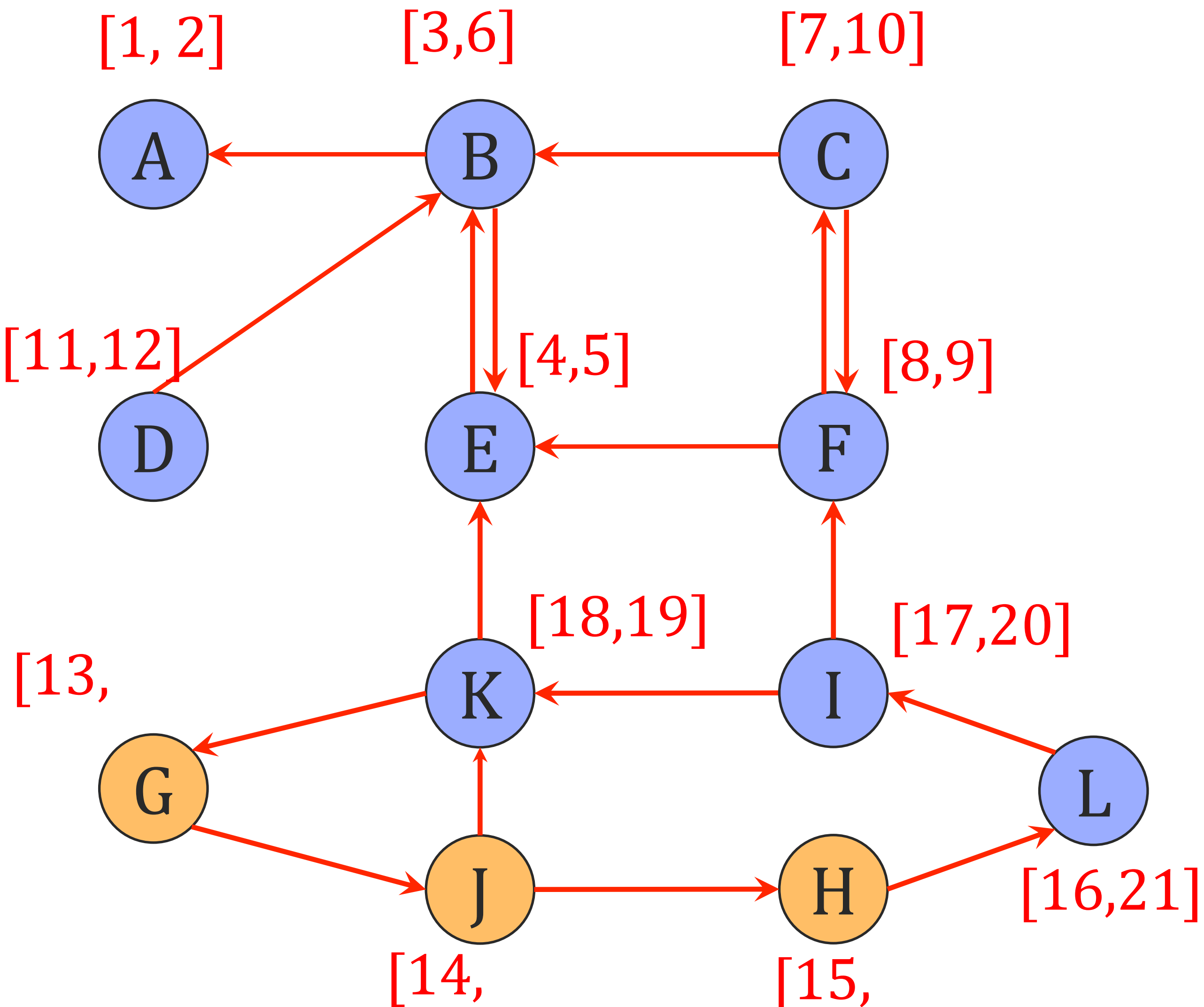
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

Been there, still exploring.

Finished exploring



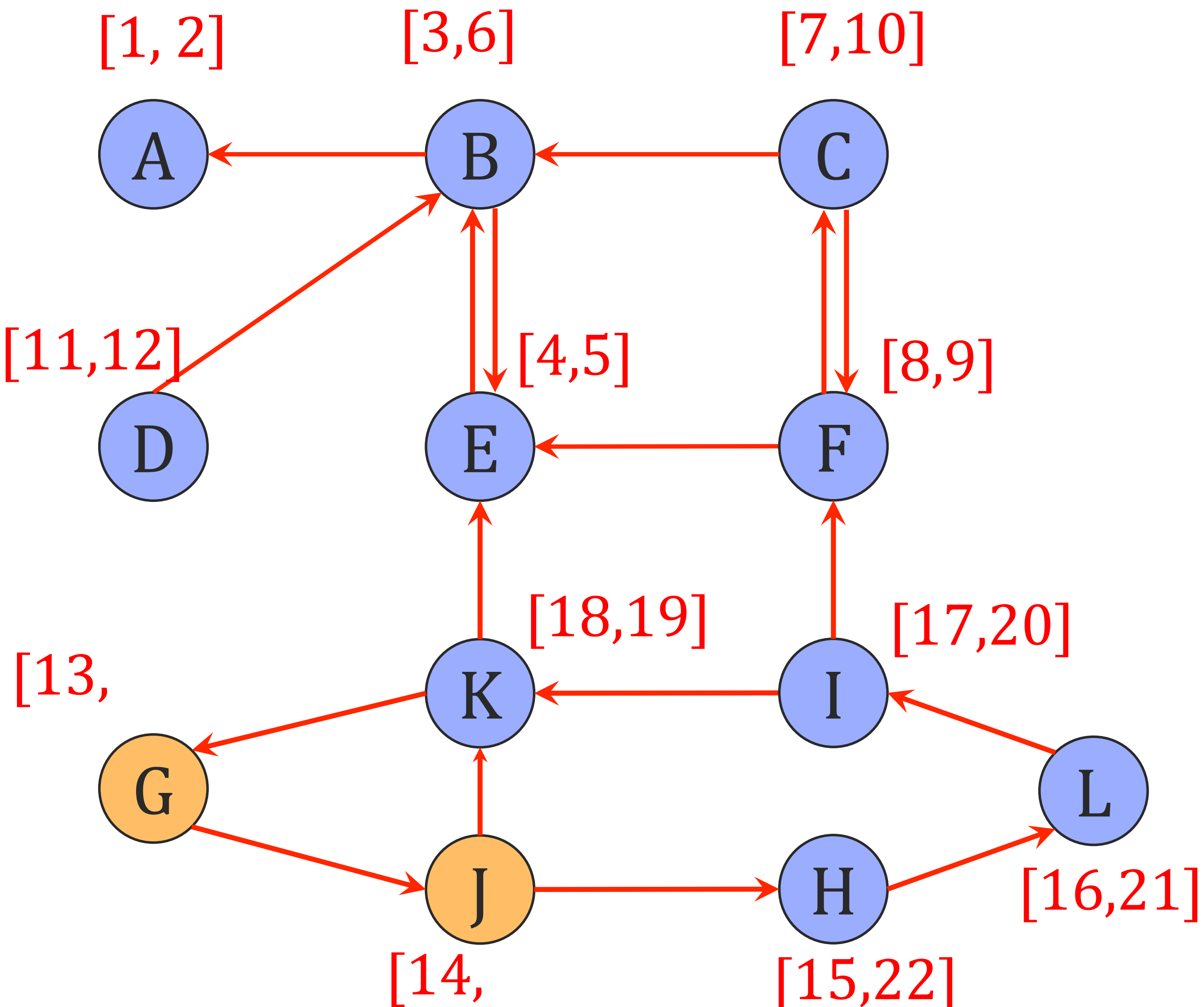
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

Been there, still exploring.

