CS 106X, Lecture 22
Graphs; BFS; DFS

reading:

Programming Abstractions in C++, Chapter 18

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Keith Schwarz, Julie Zelenski, Jerry Cain, Eric Roberts, Mehran Sahami, Stuart Reges, Cynthia Lee, Marty Stepp, Ashley Taylor and others.

Plan For Today

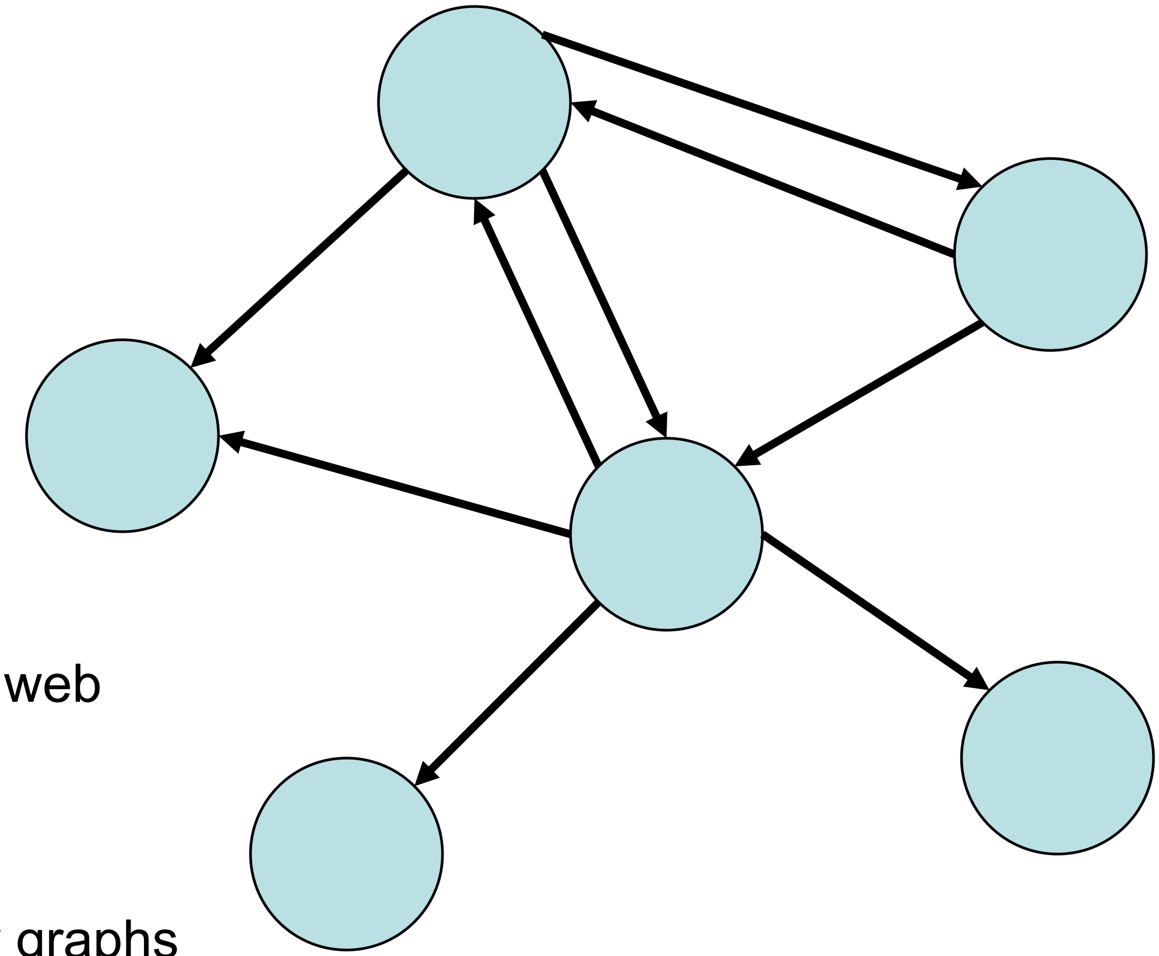
- **Recap**: Graphs
- **Practice**: Twitter Influence
- Depth-First Search (DFS)
- Announcements
- Breadth-First Search (BFS)

Plan For Today

- **Recap**: Graphs
- **Practice**: Twitter Influence
- Depth-First Search (DFS)
- Announcements
- Breadth-First Search (BFS)

Graphs

A graph consists of a set of **nodes** connected by **edges**.



Graphs can model:

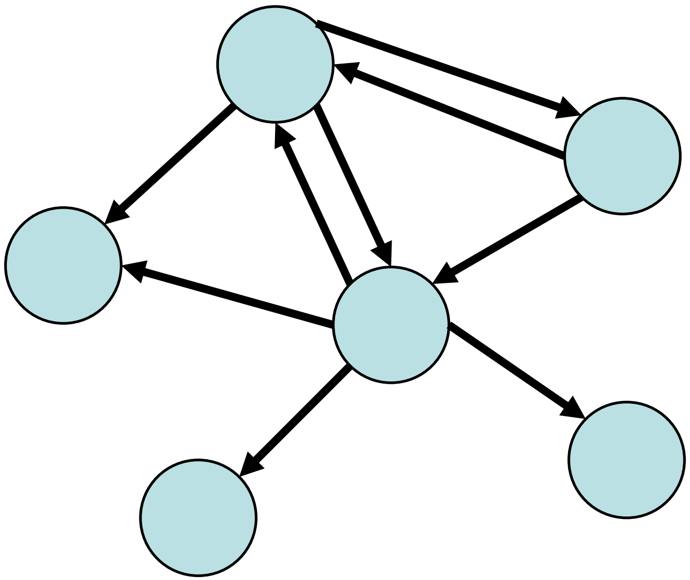
- Sites and links on the web
- Disease outbreaks
- Social networks
- Geographies
- Task and dependency graphs
- and more…

Graphs

A graph consists of a set of **nodes** connected by **edges**.

Nodes: *degree* (# connected edges)
Nodes: *in-degree (directed, # in-edges)*
Nodes: *out-degree (directed, # out-edges)*

Path: sequence of nodes/edges from one node to another
Path: node *x* is reachable from node *y* if a path exists from *y* to *x.*
Path: a *cycle* is a path that starts and ends at the same node
Path: a *loop* is an edge that connects a node to itself

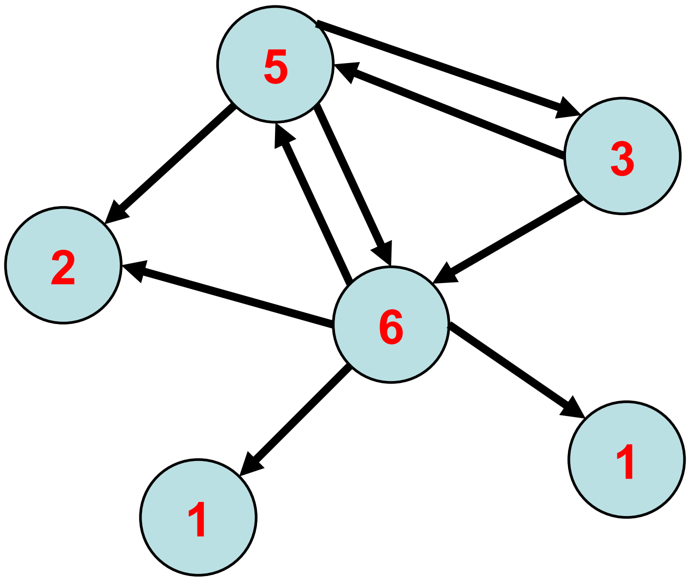


Graphs

A graph consists of a set of **nodes** connected by **edges**.

Nodes: *degree* (# connected edges)
Nodes: *in-degree (directed, # in-edges)*
Nodes: *out-degree (directed, # out-edges)*

Path: sequence of nodes/edges from one node to another
Path: node *x* is reachable from node *y* if a path exists from *y* to *x.*
Path: a *cycle* is a path that starts and ends at the same node
Path: a *loop* is an edge that connects a node to itself

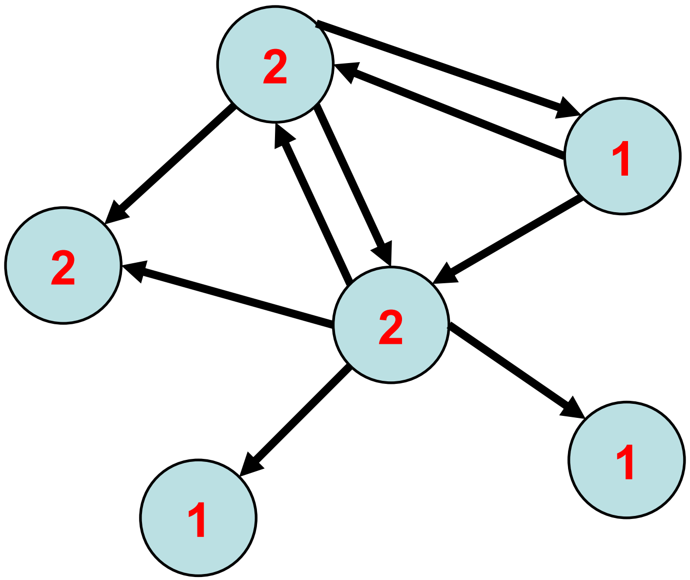


Graphs

A graph consists of a set of **nodes** connected by **edges**.

Nodes: *degree* (# connected edges)
Nodes: *in-degree (directed, # in-edges)*
Nodes: *out-degree (directed, # out-edges)*

Path: sequence of nodes/edges from one node to another
Path: node *x* is reachable from node *y* if a path exists from *y* to *x.*
Path: a *cycle* is a path that starts and ends at the same node
Path: a *loop* is an edge that connects a node to itself

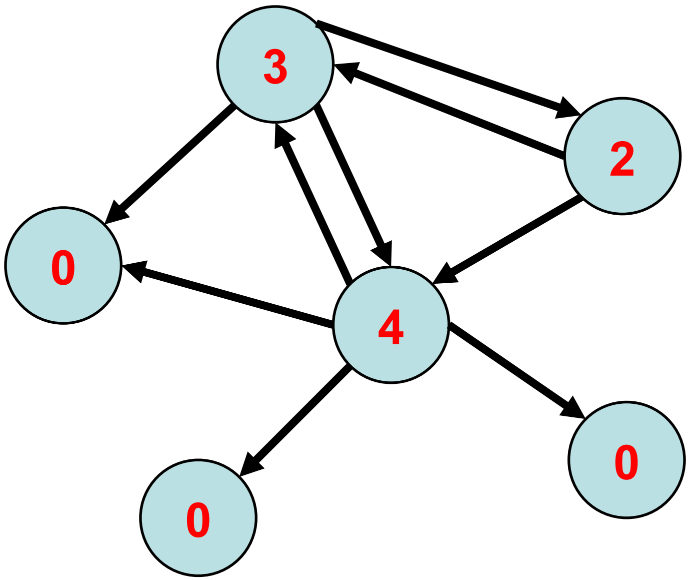


Graphs

A graph consists of a set of **nodes** connected by **edges**.

Nodes: *degree* (# connected edges)
Nodes: *in-degree (directed, # in-edges)*
Nodes: *out-degree (directed, # out-edges)*

Path: sequence of nodes/edges from one node to another
Path: node *x* is reachable from node *y* if a path exists from *y* to *x.*
Path: a *cycle* is a path that starts and ends at the same node
Path: a *loop* is an edge that connects a node to itself

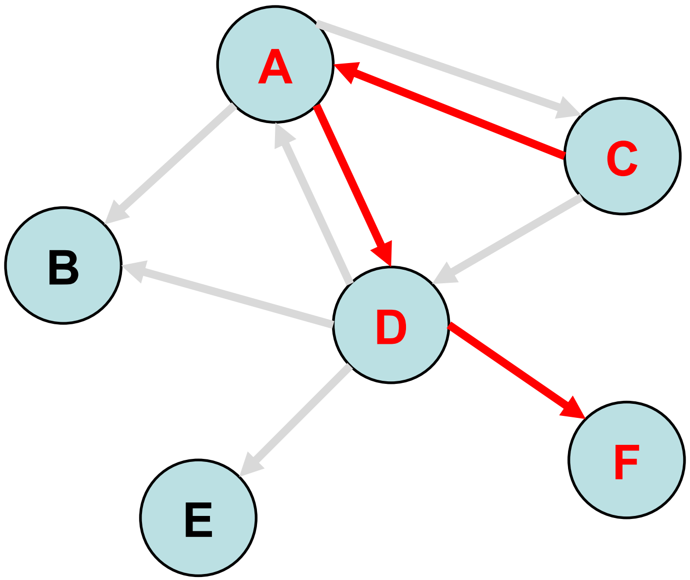


Graphs

A graph consists of a set of **nodes** connected by **edges**.

Nodes: *degree* (# connected edges)
Nodes: *in-degree (directed, # in-edges)*
Nodes: *out-degree (directed, # out-edges)*

Path: sequence of nodes/edges from one node to another
Path: node *x* is reachable from node *y* if a path exists from *y* to *x.*
Path: a *cycle* is a path that starts and ends at the same node
Path: a *loop* is an edge that connects a node to itself

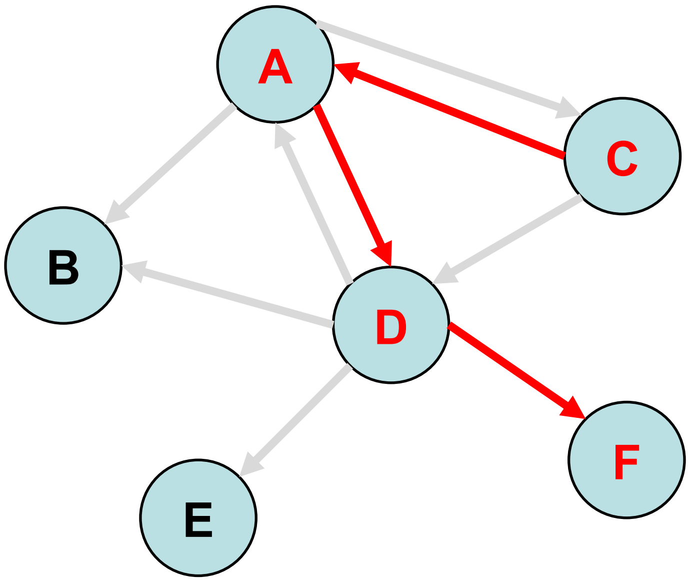


Graphs

A graph consists of a set of **nodes** connected by **edges**.

Nodes: *degree* (# connected edges)
Nodes: *in-degree (directed, # in-edges)*
Nodes: *out-degree (directed, # out-edges)*

Path: sequence of nodes/edges from one node to another
Path: node *x* is reachable from node *y* if a path exists from *y* to *x.*
Path: a *cycle* is a path that starts and ends at the same node
Path: a *loop* is an edge that connects a node to itself

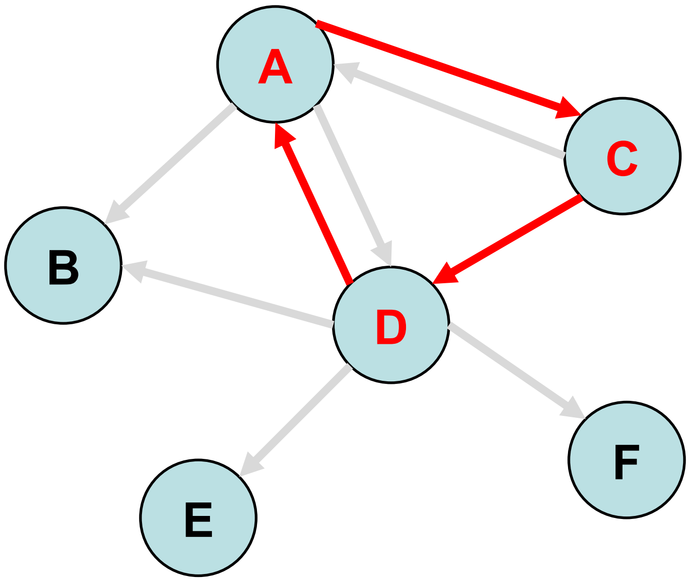


Graphs

A graph consists of a set of **nodes** connected by **edges**.

Nodes: *degree* (# connected edges)
Nodes: *in-degree (directed, # in-edges)*
Nodes: *out-degree (directed, # out-edges)*

Path: sequence of nodes/edges from one node to another
Path: node *x* is reachable from node *y* if a path exists from *y* to *x.*
Path: a *cycle* is a path that starts and ends at the same node
Path: a *loop* is an edge that connects a node to itself

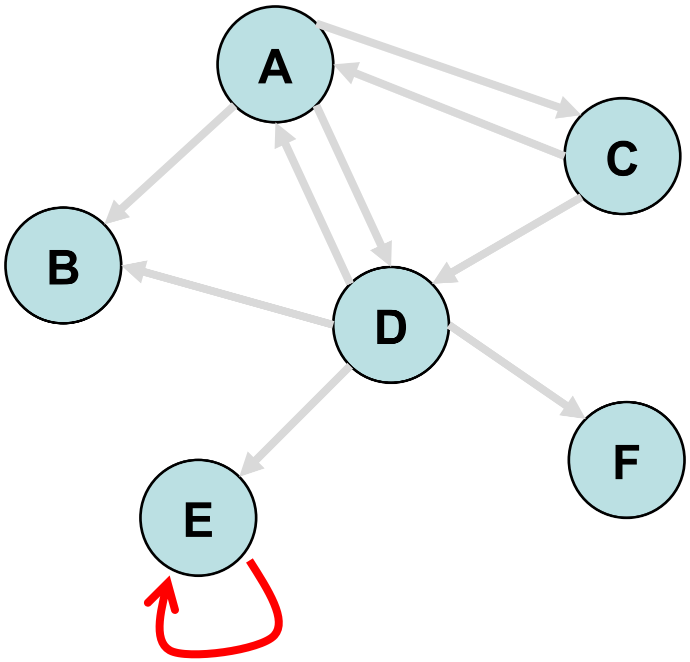


Graphs

A graph consists of a set of **nodes** connected by **edges**.

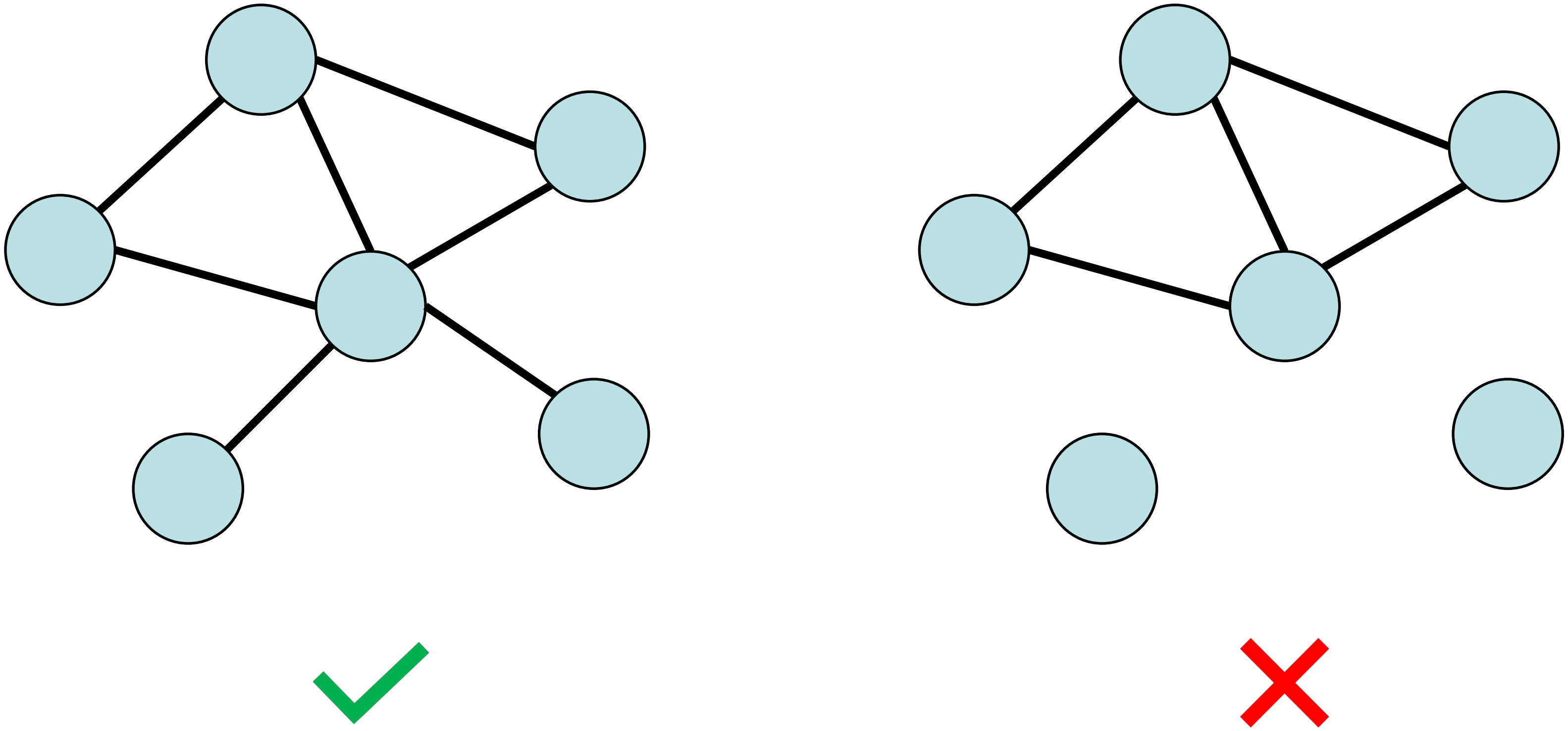
Nodes: *degree* (# connected edges)
Nodes: *in-degree (directed, # in-edges)*
Nodes: *out-degree (directed, # out-edges)*

Path: sequence of nodes/edges from one node to another
Path: node *x* is reachable from node *y* if a path exists from *y* to *x.*
Path: a *cycle* is a path that starts and ends at the same node
Path: a *loop* is an edge that connects a node to itself



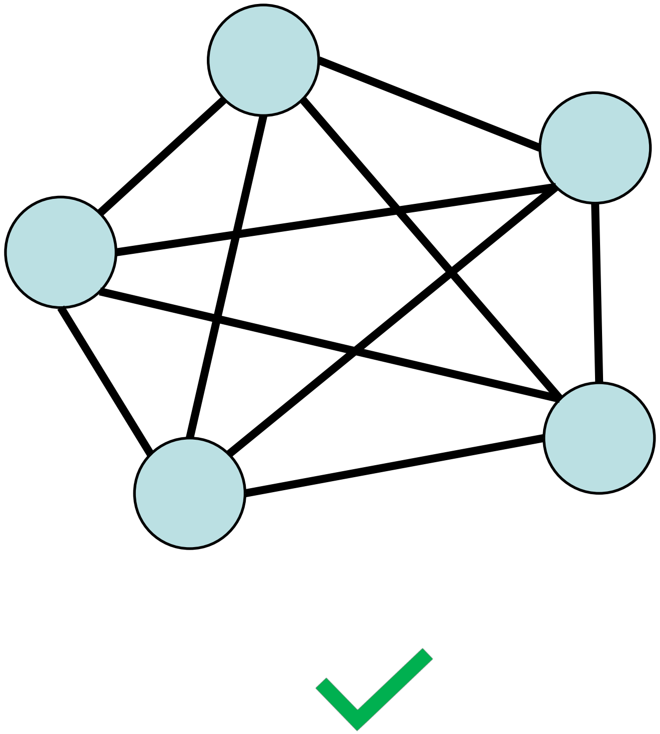
Graph Properties

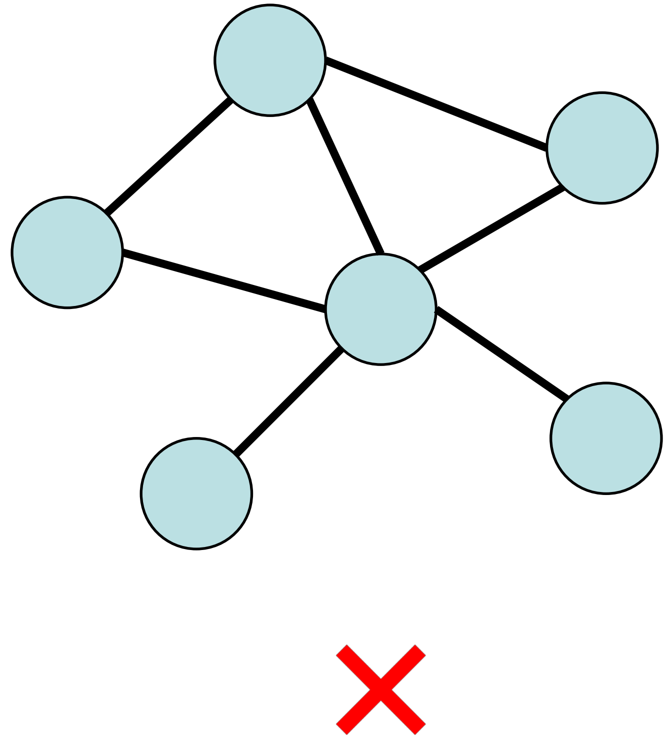
A graph is **connected** if every node is reachable from every other node.



Graph Properties

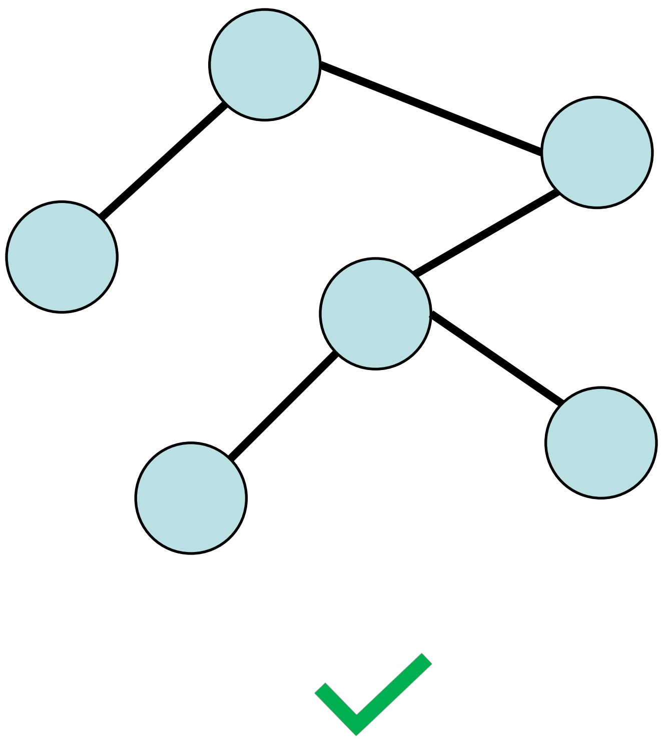
A graph is **complete** if every node has a direct edge to every other node.

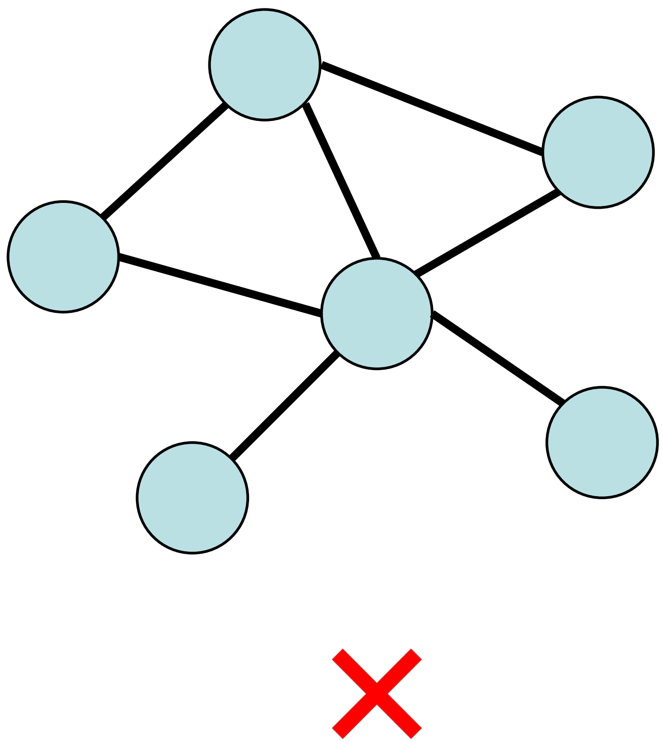




Graph Properties

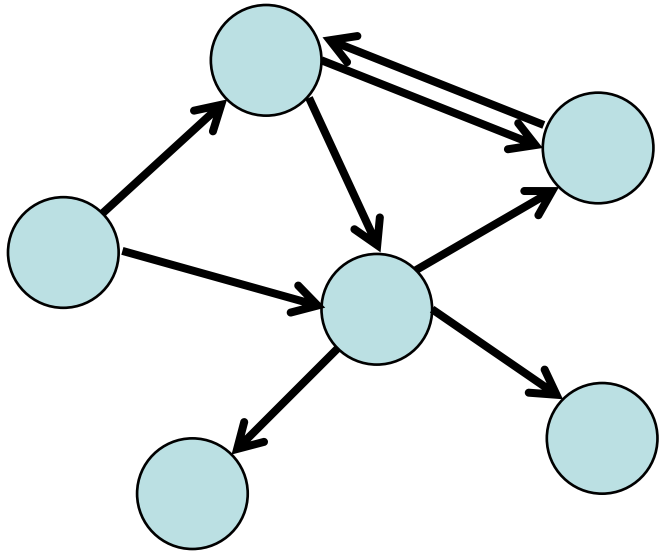
A graph is **acyclic** if it does not contain any cycles.



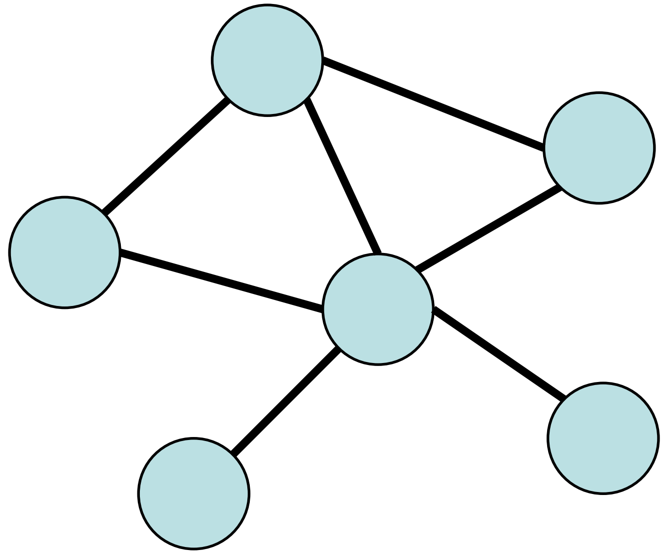


Graph Properties

A graph is **directed** if its edges have direction, or **undirected** if its edges do not have direction (aka are bidirectional).



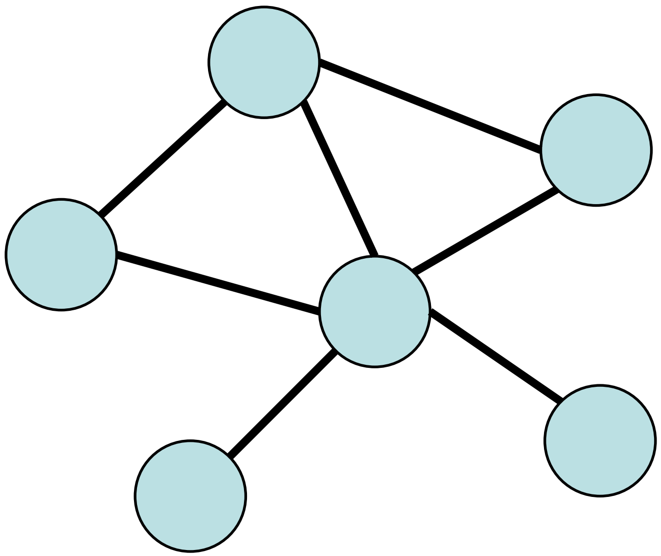
directed



undirected

Graph Properties

- Connected or unconnected
- Acyclic
- Directed or undirected
- Weighted or unweighted
- Complete



Plan For Today

- **Recap**: Graphs
- **Practice**: Twitter Influence
- Depth-First Search (DFS)
- Announcements
- Breadth-First Search (BFS)

Twitter Influence

- Twitter lets a user follow another user to see their posts.
- Following is directional (e.g. I can follow you but you don't have to follow me back ☹)
- Let's define being *influential* as having a high number of *followers-of-followers*.
  - Reasoning: doesn't just matter how many people follow you, but whether the people who follow you reach a large audience.