Finished exploring



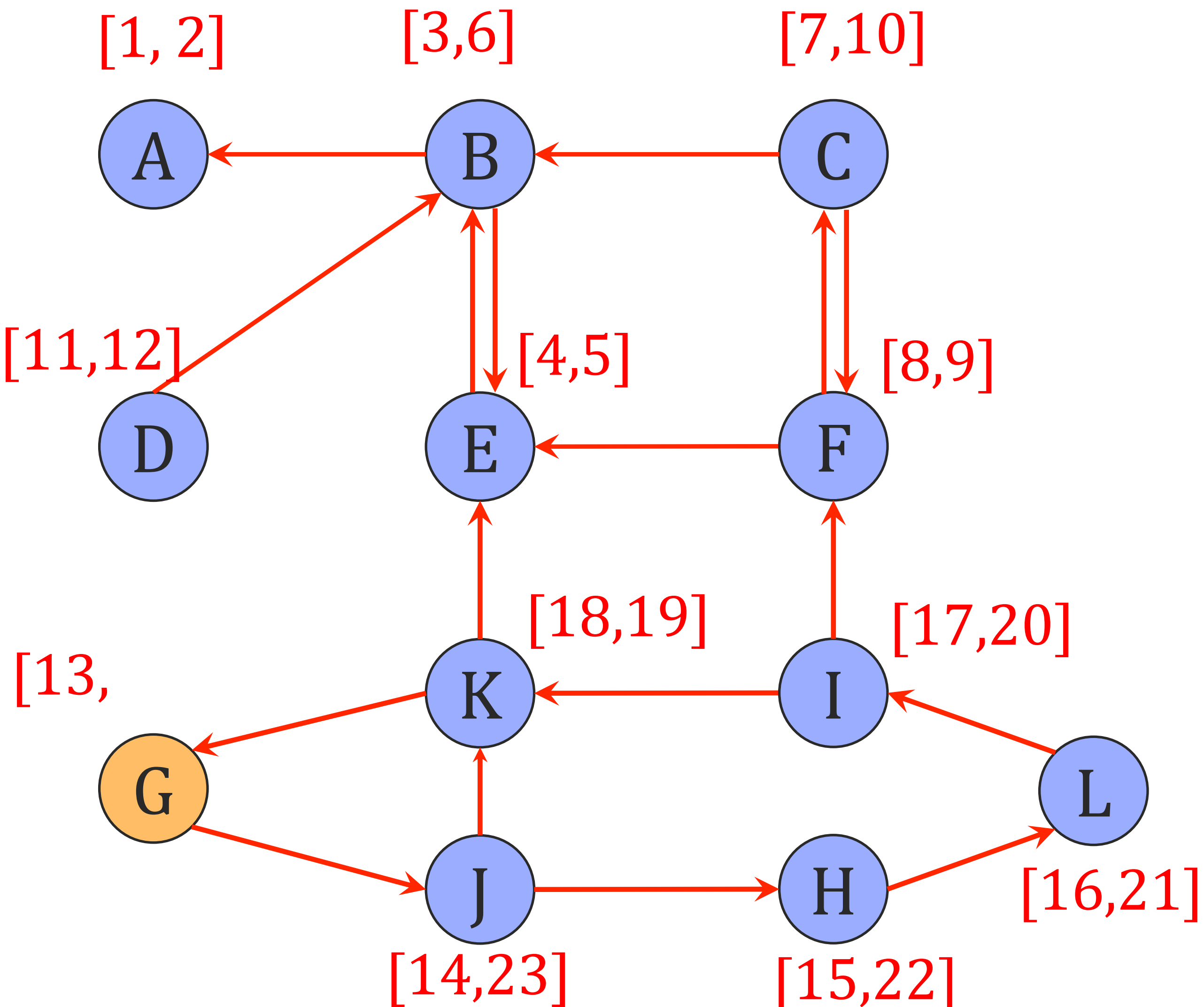
In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

Not been there yet

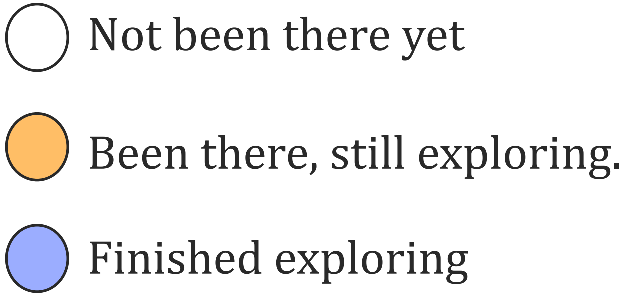
Been there, still exploring.