- Write a function **`mostInfluential`** that reads a file of Twitter relationships and outputs the most influential user.

https://about.twitter.com/en_us/company/brand-resources.html

BasicGraph members

```
#include "basicgraph.h"   // a directed, weighted graph
```

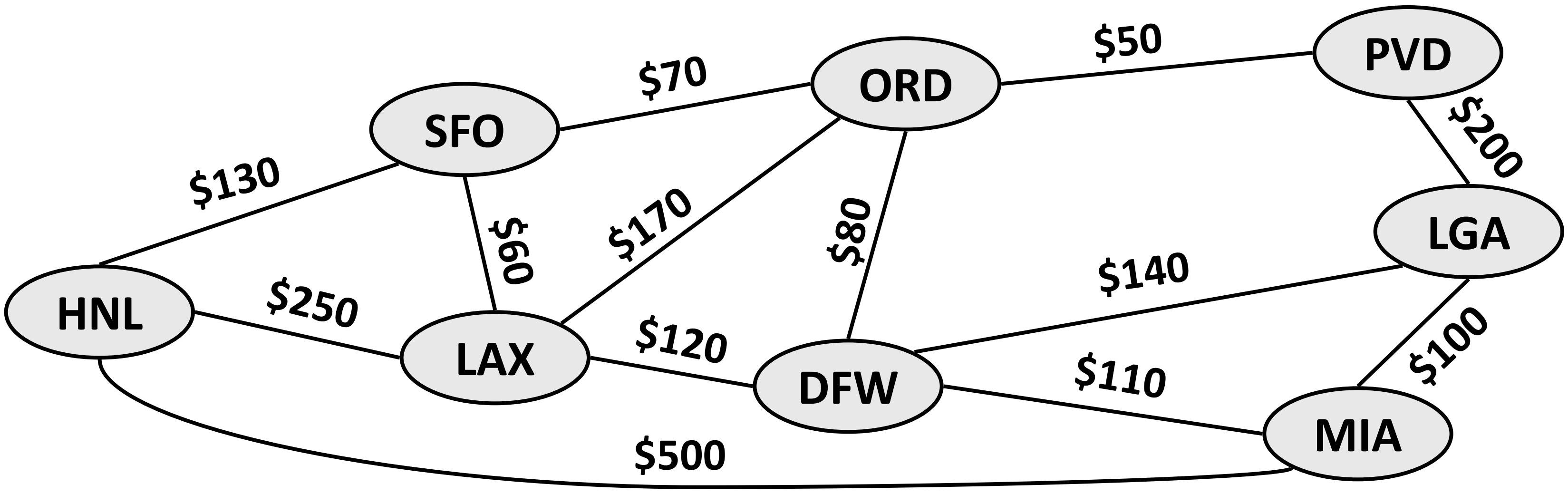
g`.addEdge(`***v1***`, `***v2***`);`	adds an edge between two vertexes
g`.addVertex(`***name***`);`	adds a vertex to the graph
g`.clear();`	removes all vertexes/edges from the graph
g`.getEdgeSet()` ***g***`.getEdgeSet(`***v***`)`	returns all edges, or all edges that start at ***v***, as a `Set` of pointers
g`.getNeighbors(`***v***`)`	returns a set of all vertices that ***v*** has an edge to
g`.getVertex(`***name***`)`	returns pointer to vertex with the given name
g`.getVertexSet()`	returns a set of all vertexes
g`.isNeighbor(`***v1***`, `***v2***`)`	returns `true` if there is an edge from vertex ***v1*** to ***v2***
g`.isEmpty()`	returns `true` if queue contains no vertexes/edges
g`.removeEdge(`***v1***`, `***v2***`);`	removes an edge from the graph
g`.removeVertex(`***name***`);`	removes a vertex from the graph
g`.size()`	returns the number of vertexes in the graph
g`.toString()`	returns a string such as `"{a, b, c, a -> b}"`

Plan For Today

- **Recap**: Graphs
- **Practice**: Twitter Influence
- Depth-First Search (DFS)
- Announcements
- Breadth-First Search (BFS)

Searching for paths

- Searching for a path from one vertex to another:
  - Sometimes, we just want *any* path (or want to know there *is* a path).
  - Sometimes, we want to minimize path *length* (# of edges).
  - Sometimes, we want to minimize path *cost* (sum of edge weights).



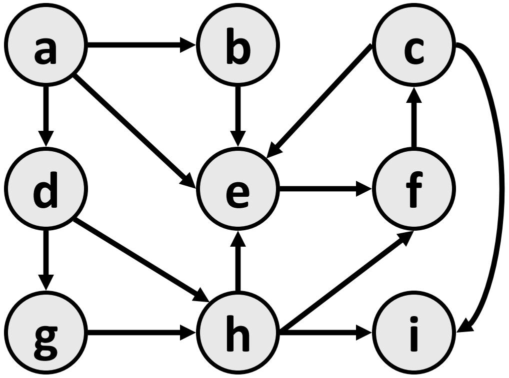
Finding Paths

- Easiest way: Depth-First Search (DFS)
  - Recursive backtracking!
- Finds a path between two nodes if it exists
  - Or can find all the nodes **reachable** from a node
    - Where can I travel to starting in San Francisco?
    - If all my friends (and their friends, and so on) share my post, how many will eventually see it?

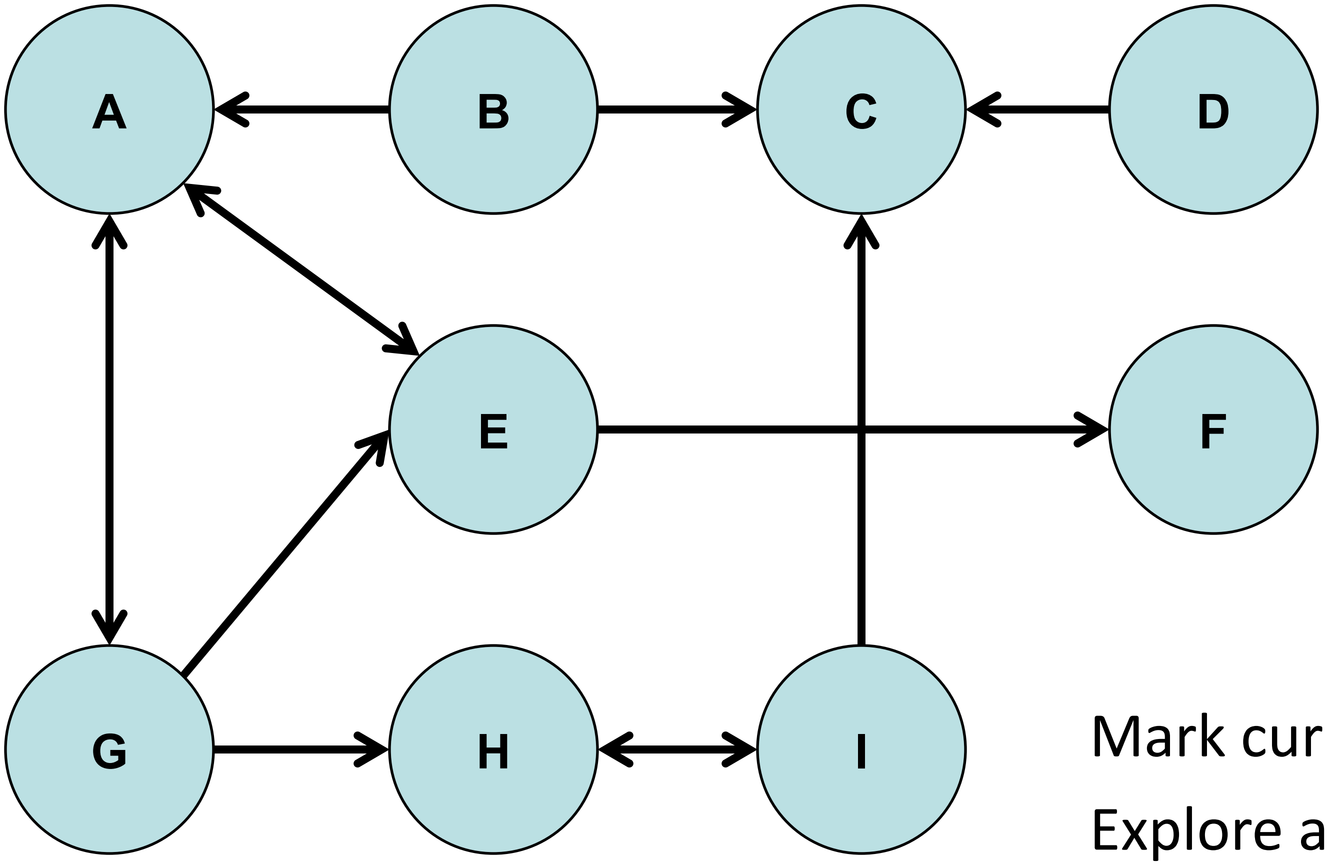
Depth-first search (18.4)

- **depth-first search** (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking.
  - Often implemented recursively.
  - Many graph algorithms involve *visiting* or *marking* vertices.

- DFS from *a* to *h* (assuming A-Z order) visits:
  - **a**
    - **b**
      - **e**
        - **f**
          - **c**
            - **i**
    - **d**
      - **g**
        - **h**
  - path found: **{a, d, g, h}**