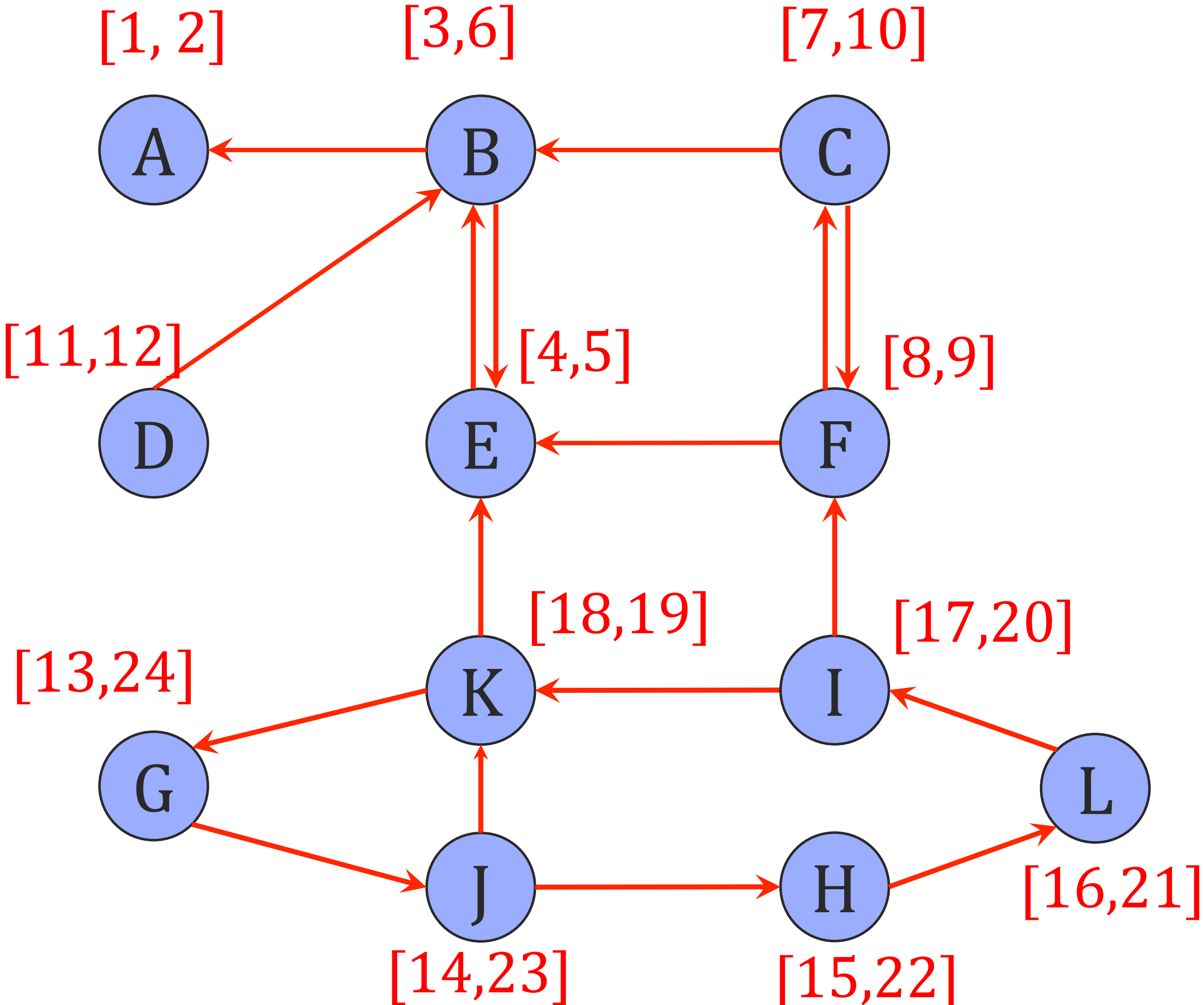
Finished exploring



In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

Alg step 2: Run DFS on the reverse of G

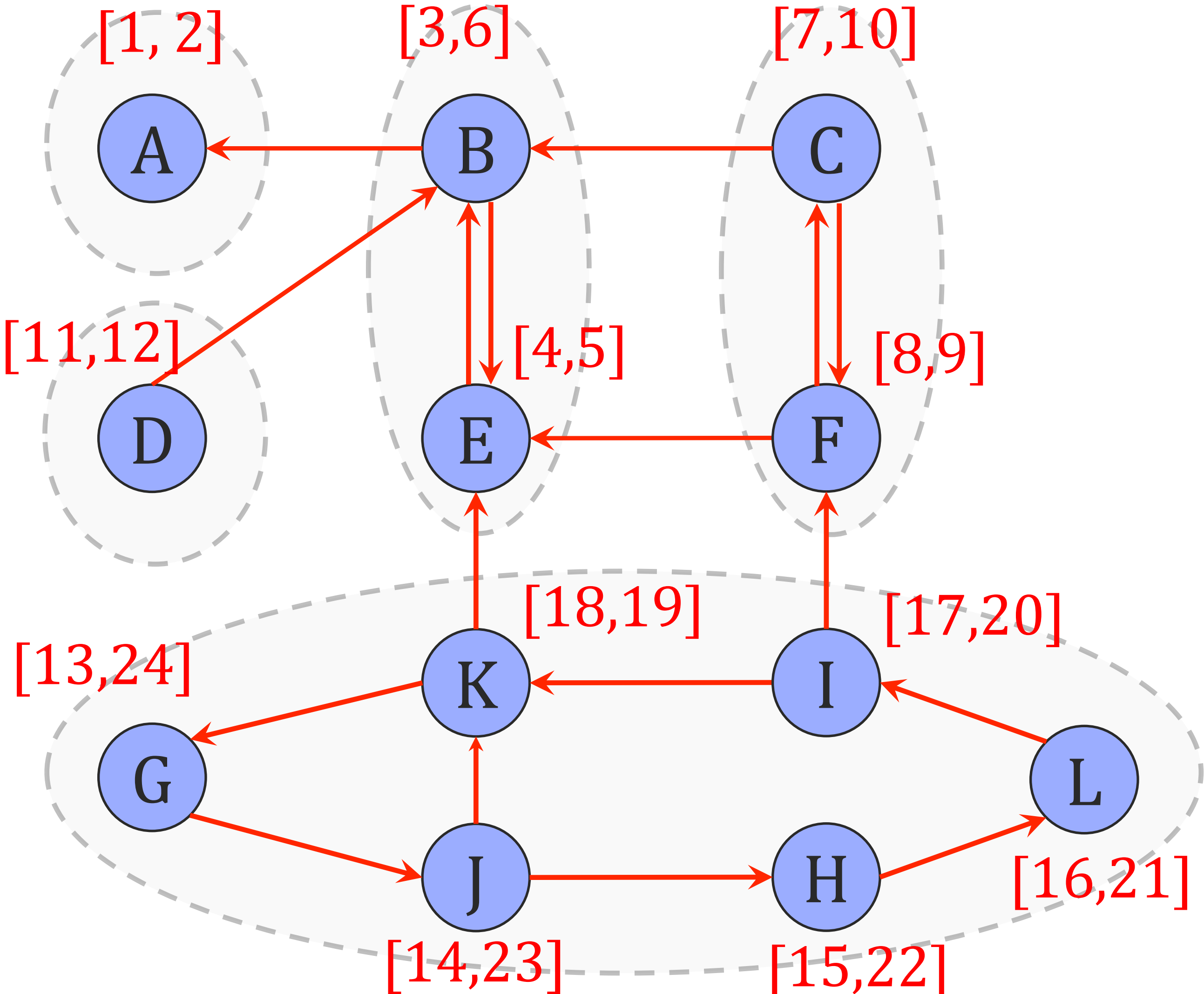




In class, we explored K before H in slide 50 even though alphabetically H comes before K. In these slides the nodes are altered to reflect the correct alphabetic order of exploration.

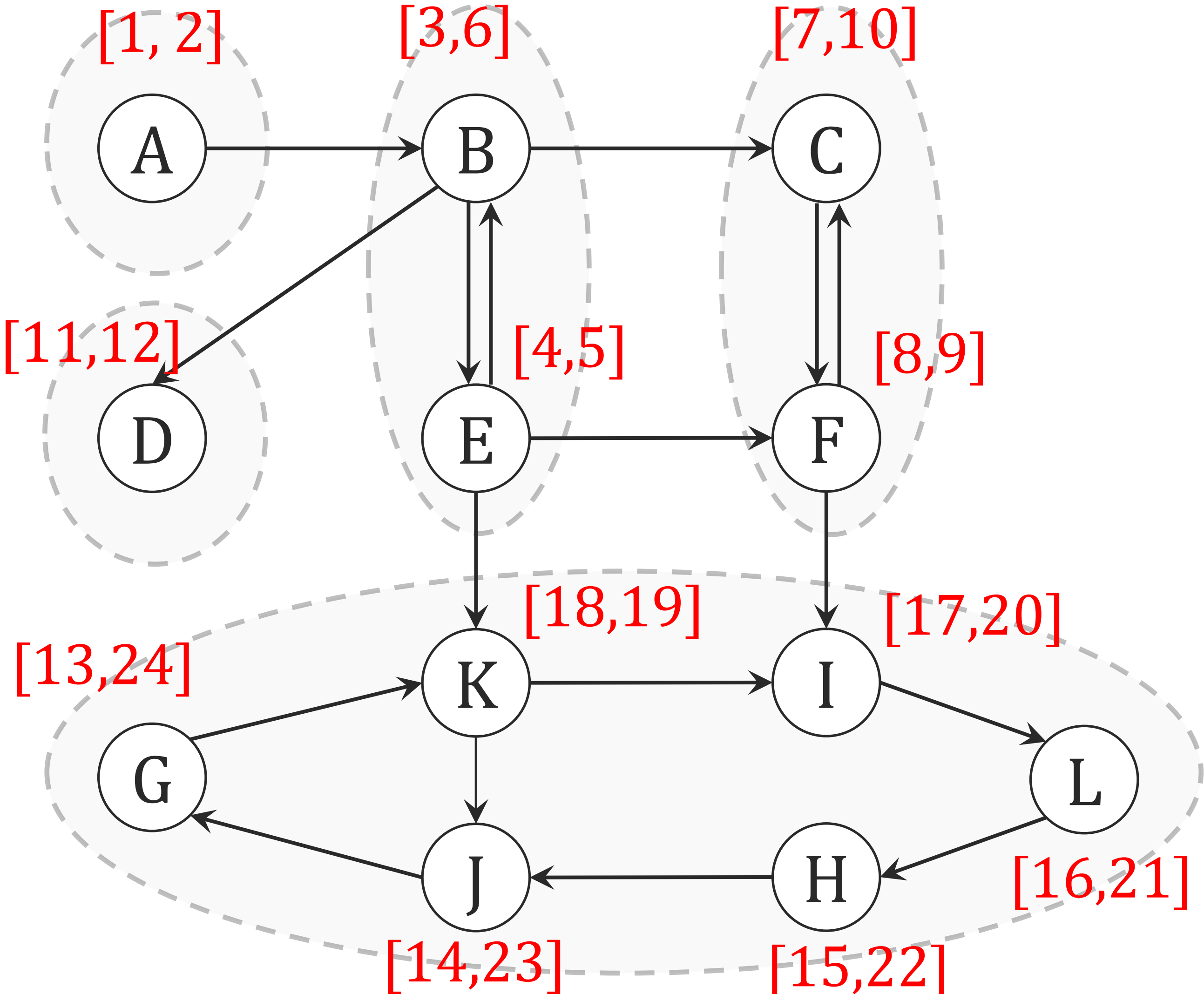
Sink node and the highest post number

Consider the graph G, not reverse of G.



Sink node and the highest post number

Consider the graph G, not reverse of G.