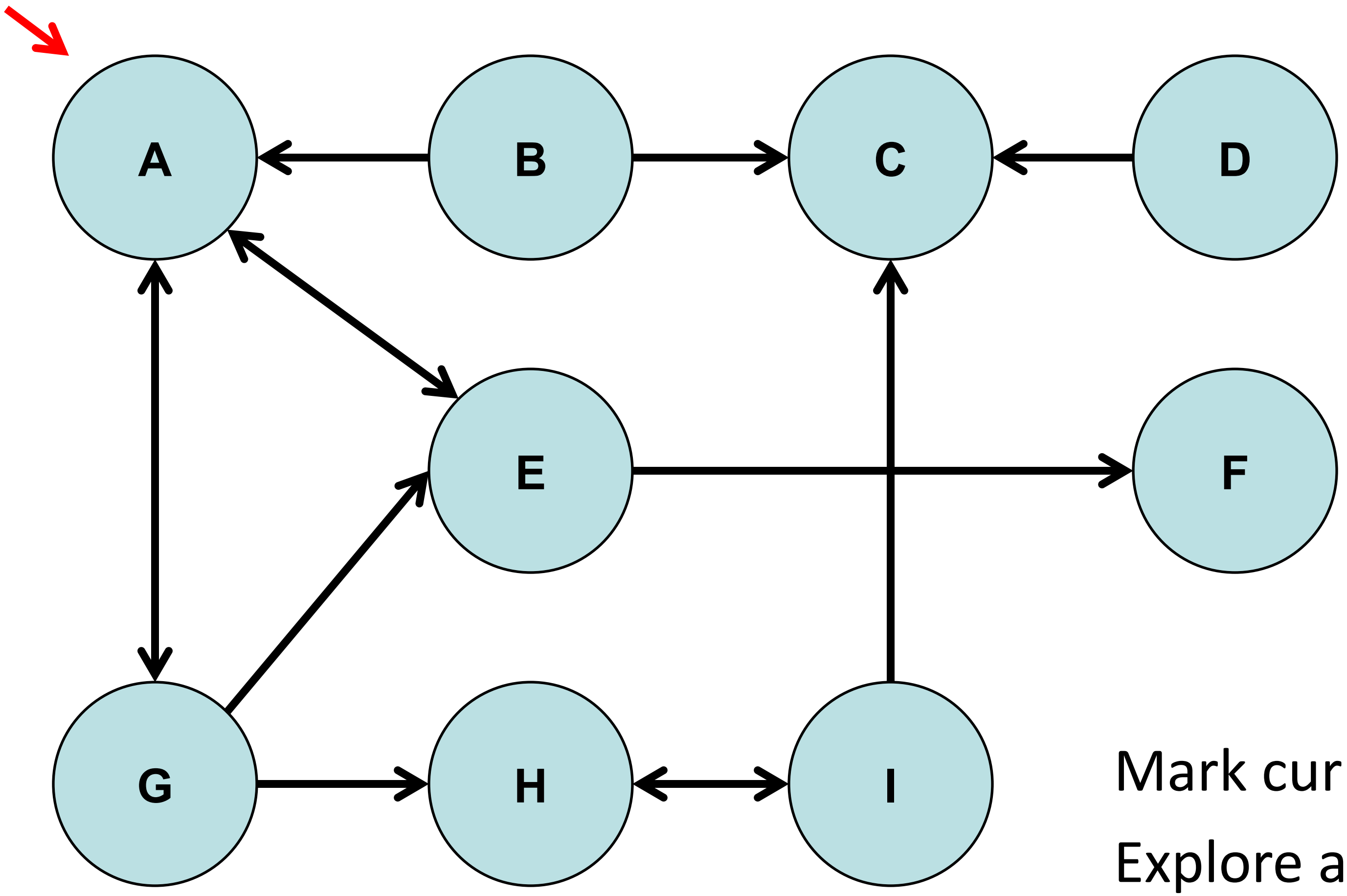
DFS



Mark current as visited

Explore all the unvisited nodes from this node

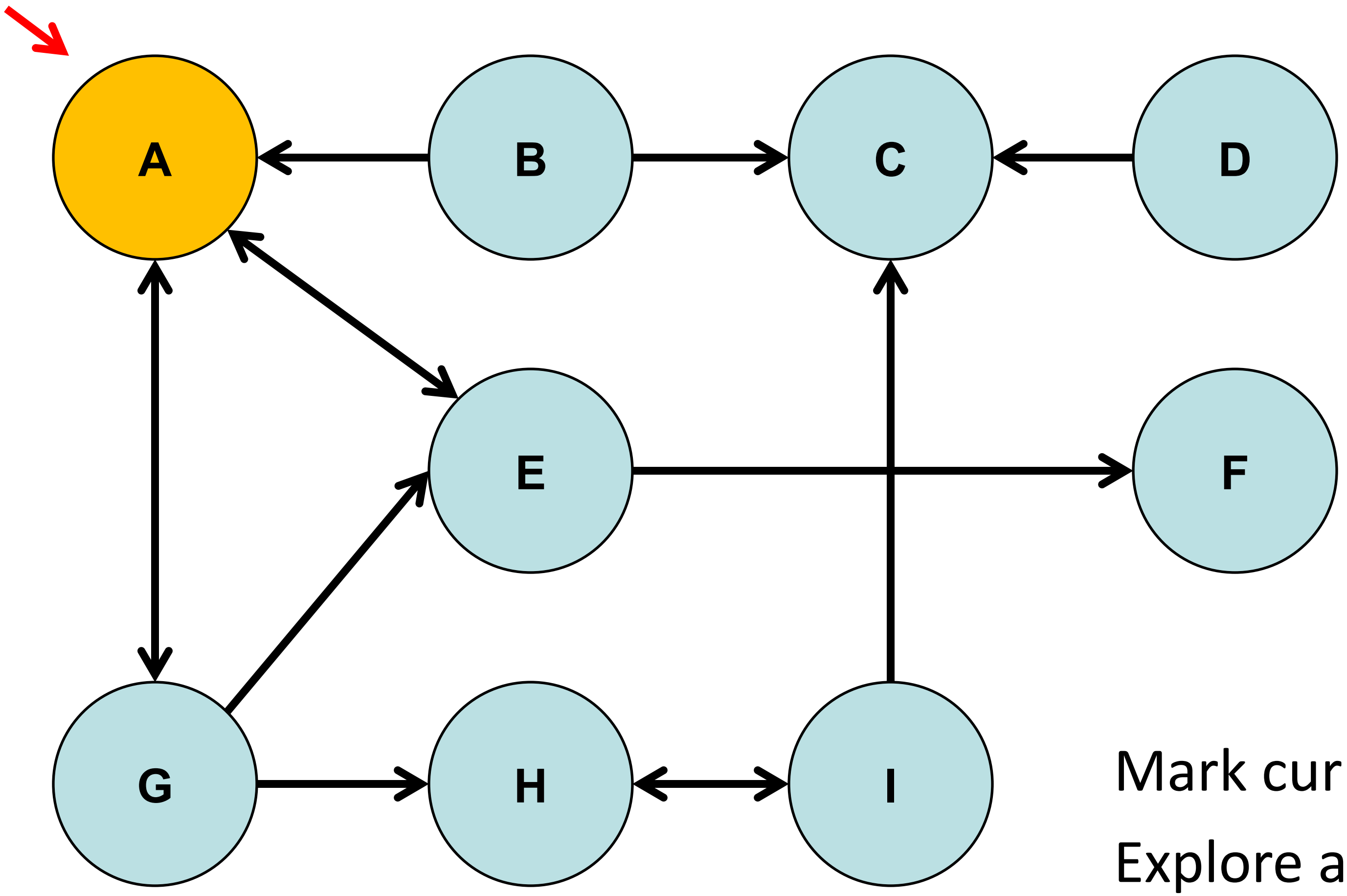
DFS



Mark current as visited

Explore all the unvisited nodes from this node

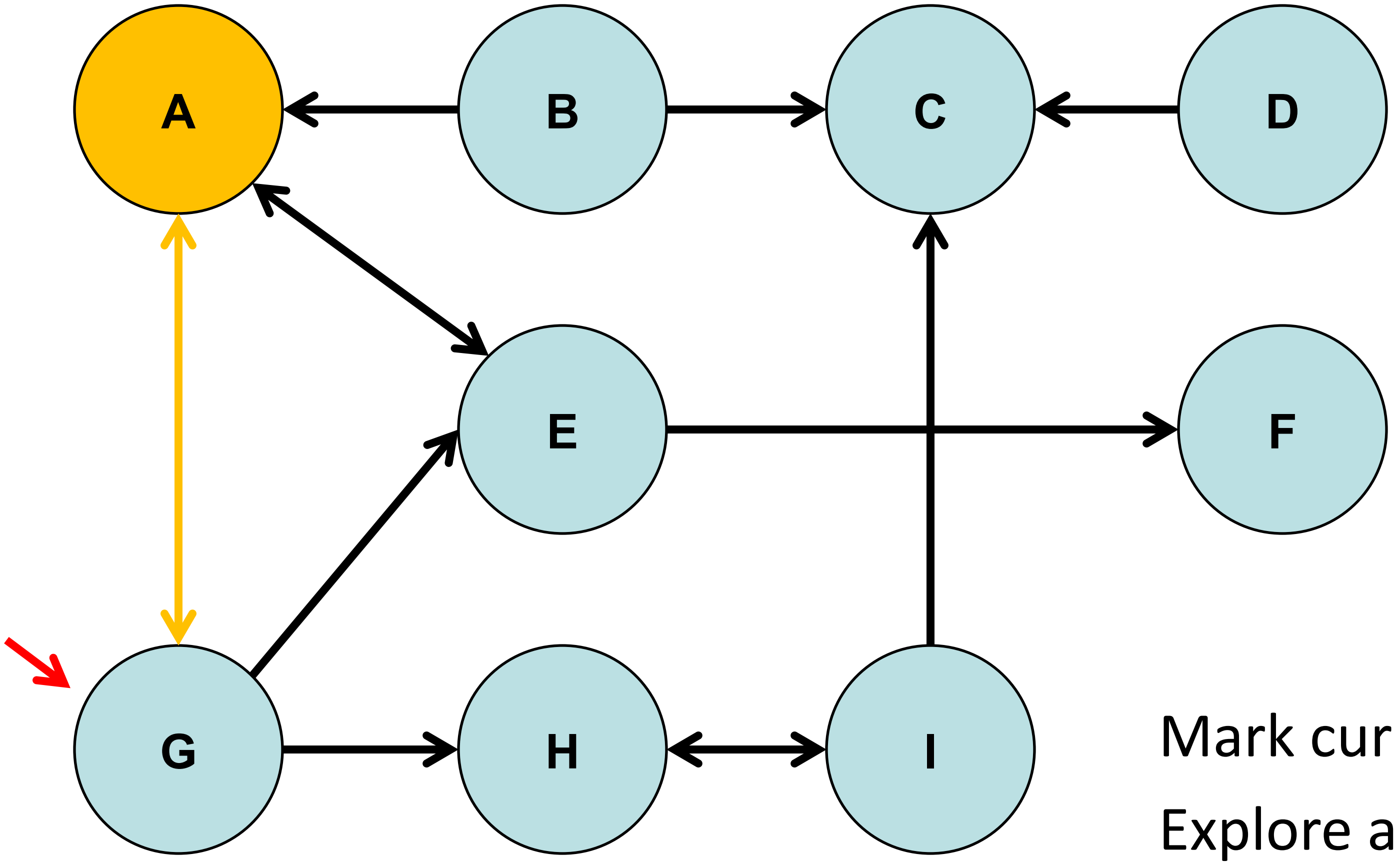
DFS



Mark current as visited

Explore all the unvisited nodes from this node

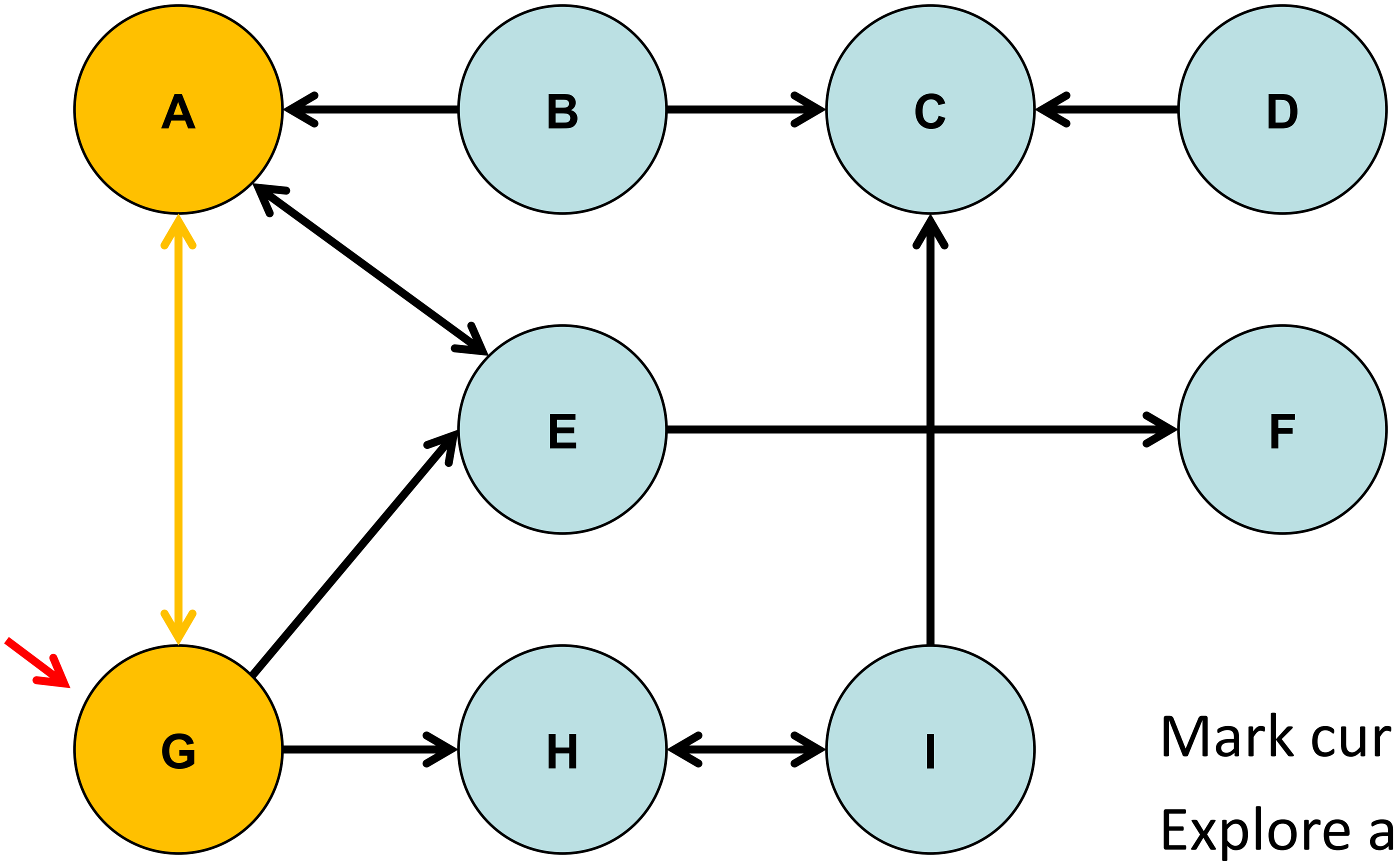
DFS



Mark current as visited

Explore all the unvisited nodes from this node

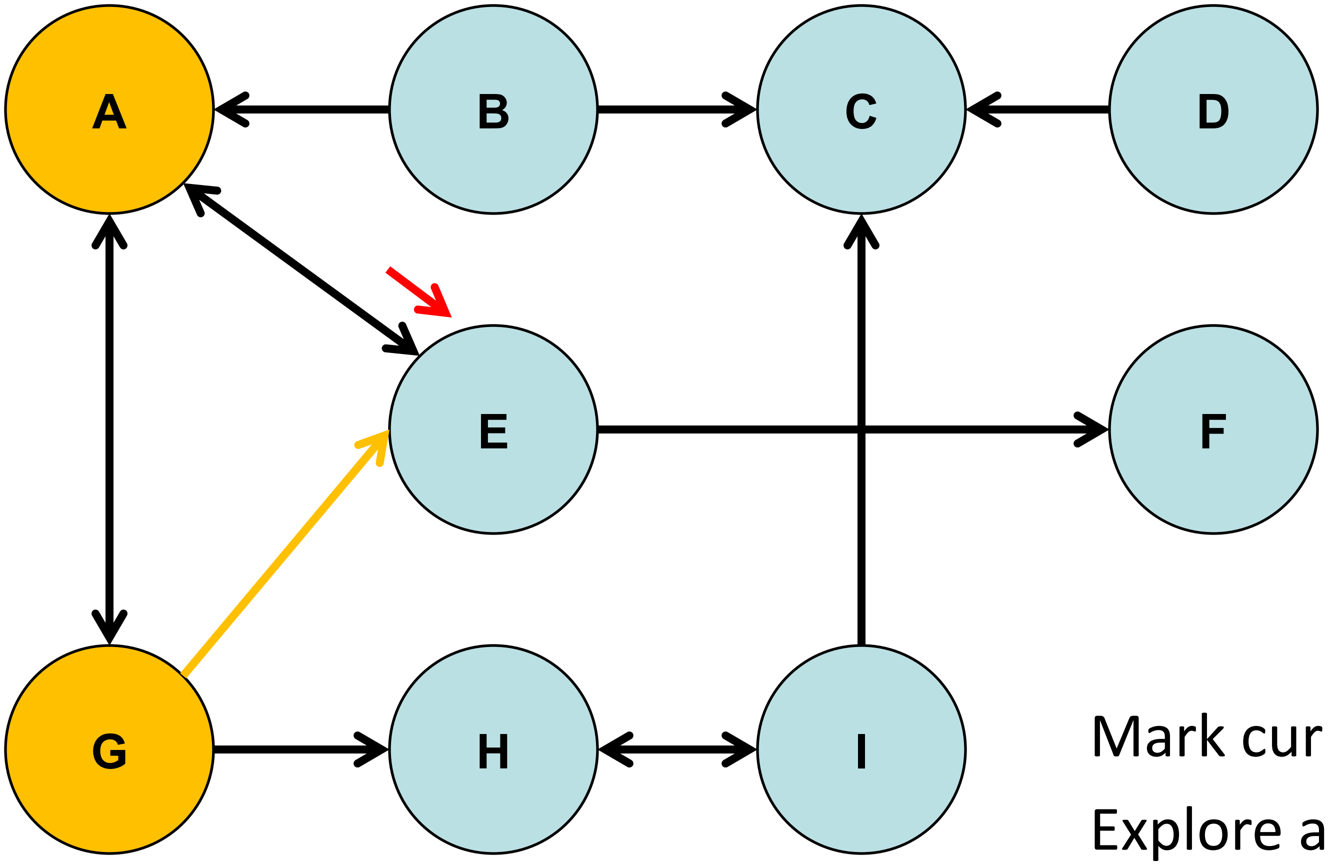
DFS



Mark current as visited

Explore all the unvisited nodes from this node

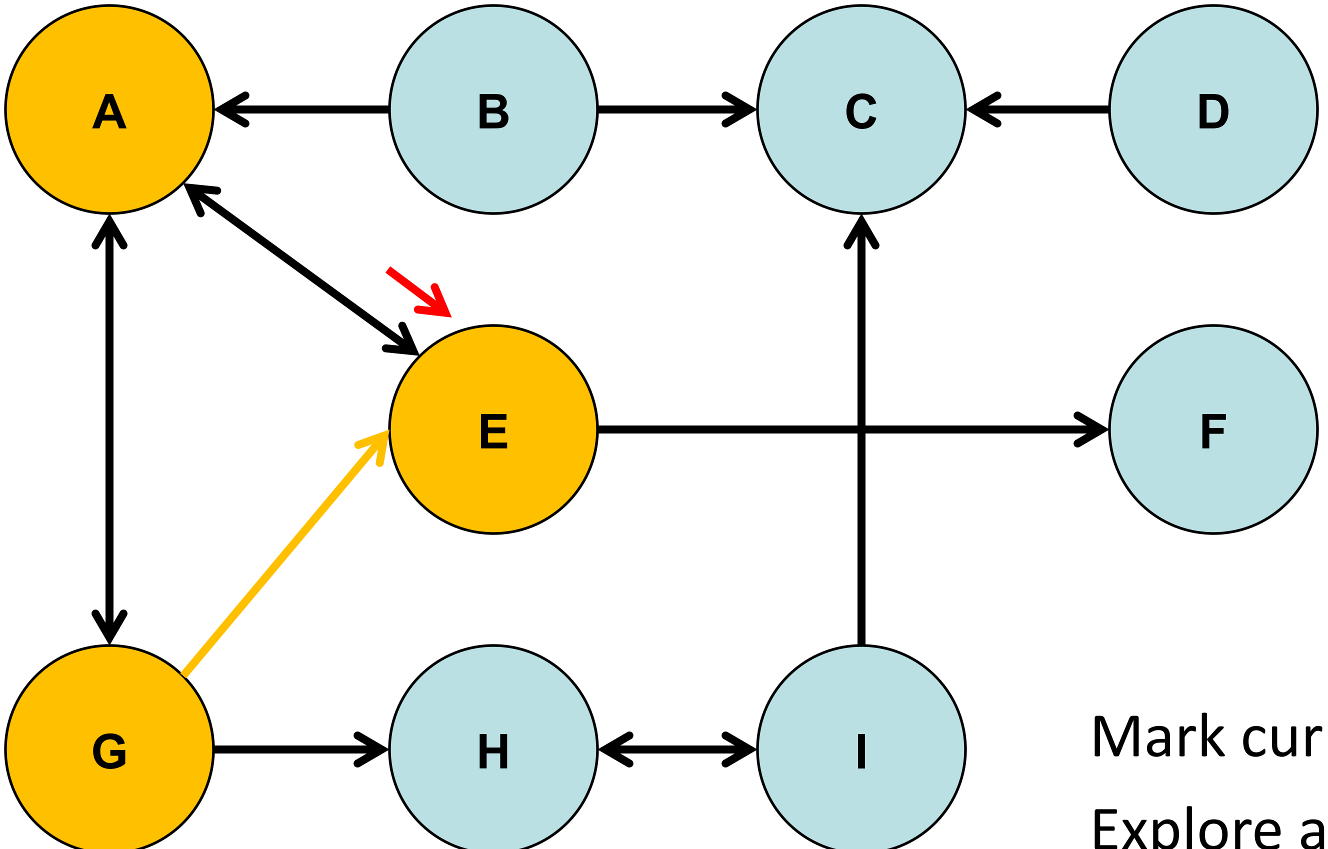