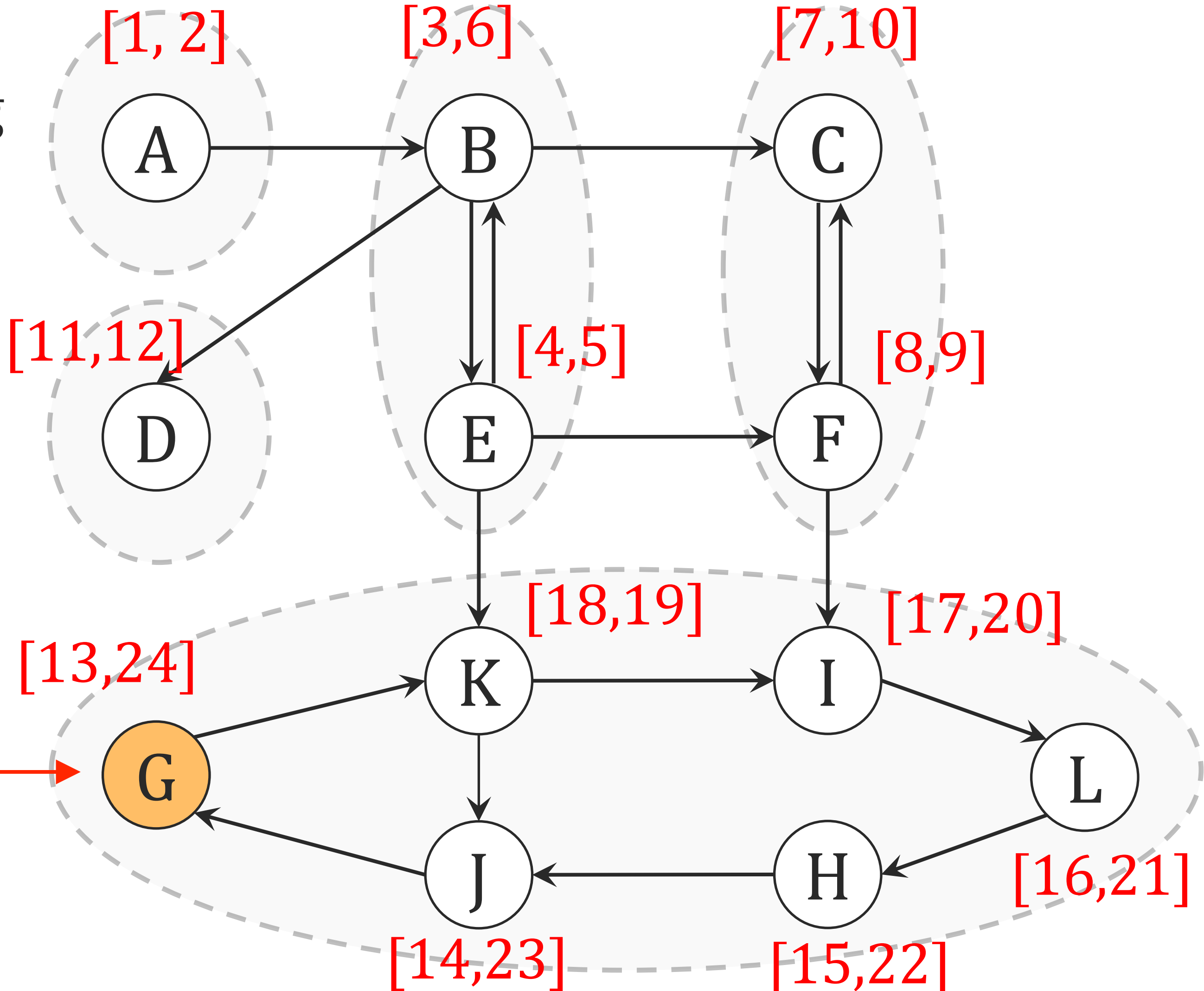


Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

Run DFS and call explore in the decreasing order of post-r numbers.

Highest post-r

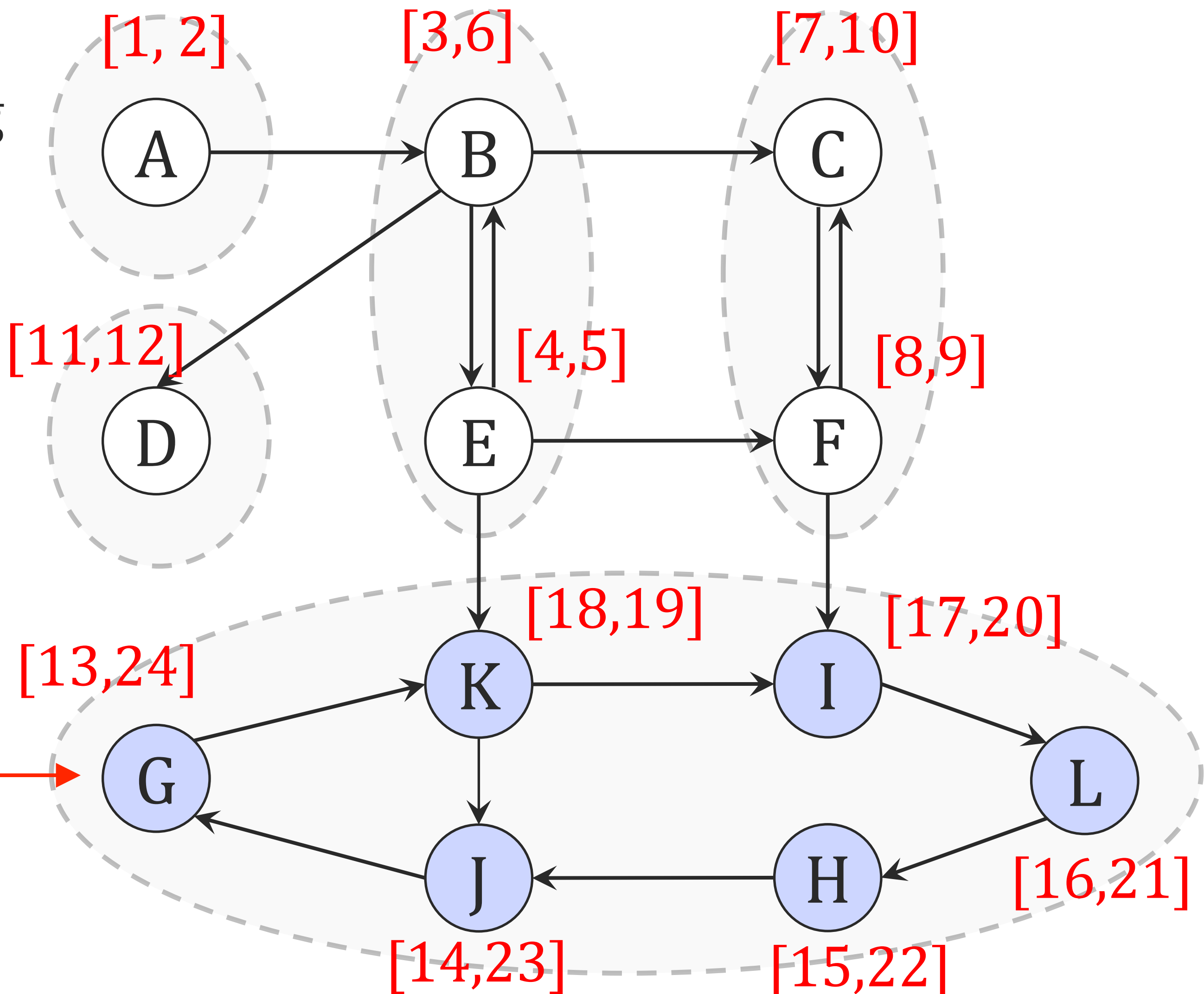


Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

Run DFS and call explore in the decreasing order of post-r numbers.

Highest post-r



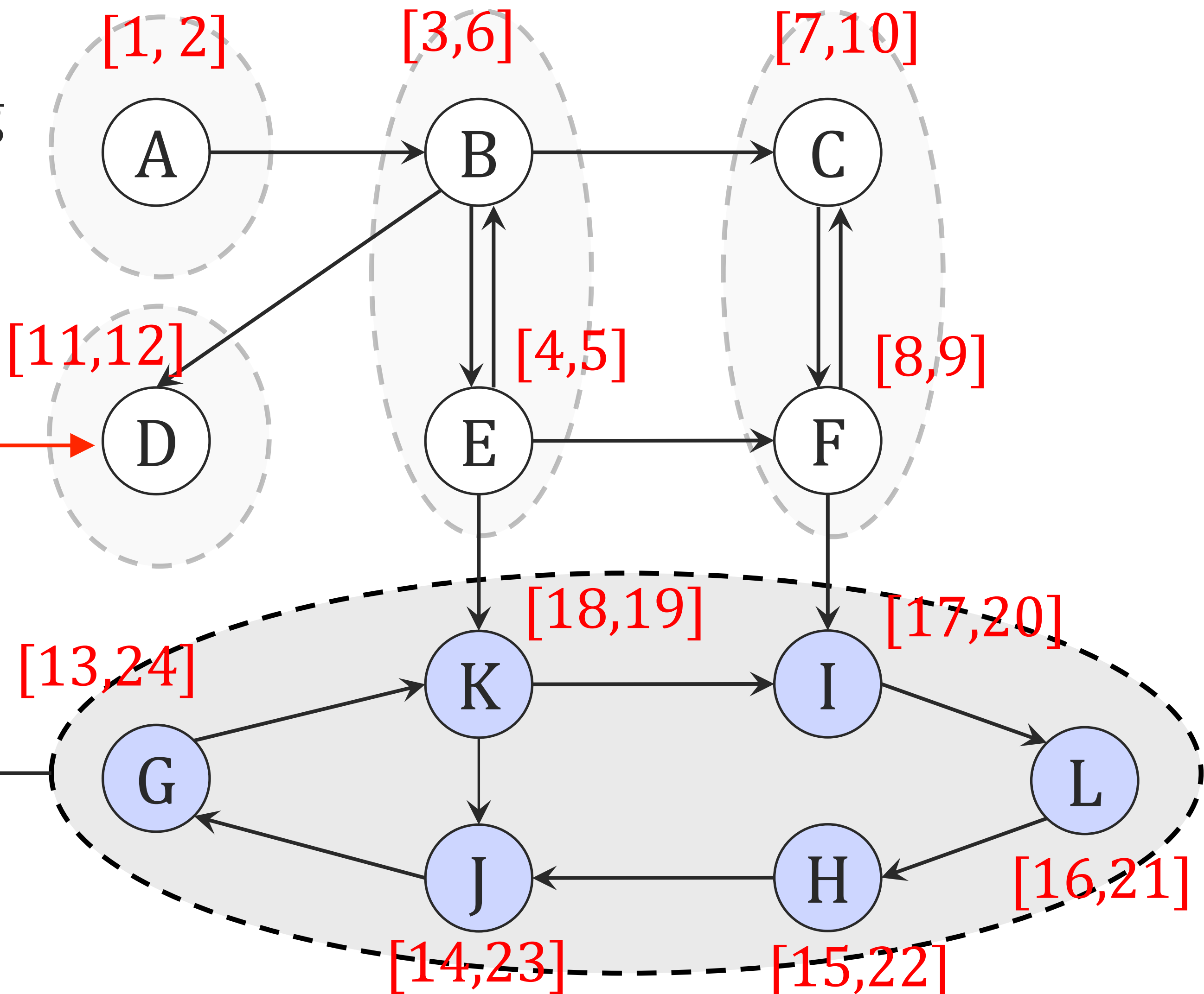
Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

Run DFS and call explore in the decreasing order of post-r numbers.