DFS



Mark current as visited

Explore all the unvisited nodes from this node

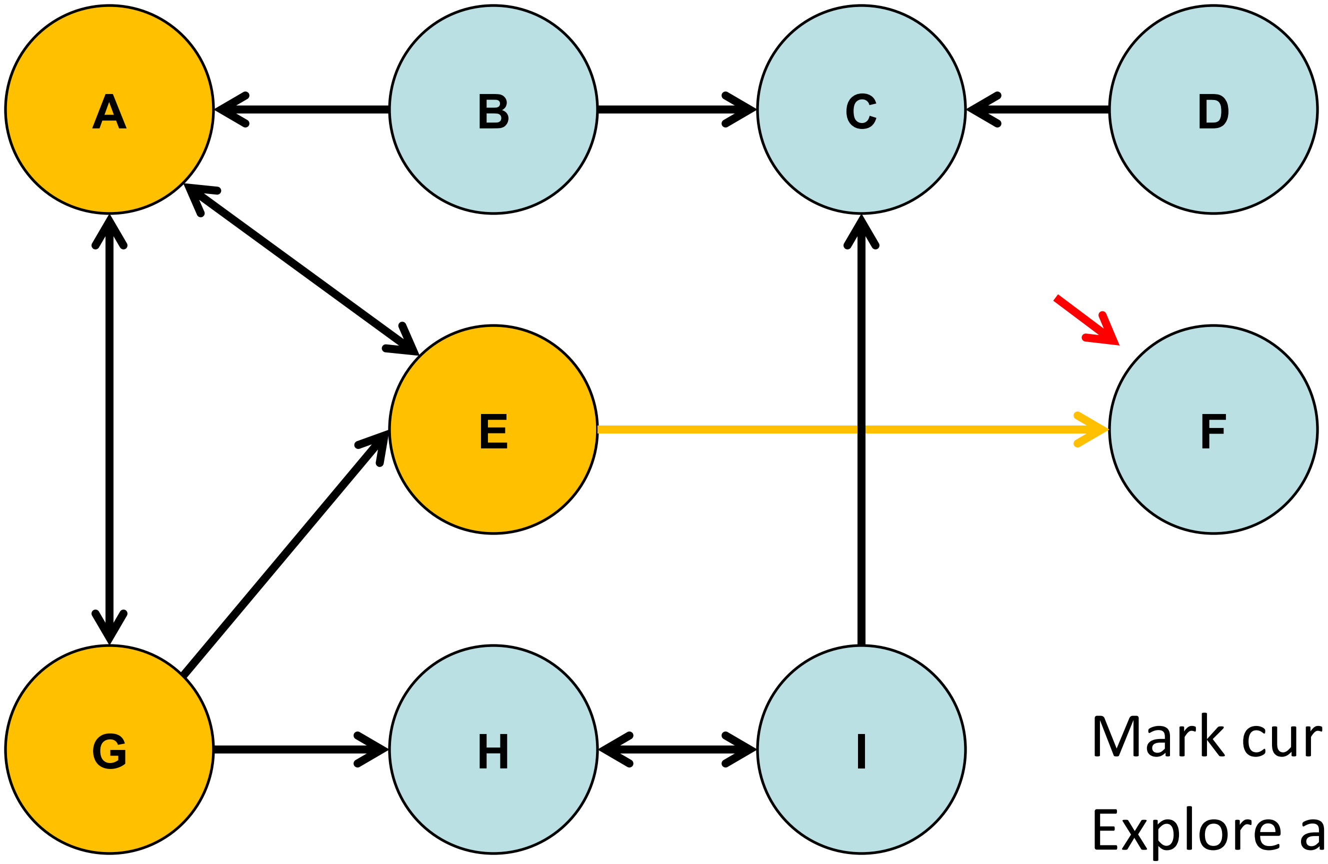
DFS



Mark current as visited

Explore all the unvisited nodes from this node

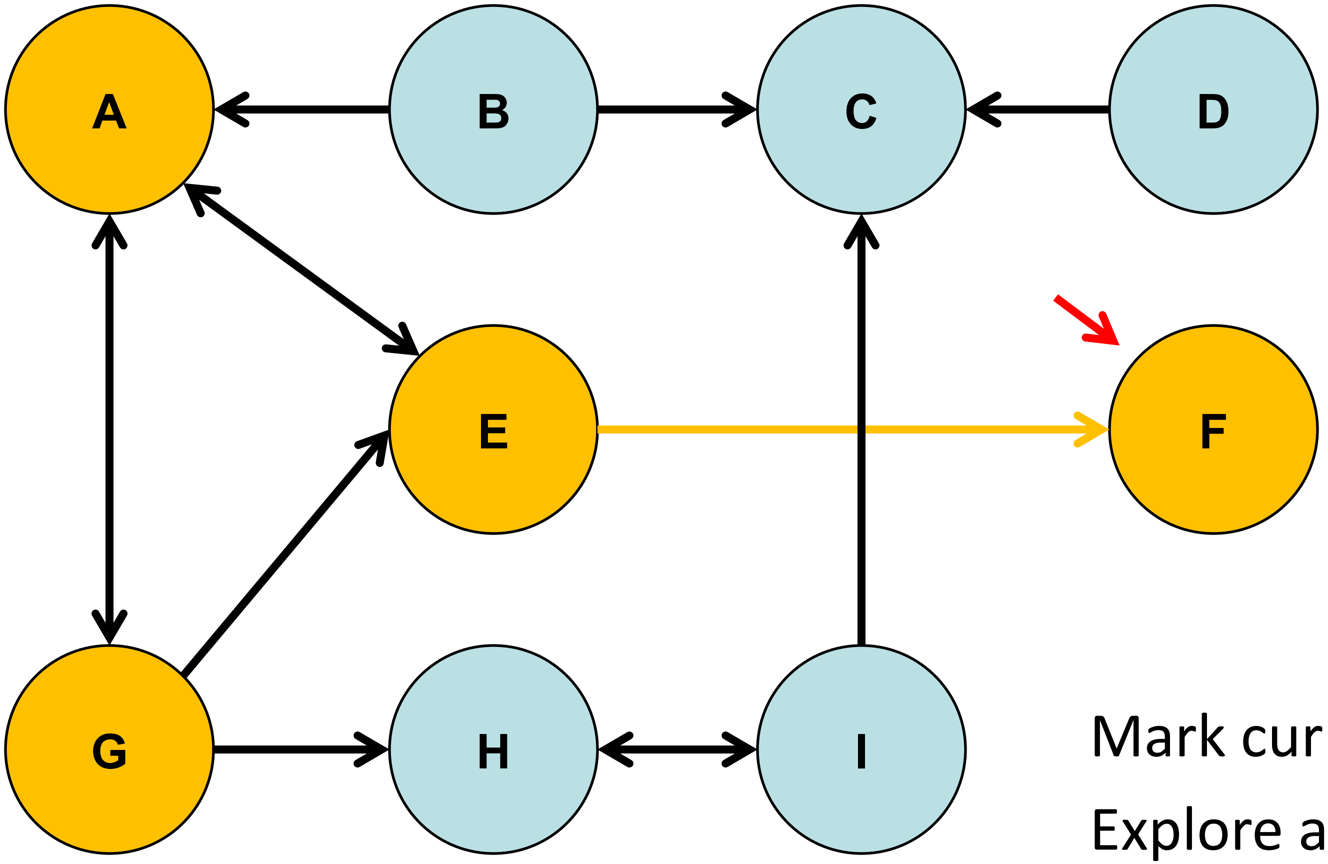
DFS



Mark current as visited

Explore all the unvisited nodes from this node

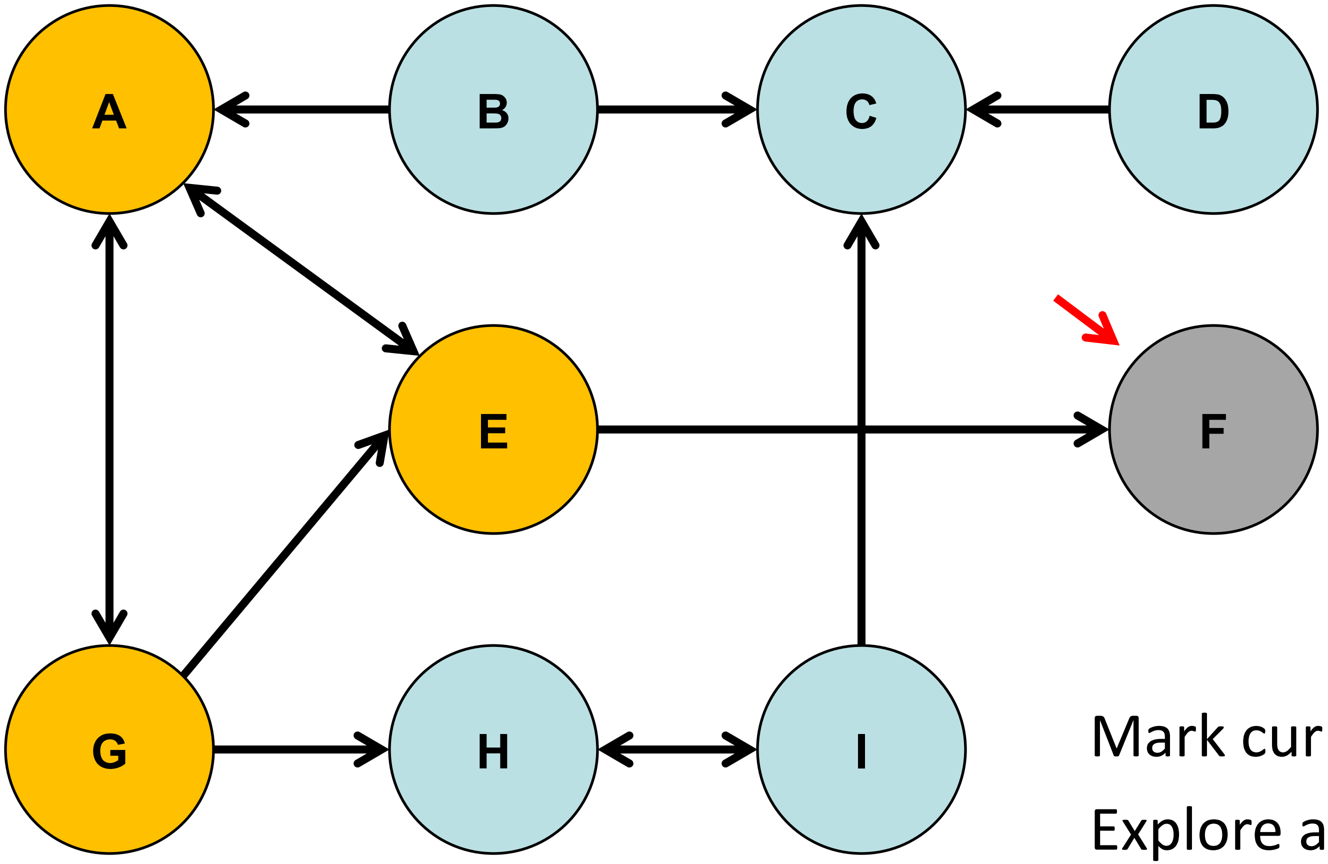
DFS



Mark current as visited

Explore all the unvisited nodes from this node

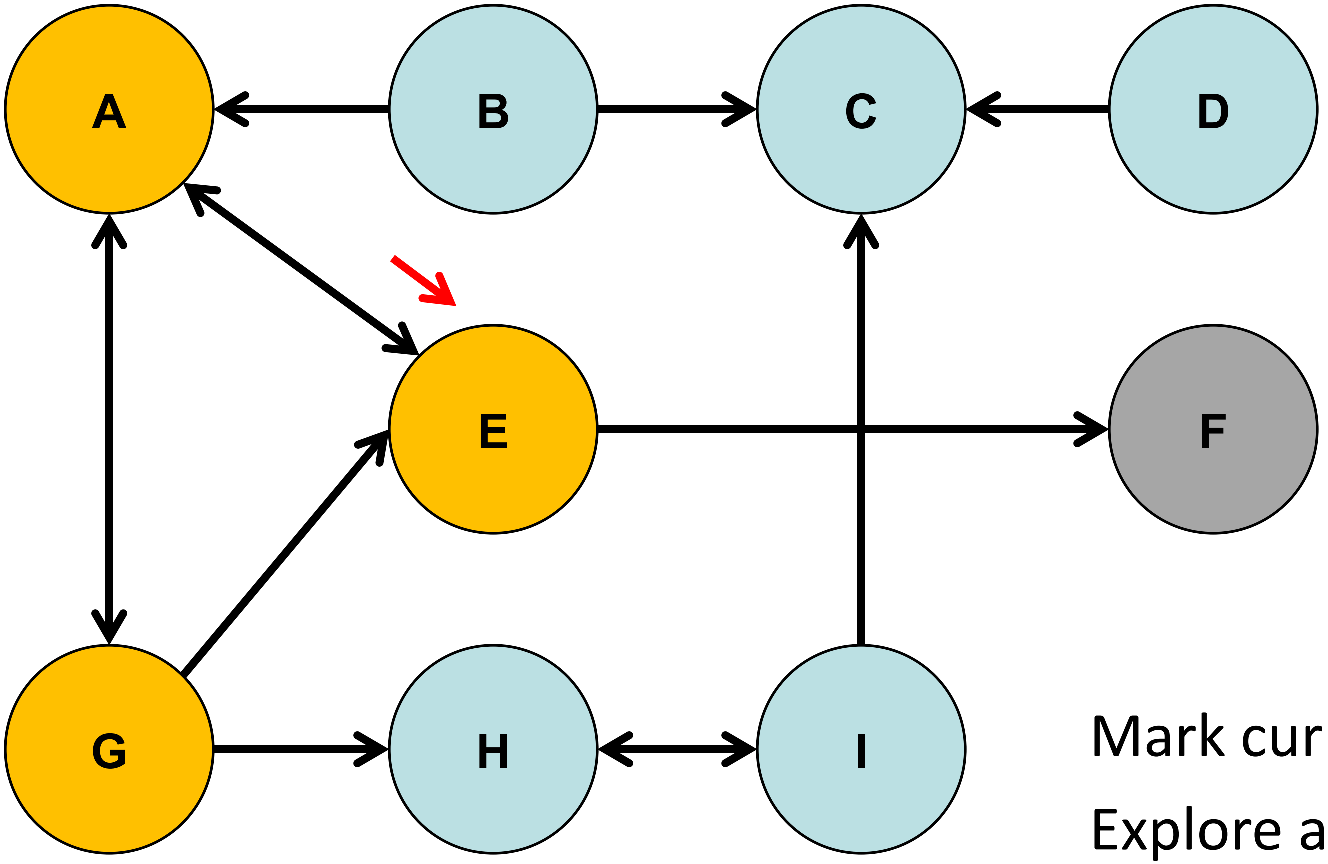
DFS



Mark current as visited

Explore all the unvisited nodes from this node

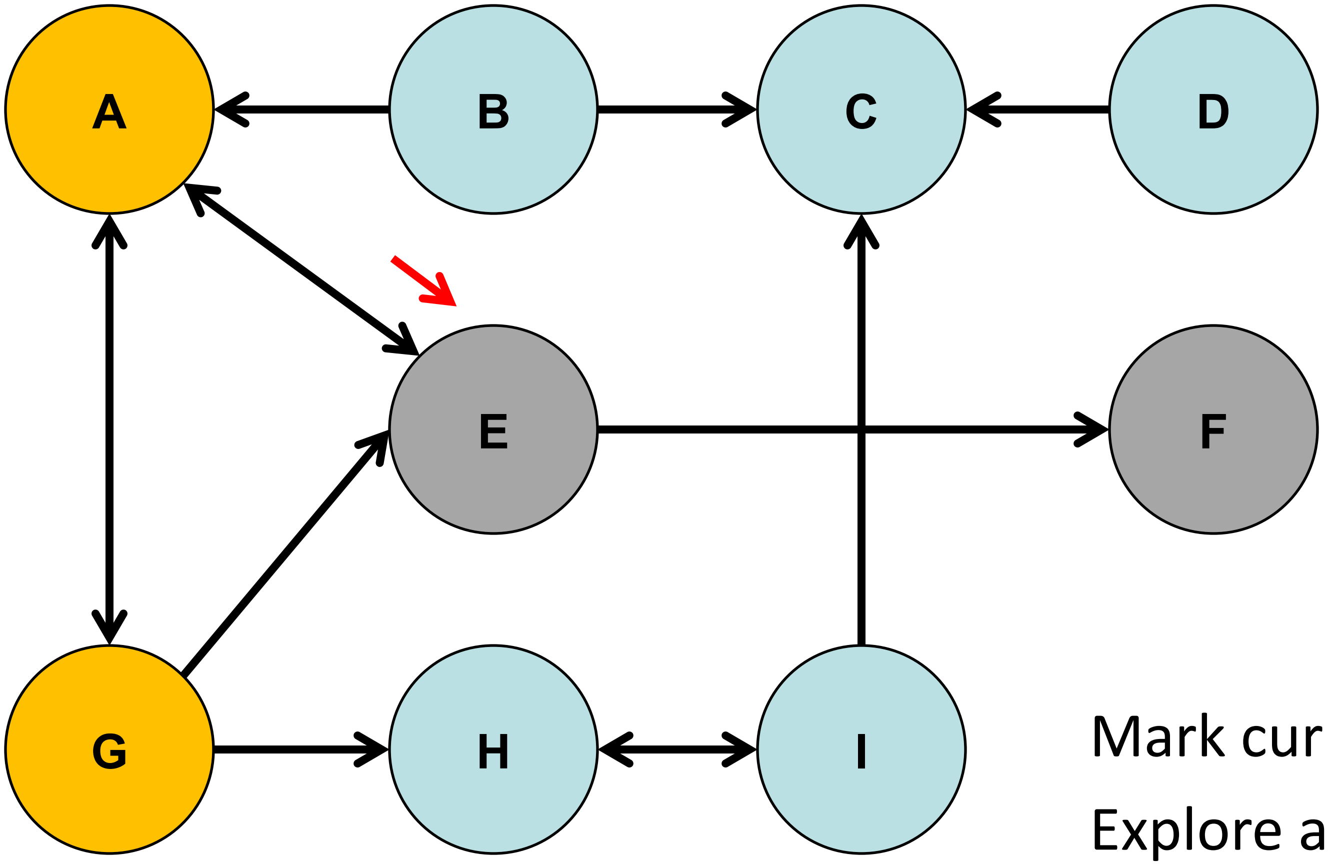
DFS