Highest remaining post-r

sccnum = 1



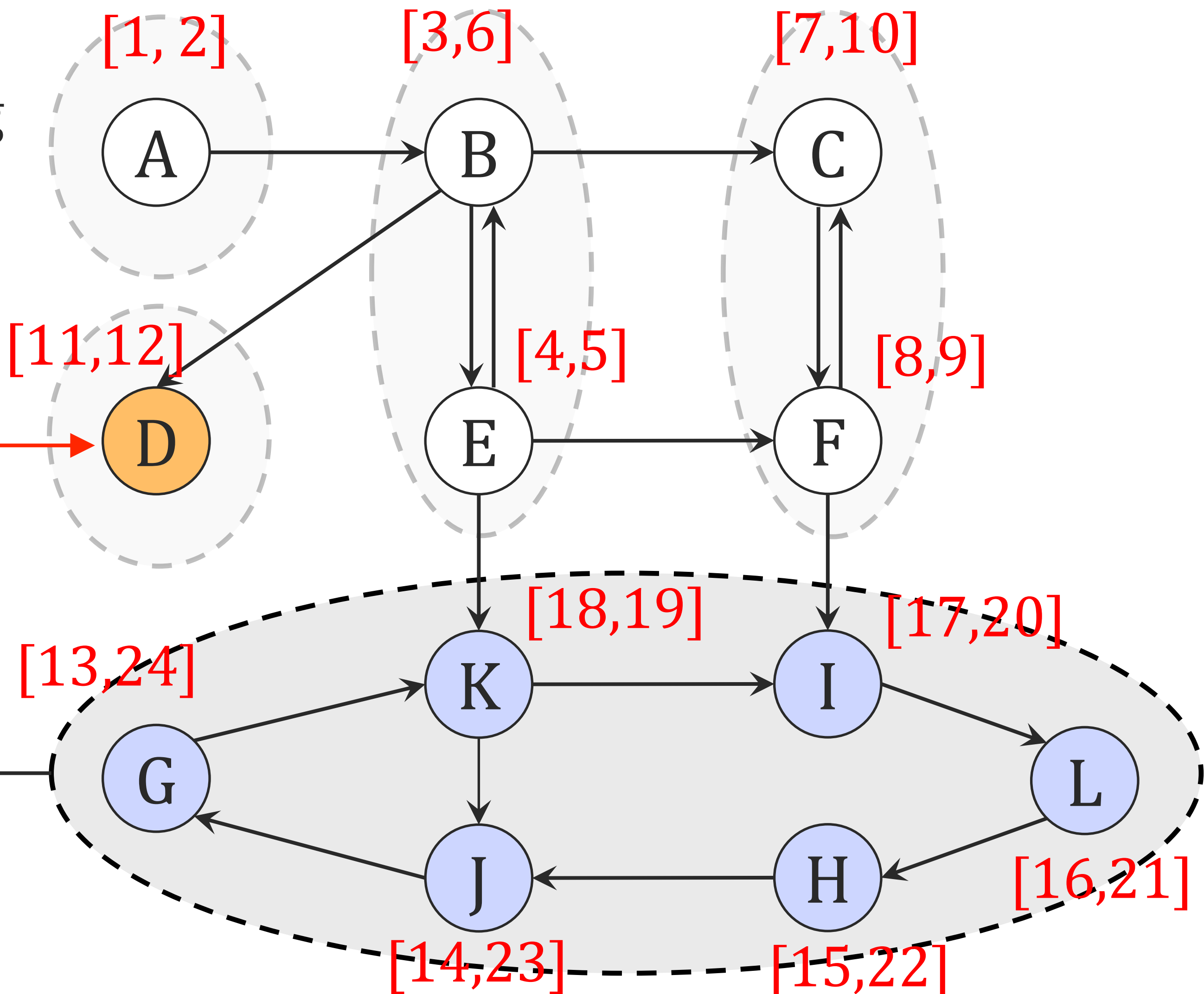
Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

Run DFS and call explore in the decreasing order of post-r numbers.