Mark current as visited

Explore all the unvisited nodes from this node

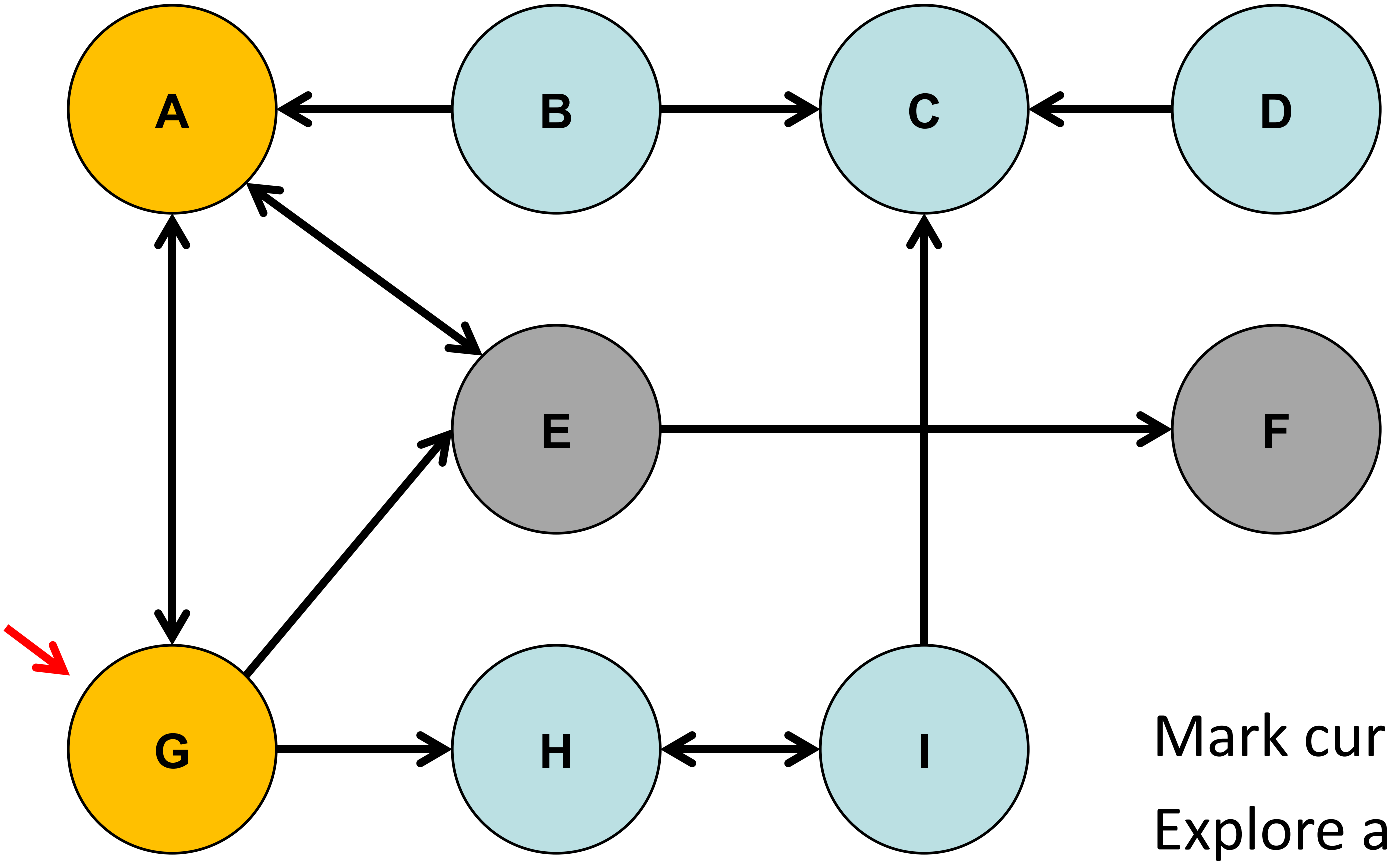
DFS



Mark current as visited

Explore all the unvisited nodes from this node

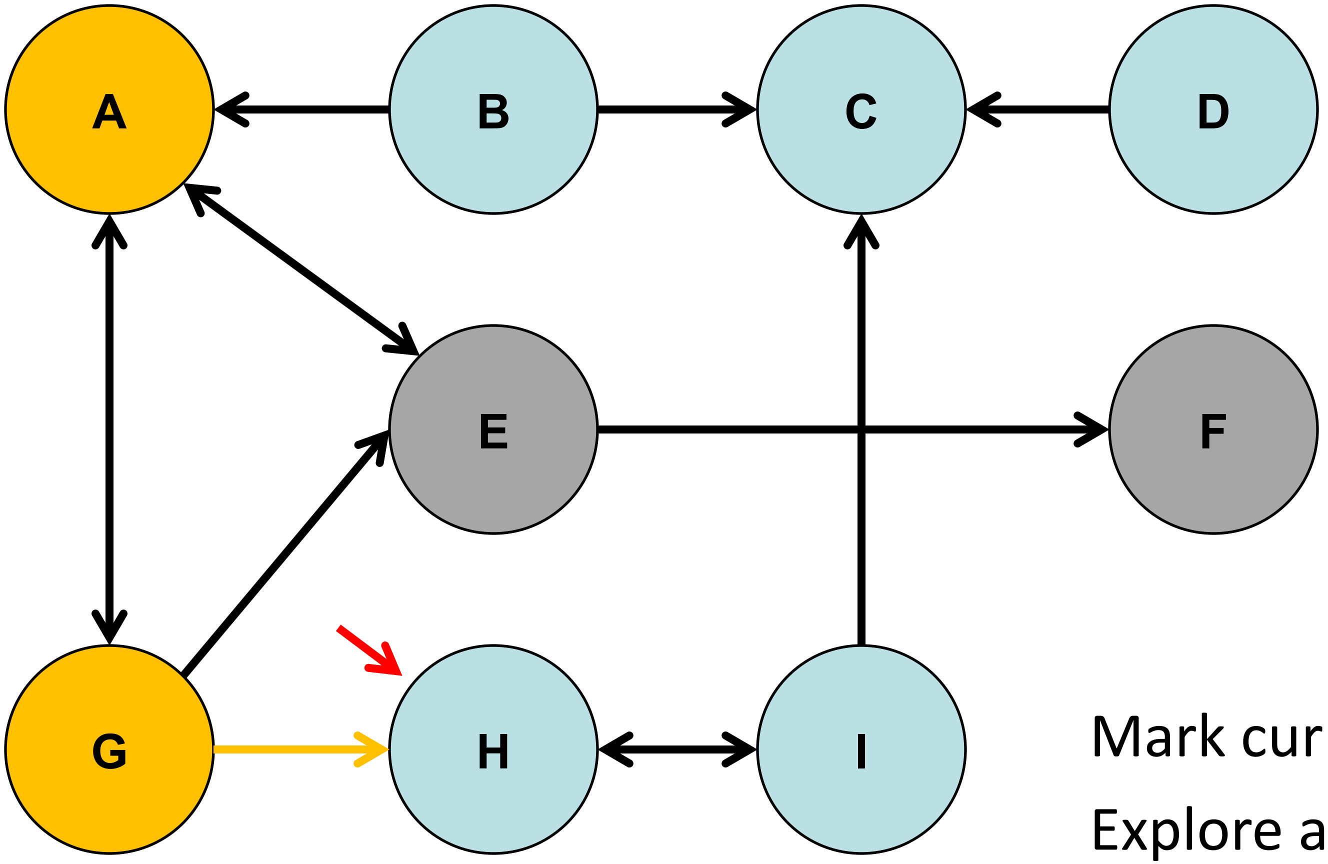
DFS



Mark current as visited

Explore all the unvisited nodes from this node

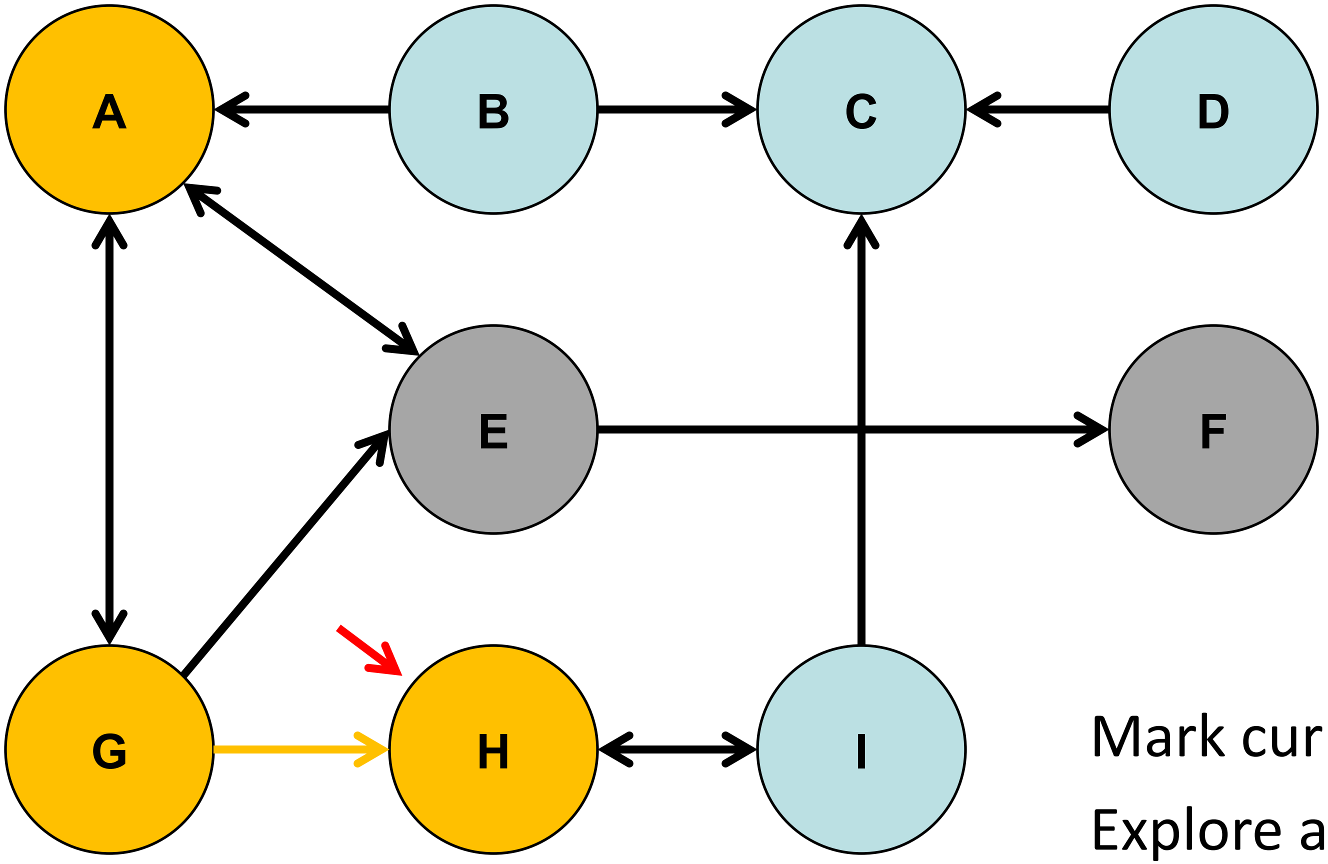
DFS



Mark current as visited

Explore all the unvisited nodes from this node

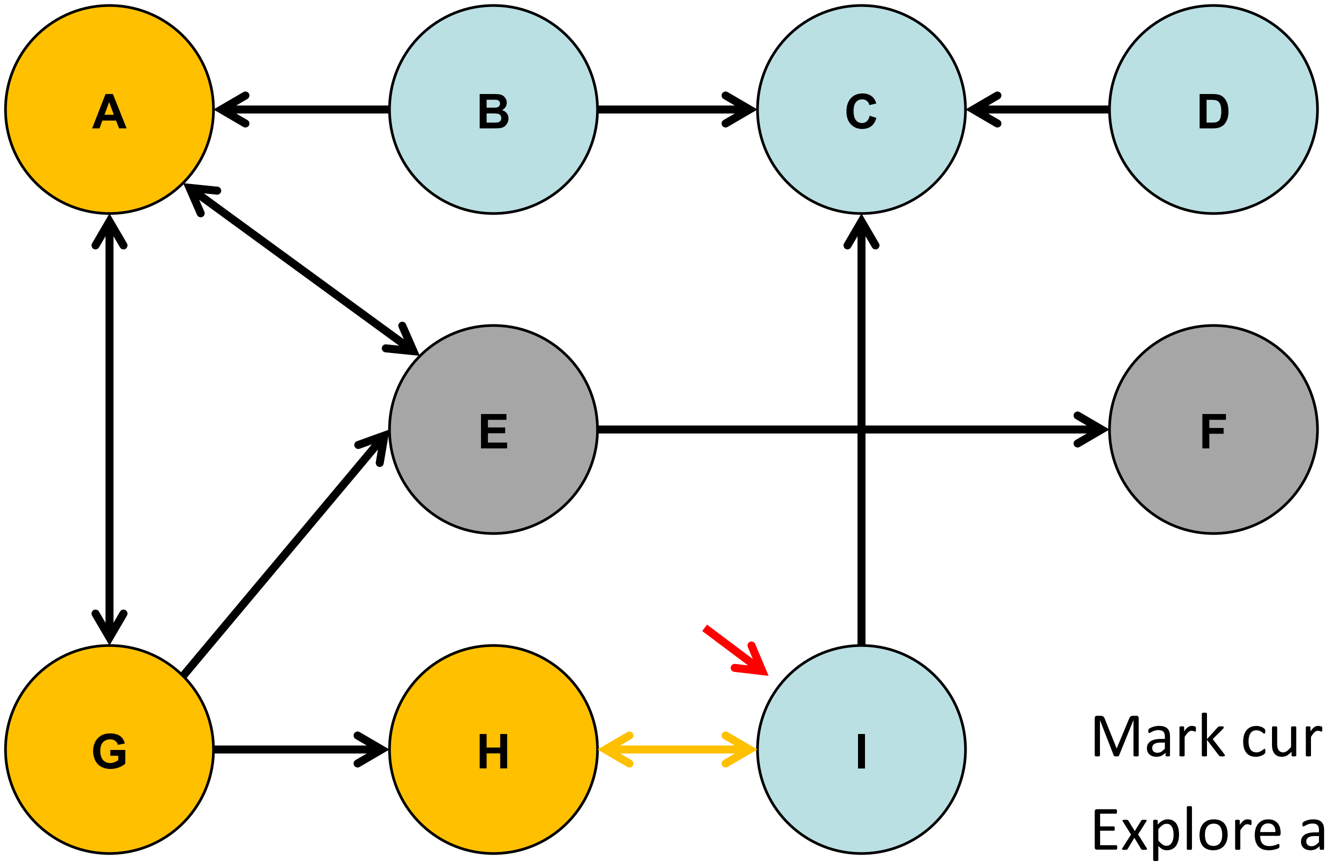
DFS



Mark current as visited

Explore all the unvisited nodes from this node

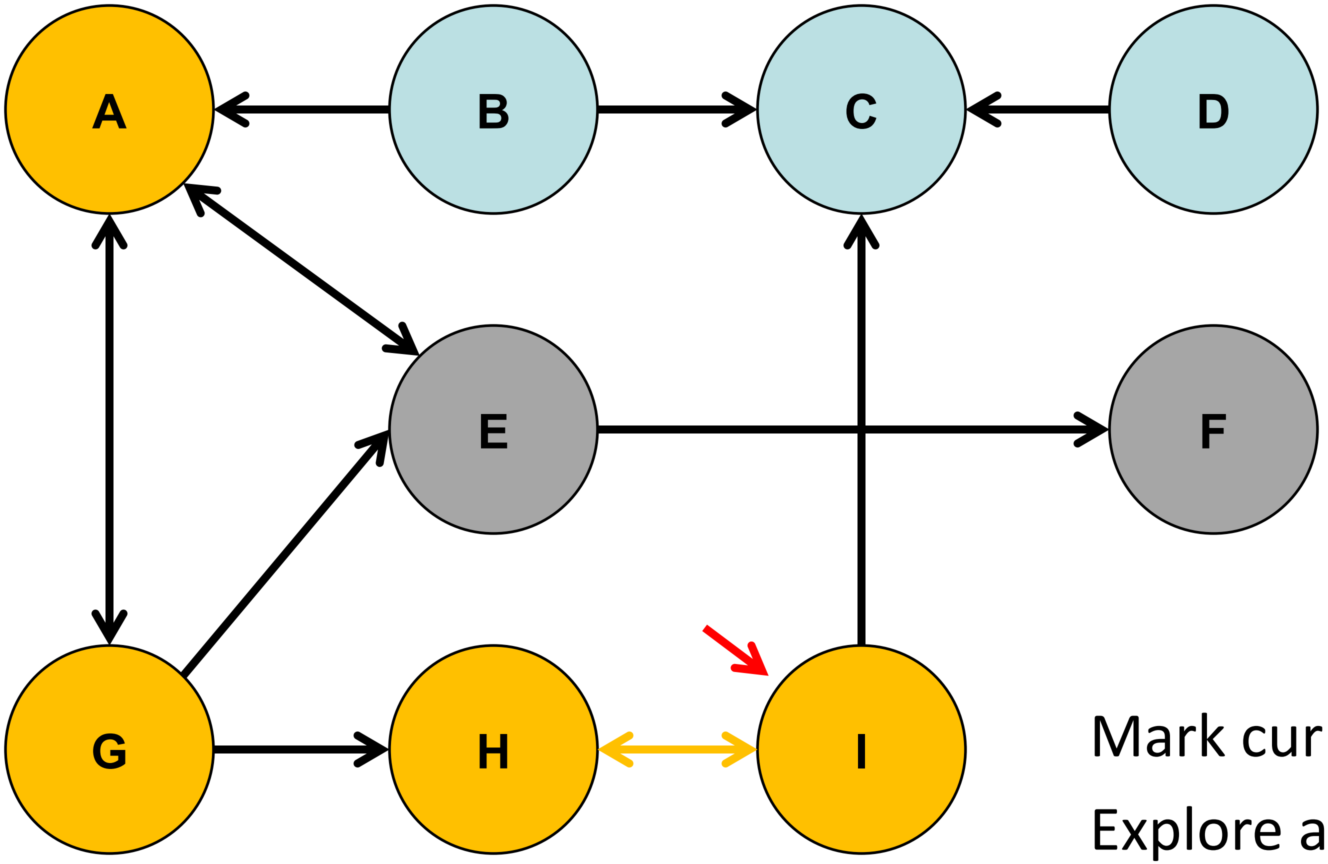
DFS



Mark current as visited

Explore all the unvisited nodes from this node

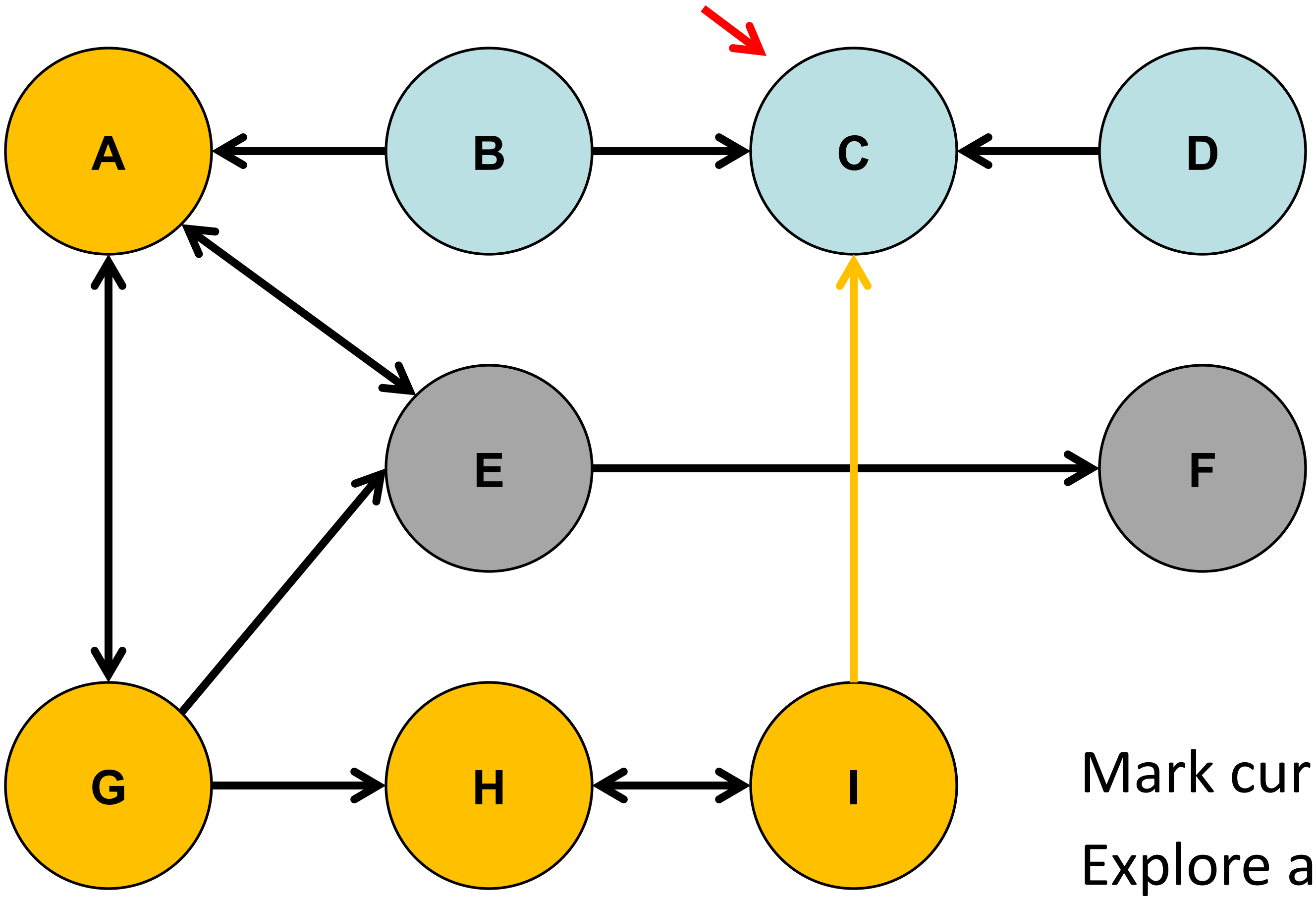
DFS



Mark current as visited

Explore all the unvisited nodes from this node

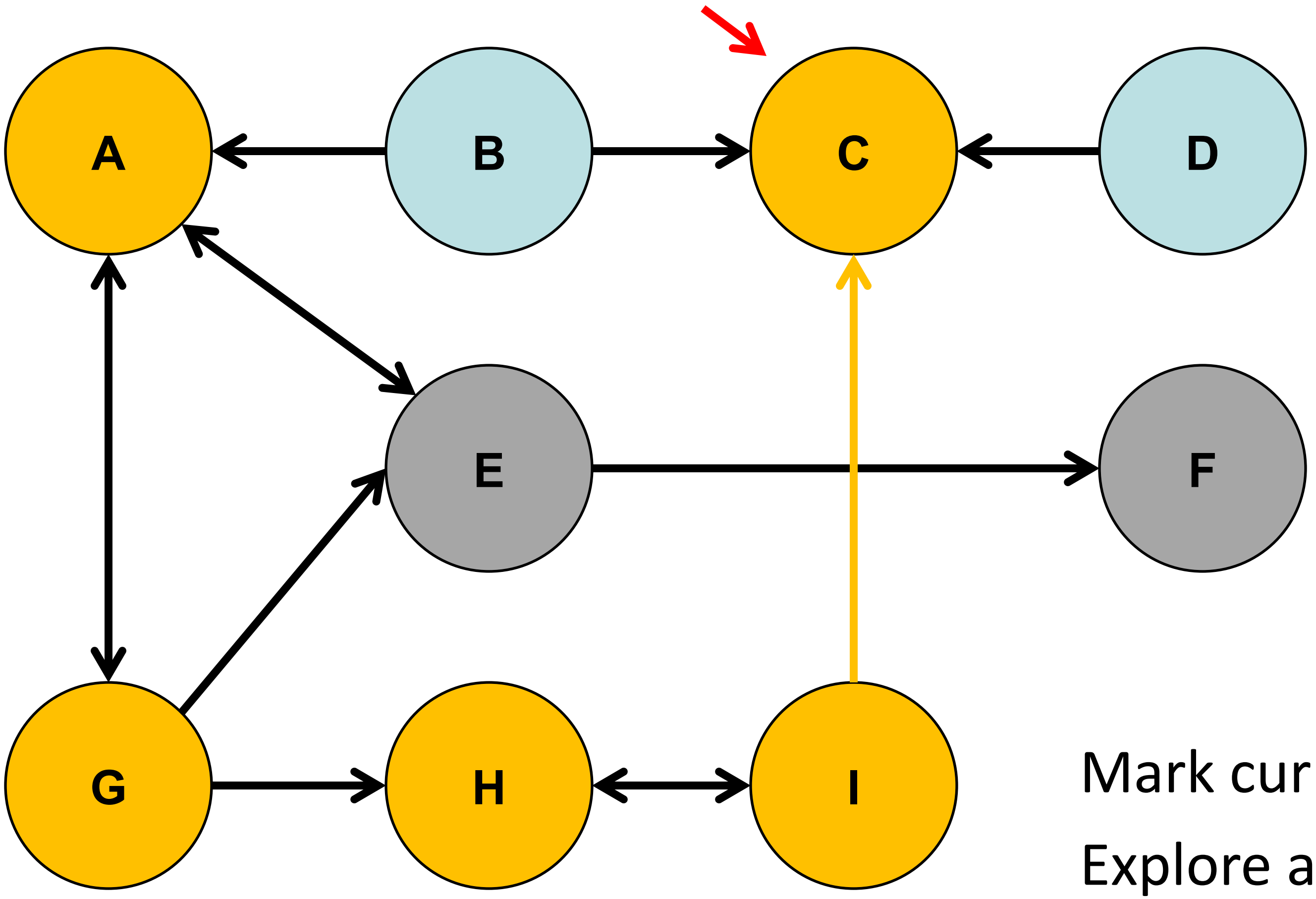
DFS



Mark current as visited

Explore all the unvisited nodes from this node

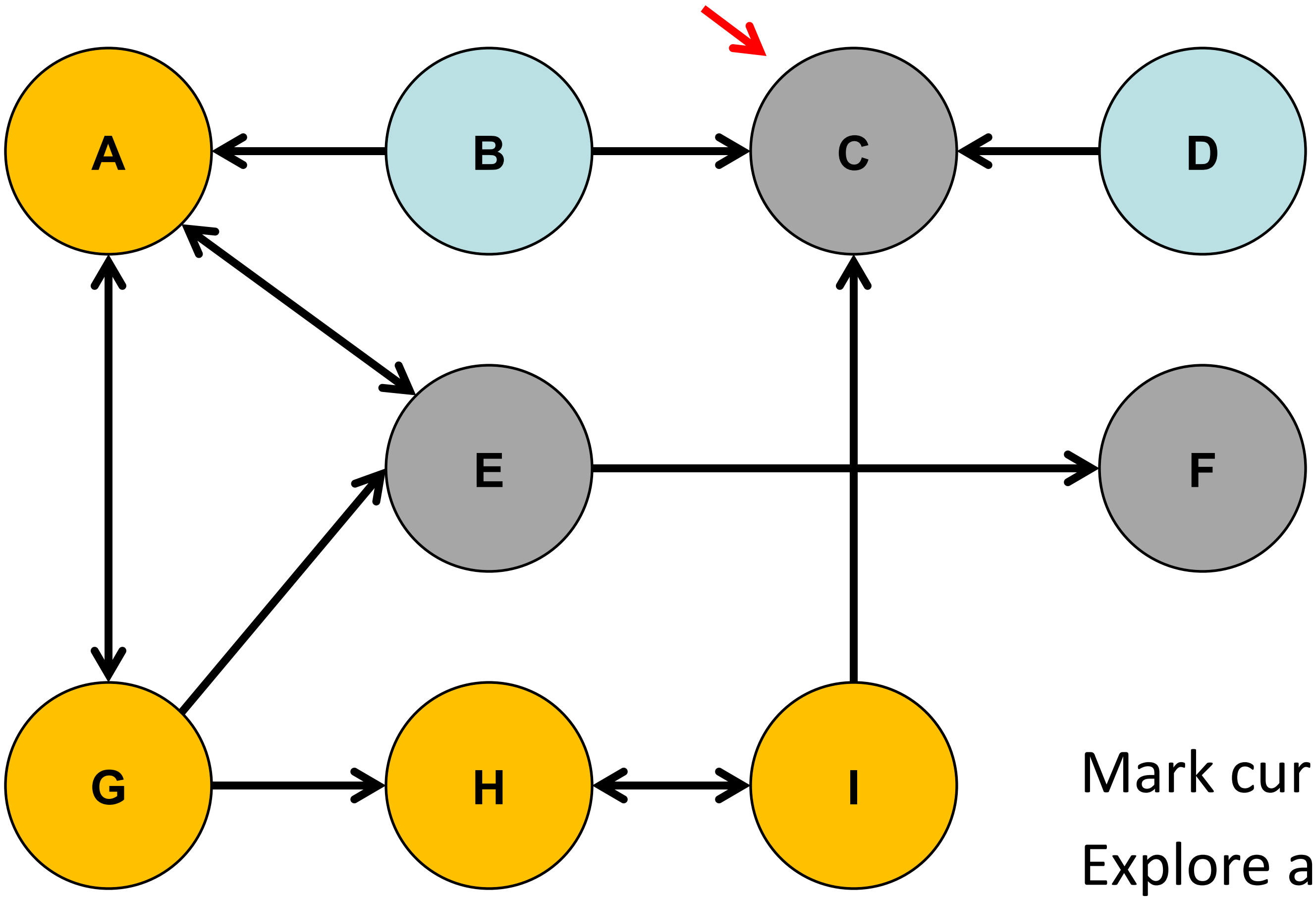
DFS



Mark current as visited

Explore all the unvisited nodes from this node

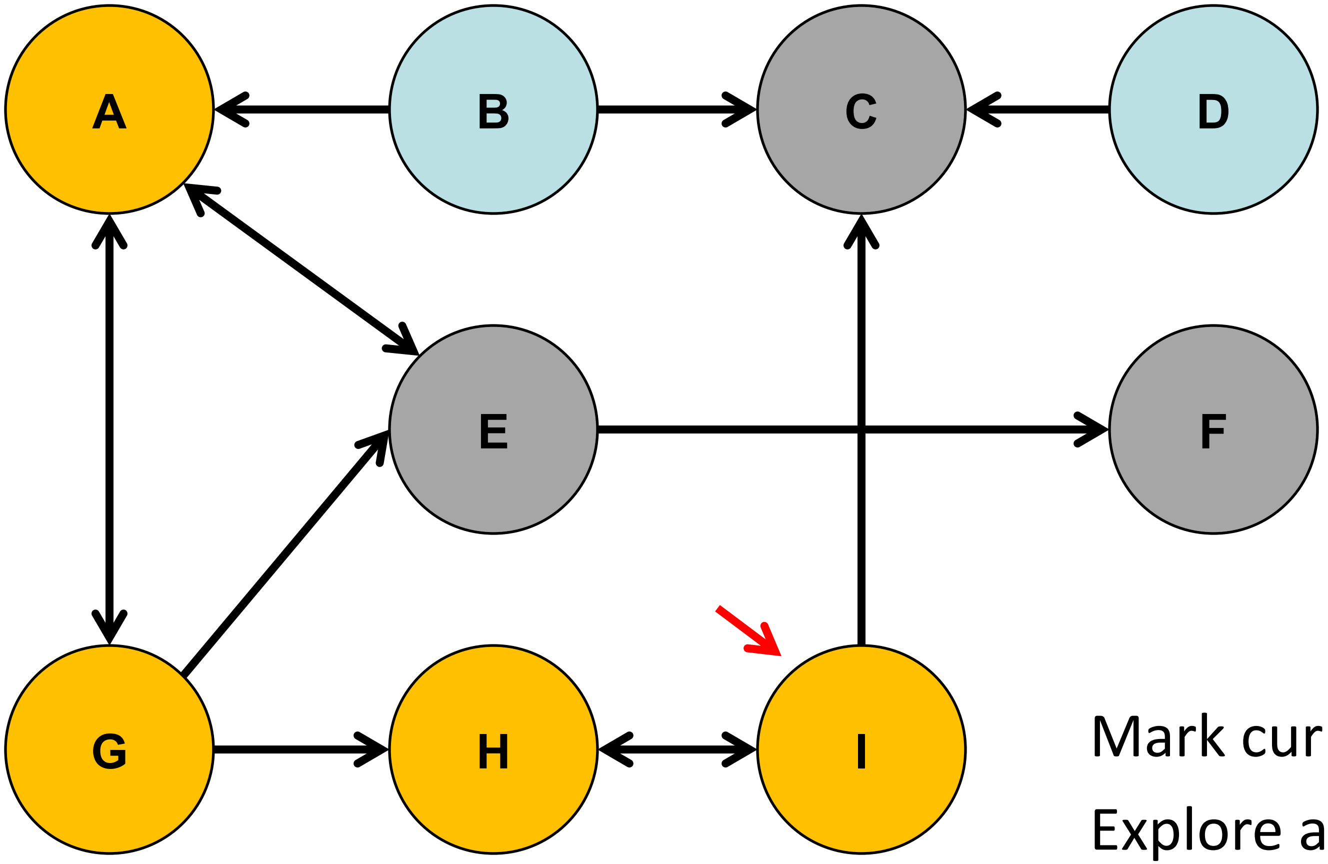
DFS



Mark current as visited

Explore all the unvisited nodes from this node

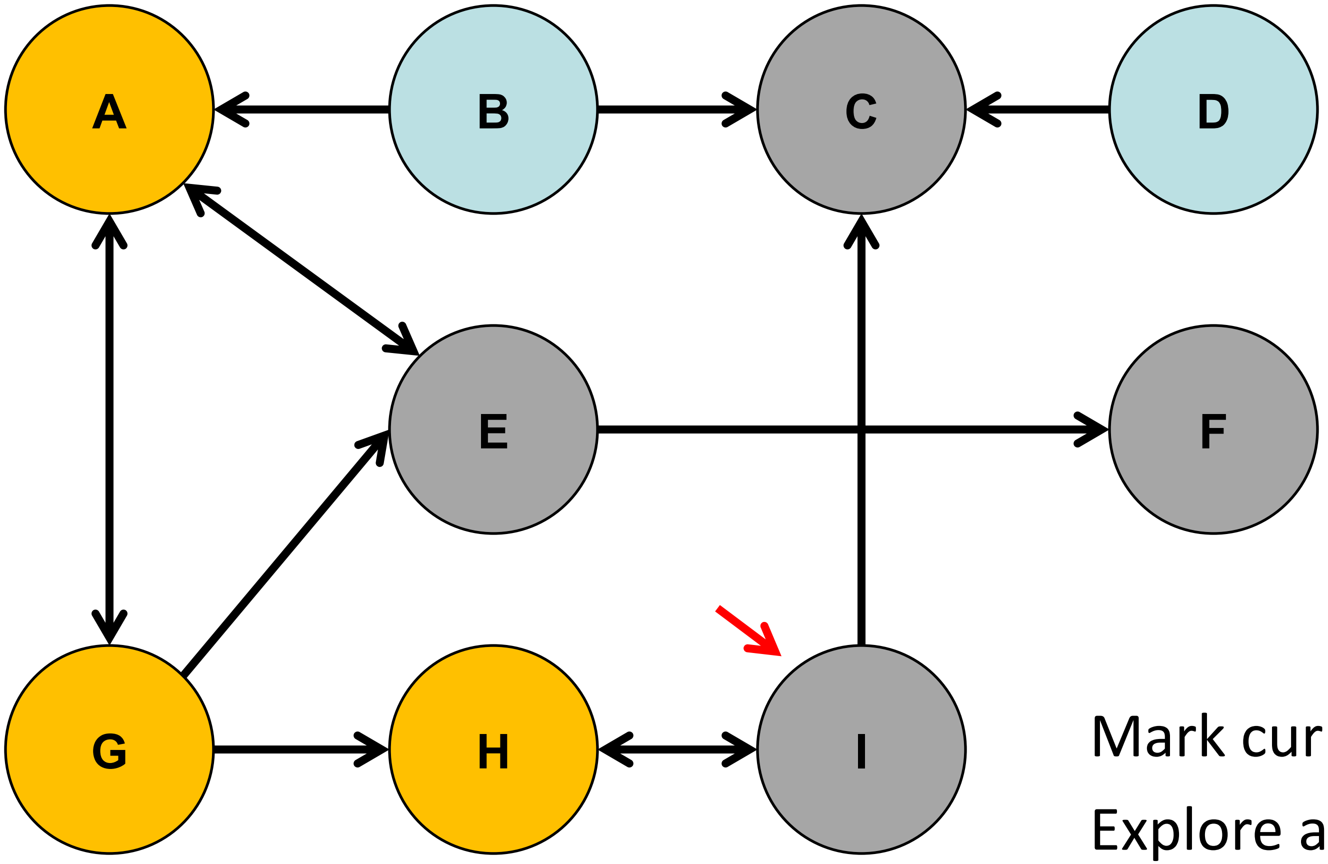
DFS



Mark current as visited