Highest remaining post-r

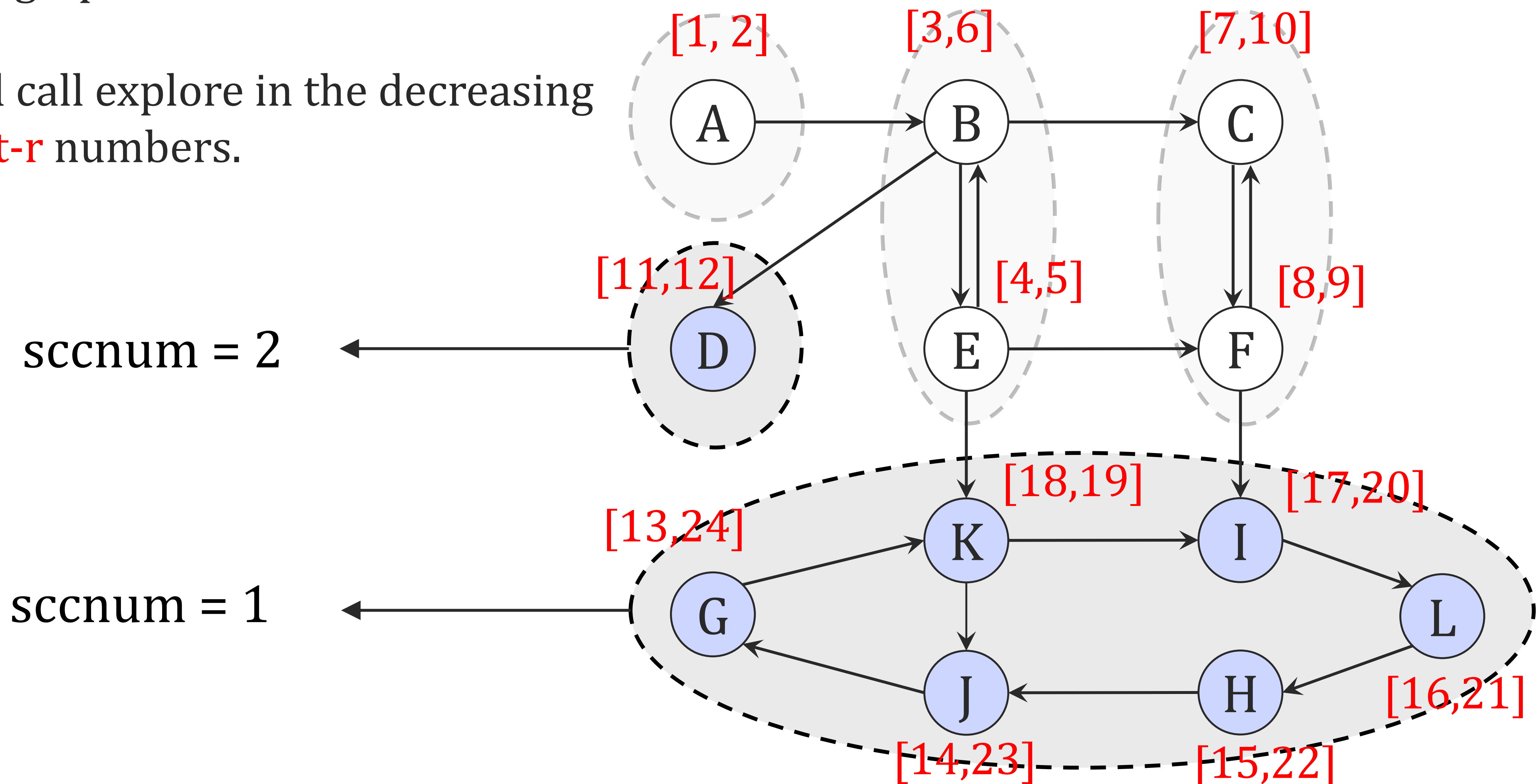
sccnum = 1



Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

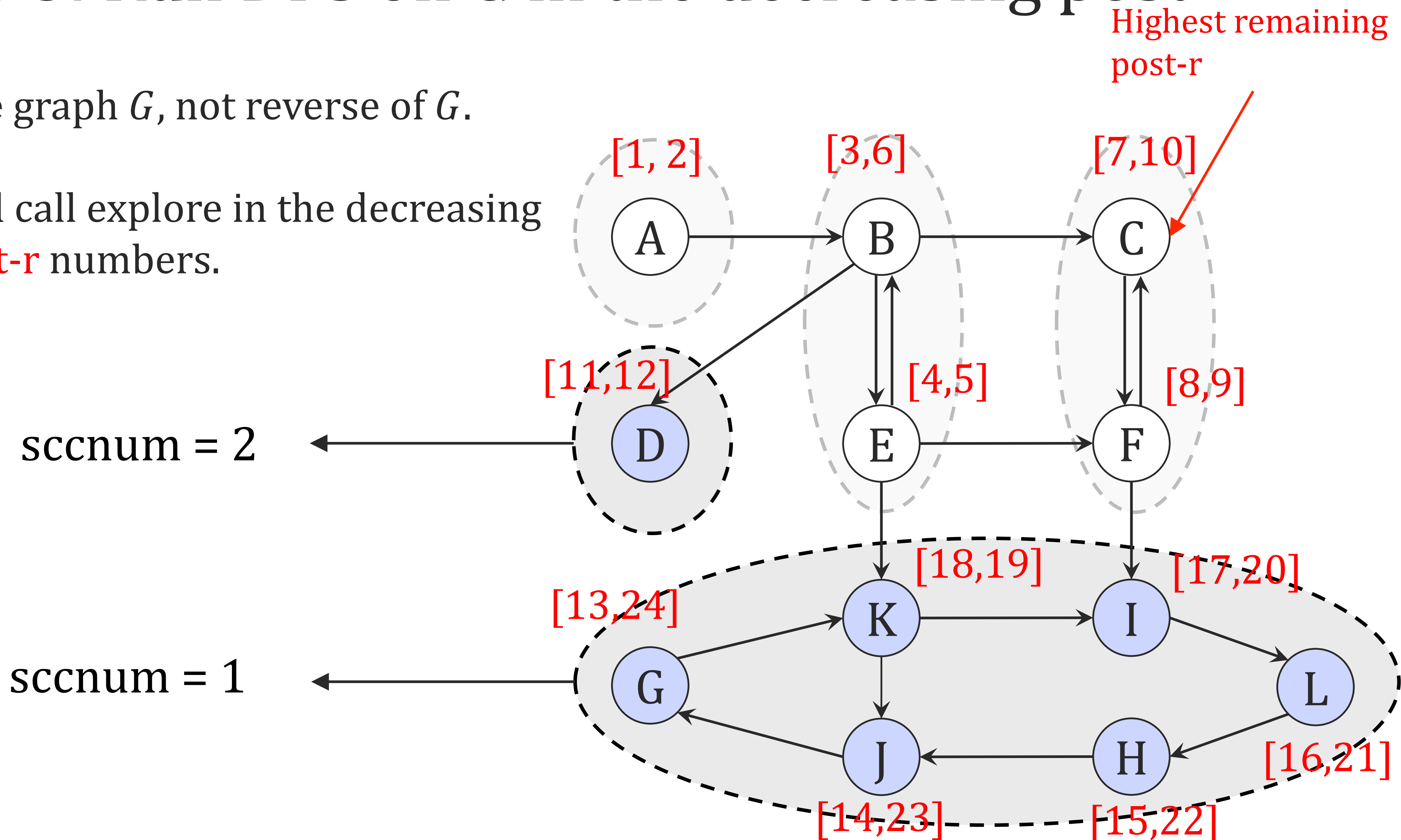
Run DFS and call explore in the decreasing order of post-r numbers.



Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

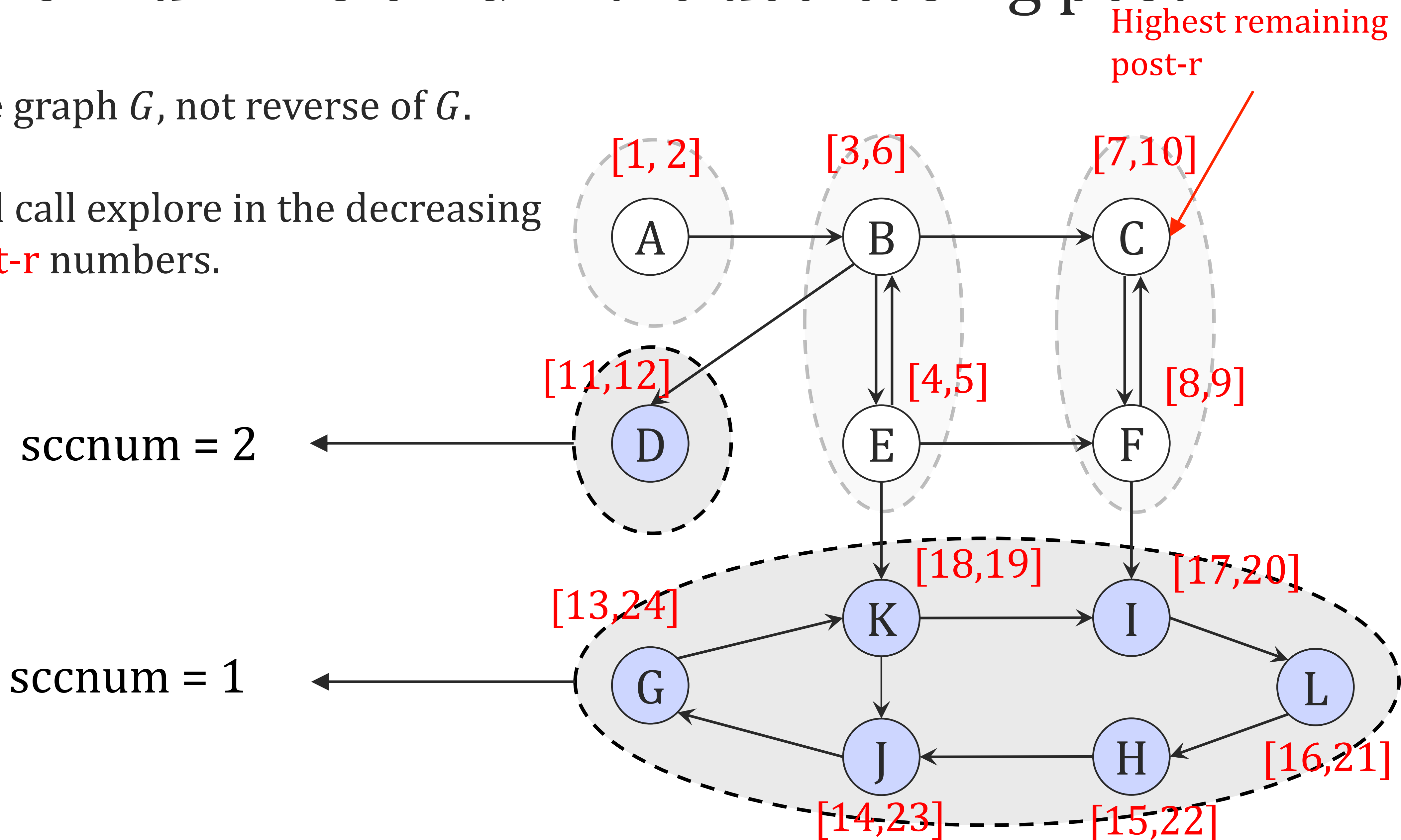
Run DFS and call explore in the decreasing order of post-r numbers.



Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

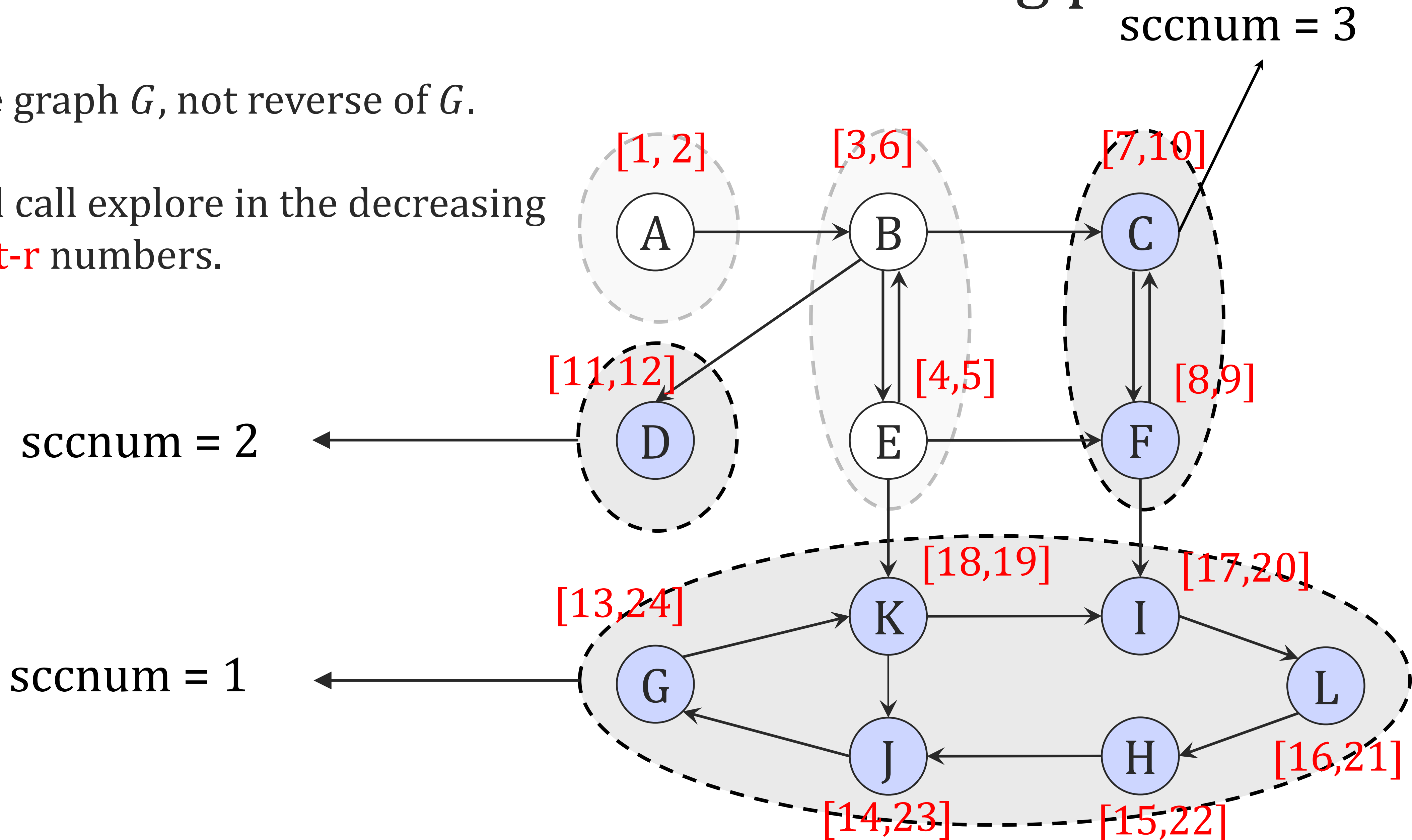
Run DFS and call explore in the decreasing order of post-r numbers.



Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

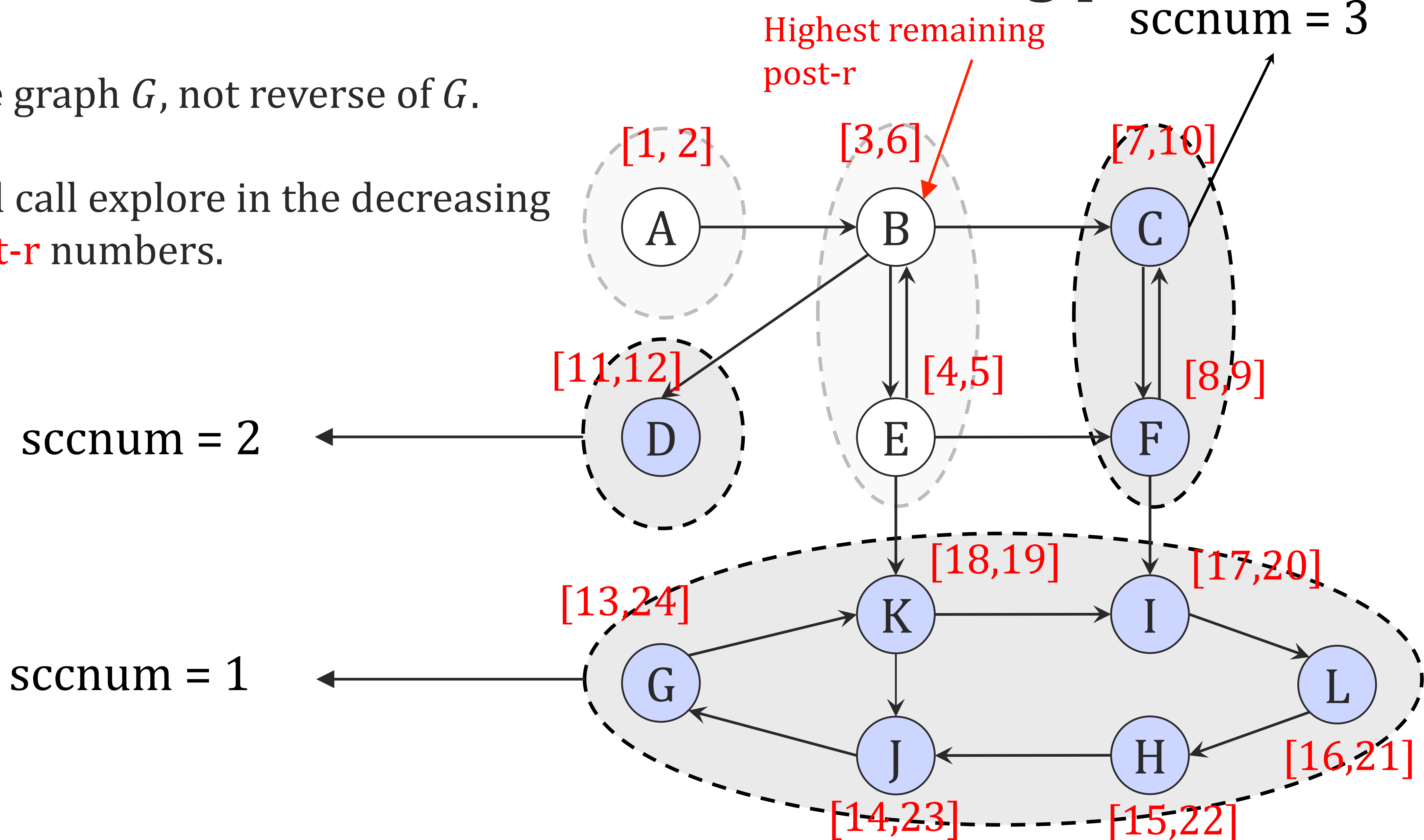
Run DFS and call explore in the decreasing order of post-r numbers.



Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

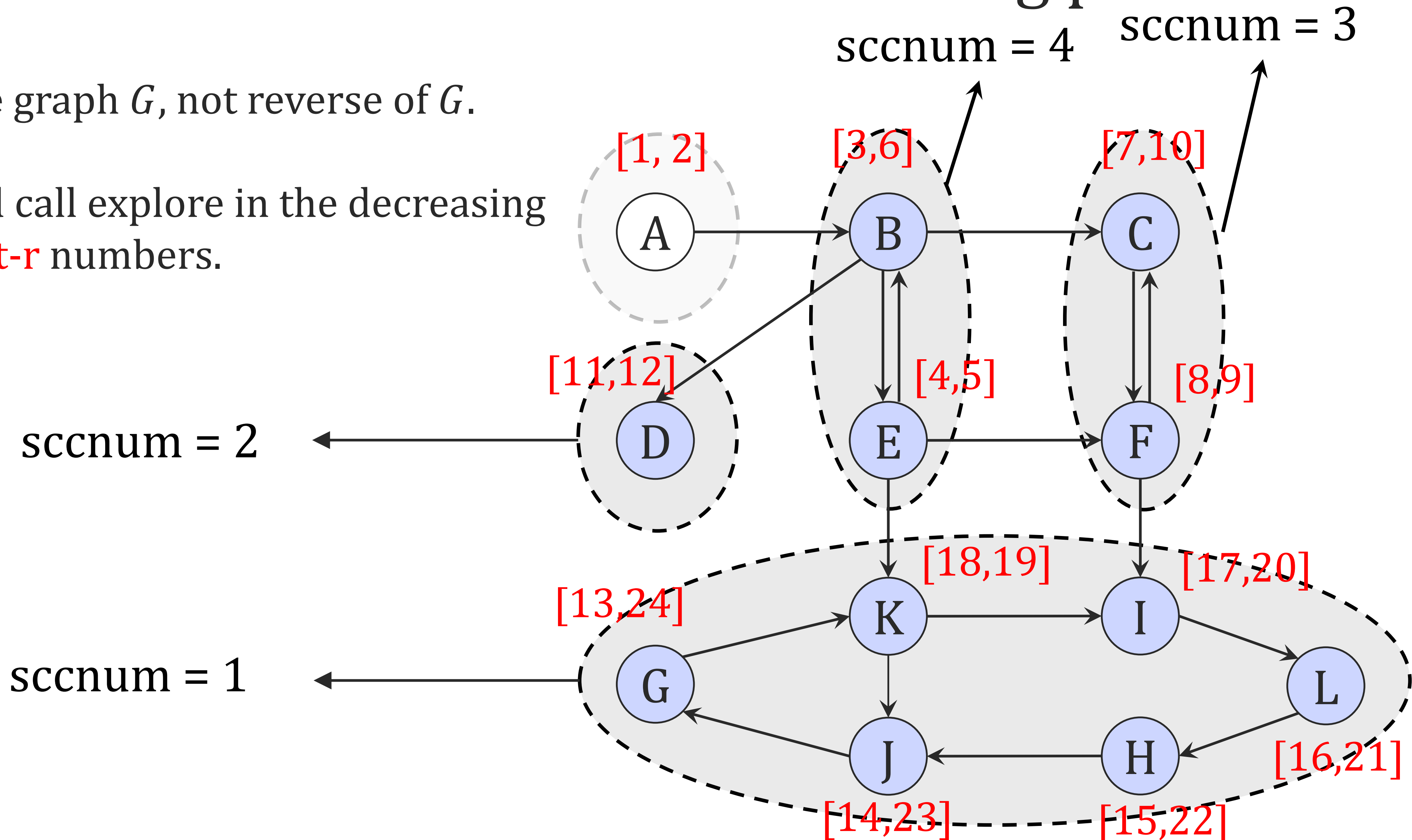
Run DFS and call explore in the decreasing order of post-r numbers.



Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

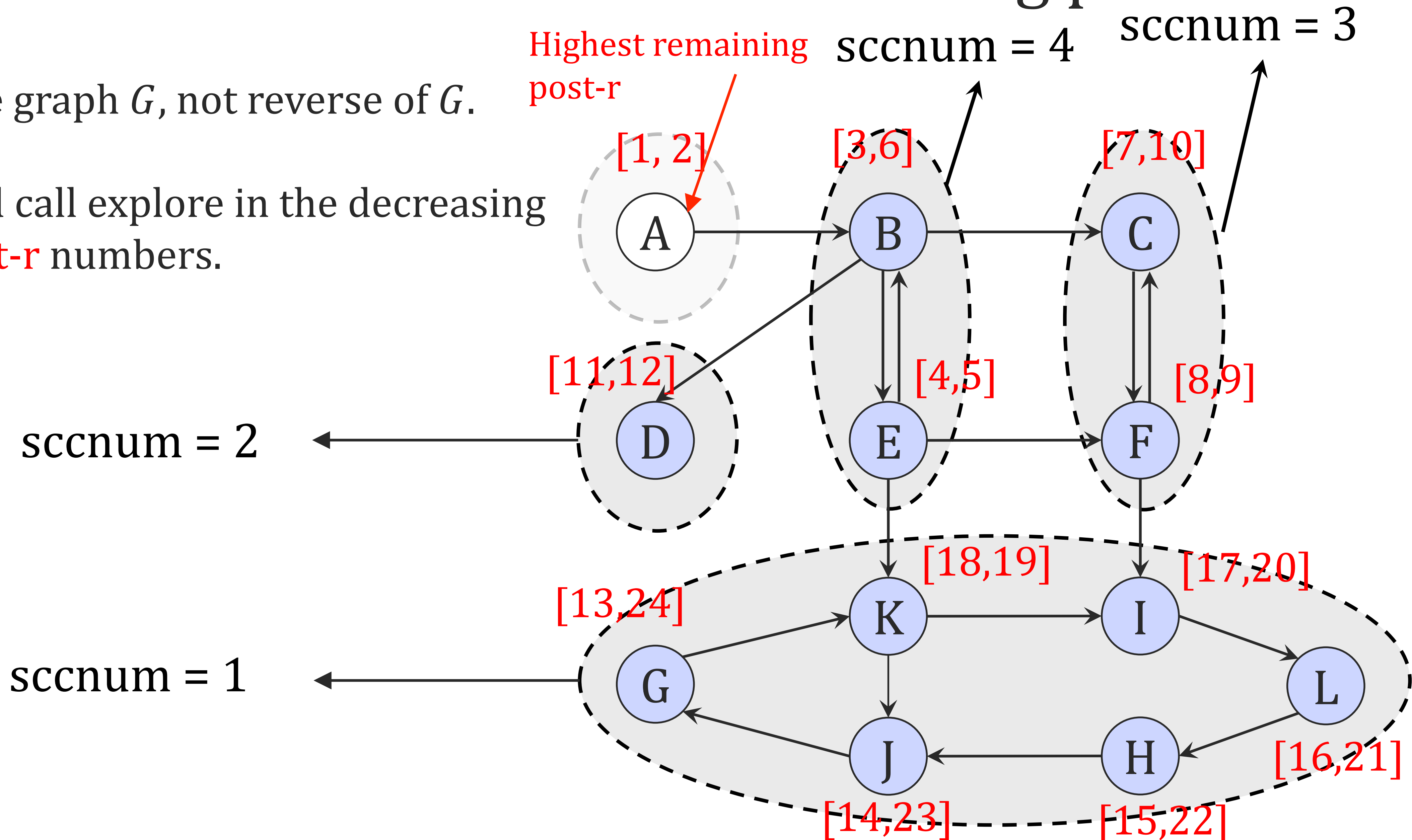
Run DFS and call explore in the decreasing order of post-r numbers.



Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

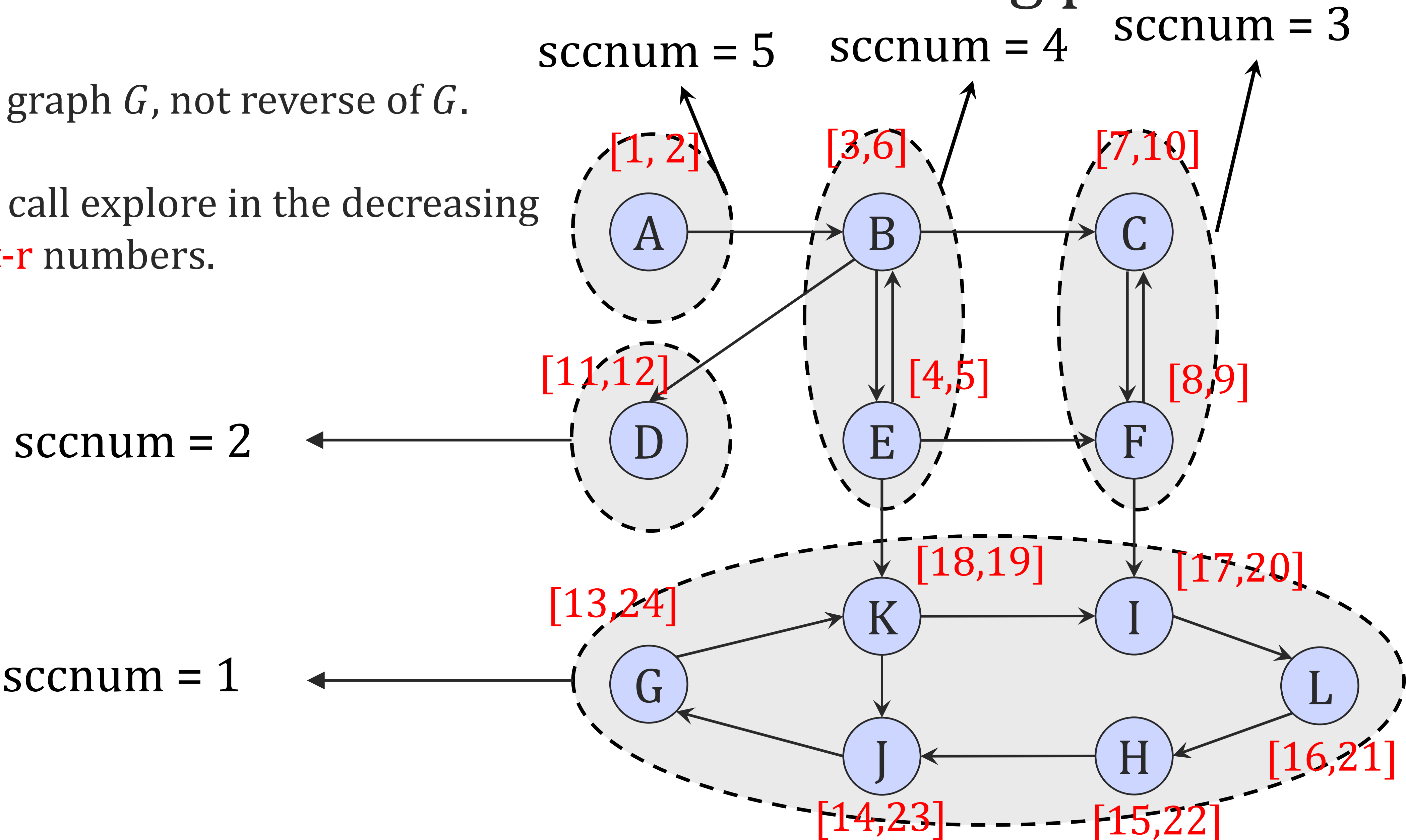
Run DFS and call explore in the decreasing order of post-r numbers.



Alg step 3: Run DFS on G in the decreasing post

Consider the graph G, not reverse of G.

Run DFS and call explore in the decreasing order of post-r numbers.



Wrap up

DFS is awesome!

→ Edge types are important → Topological sort and DAGs
→ Simple book keeping tells us about edge types too.
→ Book keeping helps a lot with other algorithm design problems, like finding SCCs.

Next time

- More with graphs
- Paths