Explore all the unvisited nodes from this node

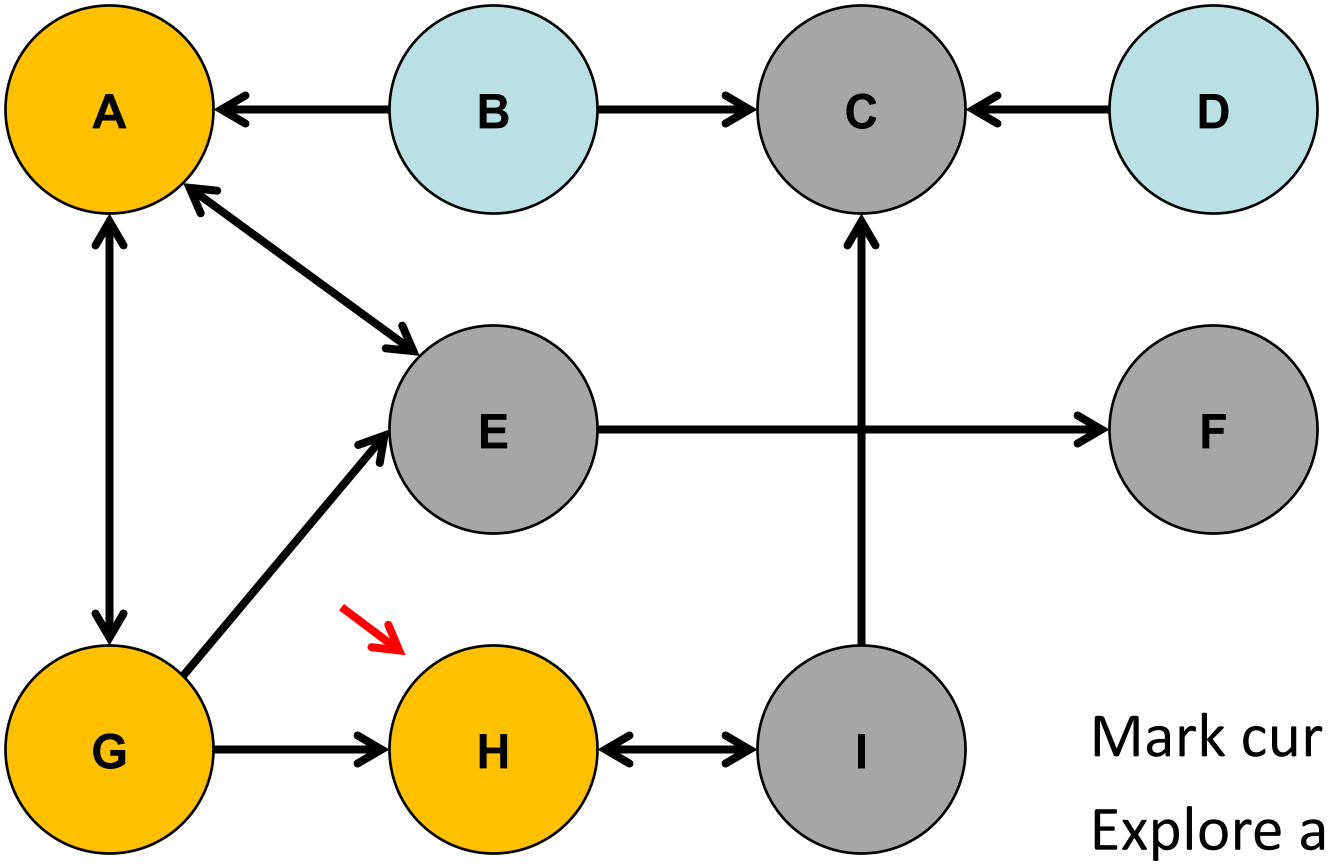
DFS



Mark current as visited

Explore all the unvisited nodes from this node

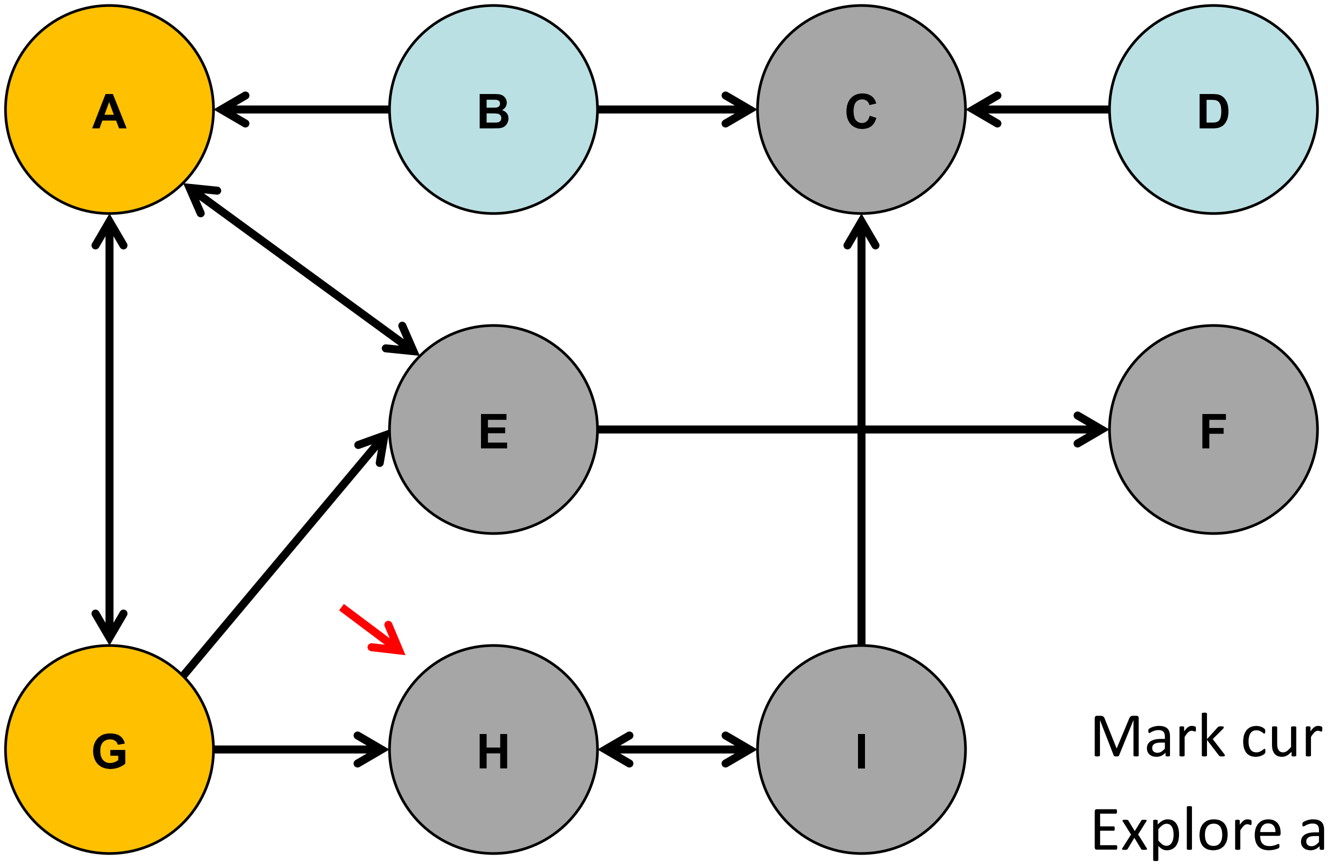
DFS



Mark current as visited

Explore all the unvisited nodes from this node

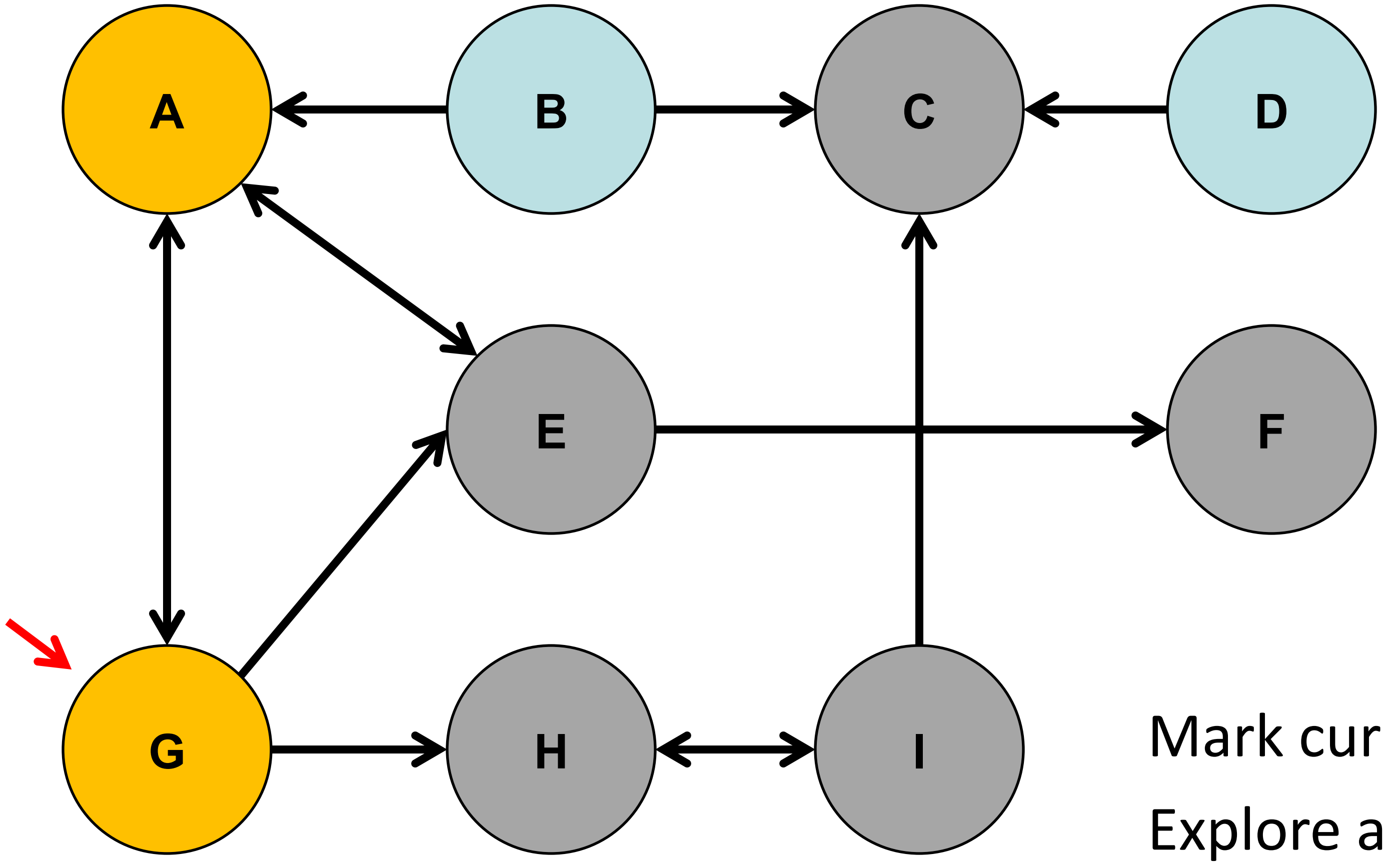
DFS



Mark current as visited

Explore all the unvisited nodes from this node

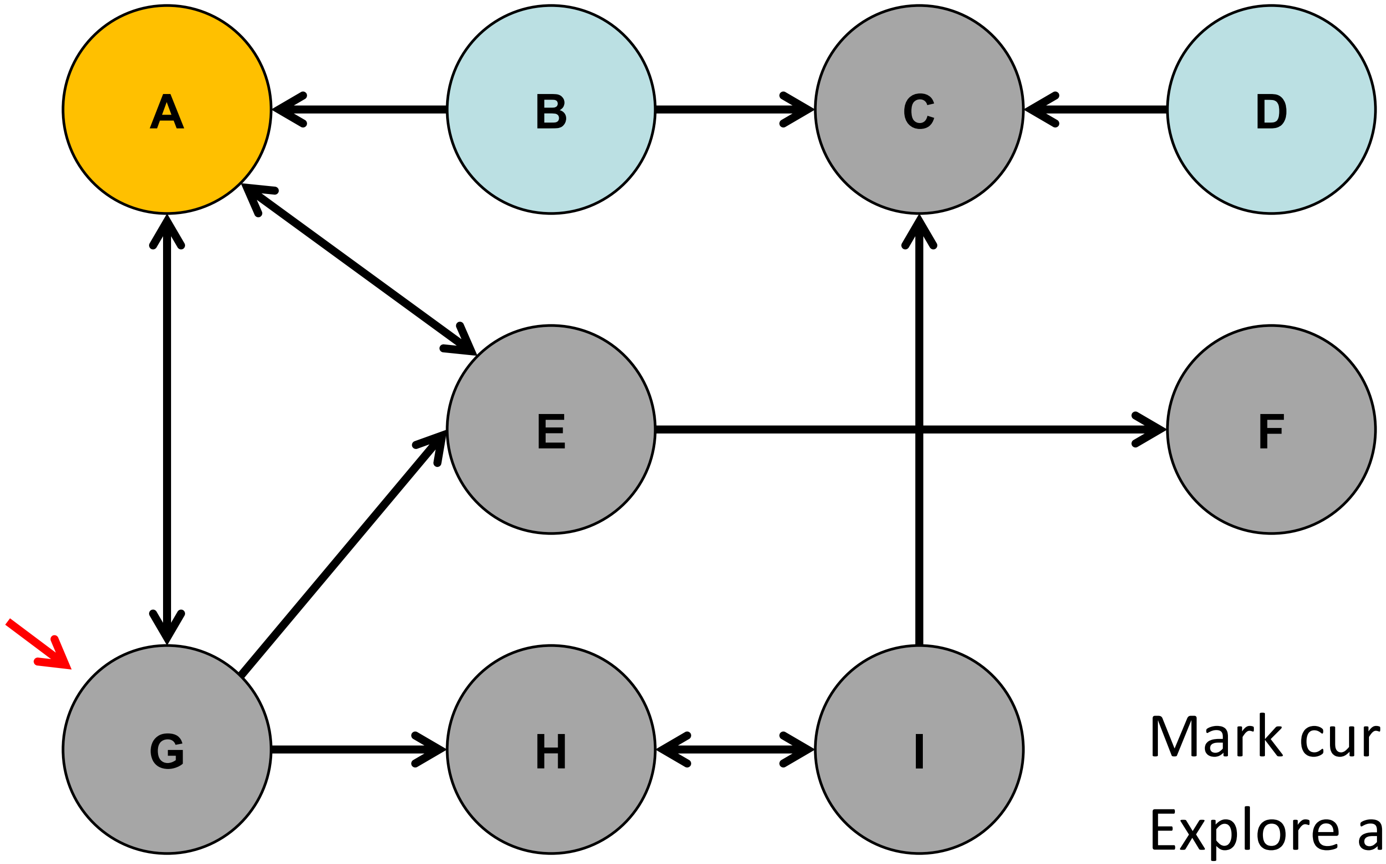
DFS



Mark current as visited

Explore all the unvisited nodes from this node

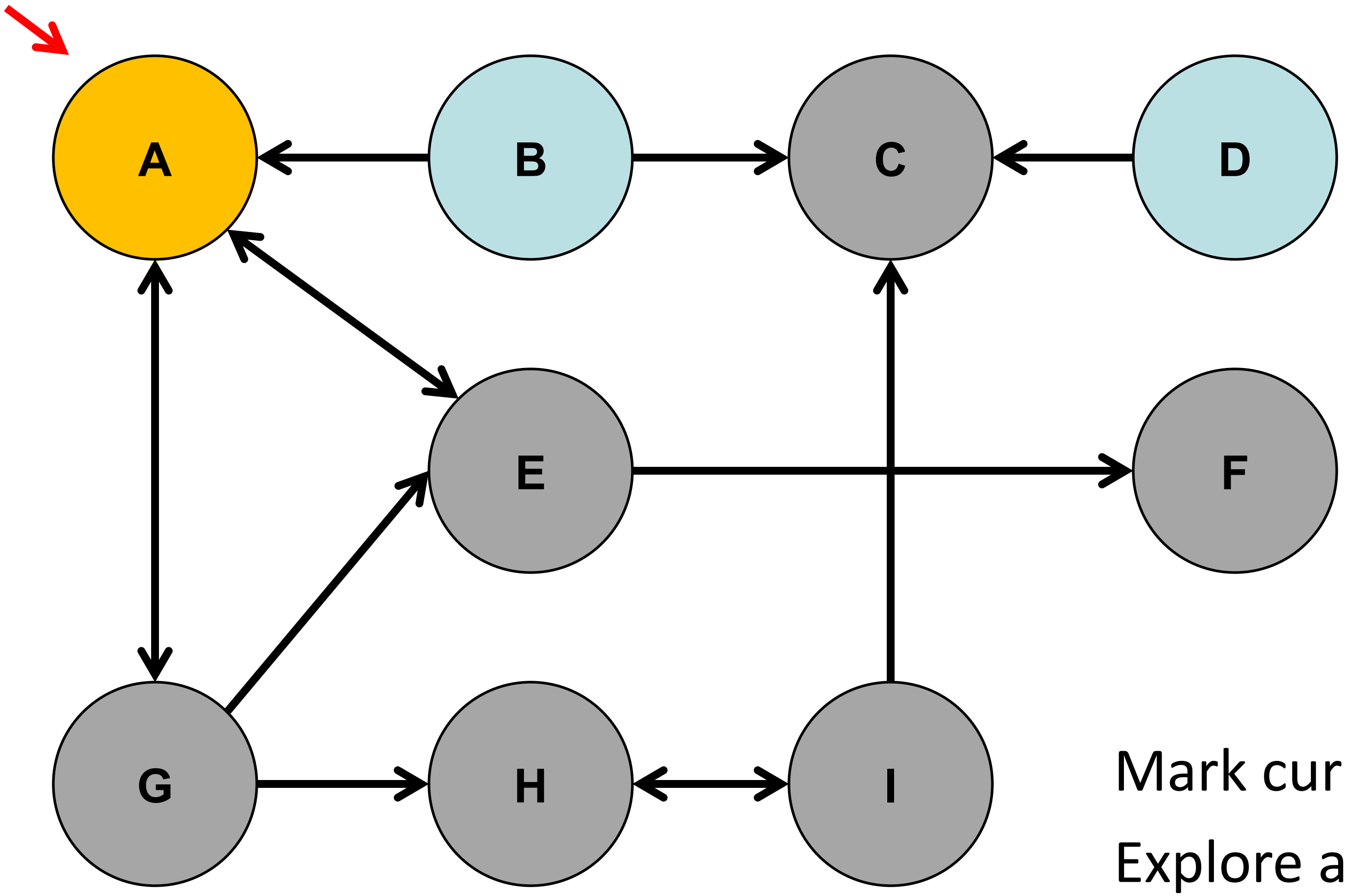
DFS



Mark current as visited

Explore all the unvisited nodes from this node

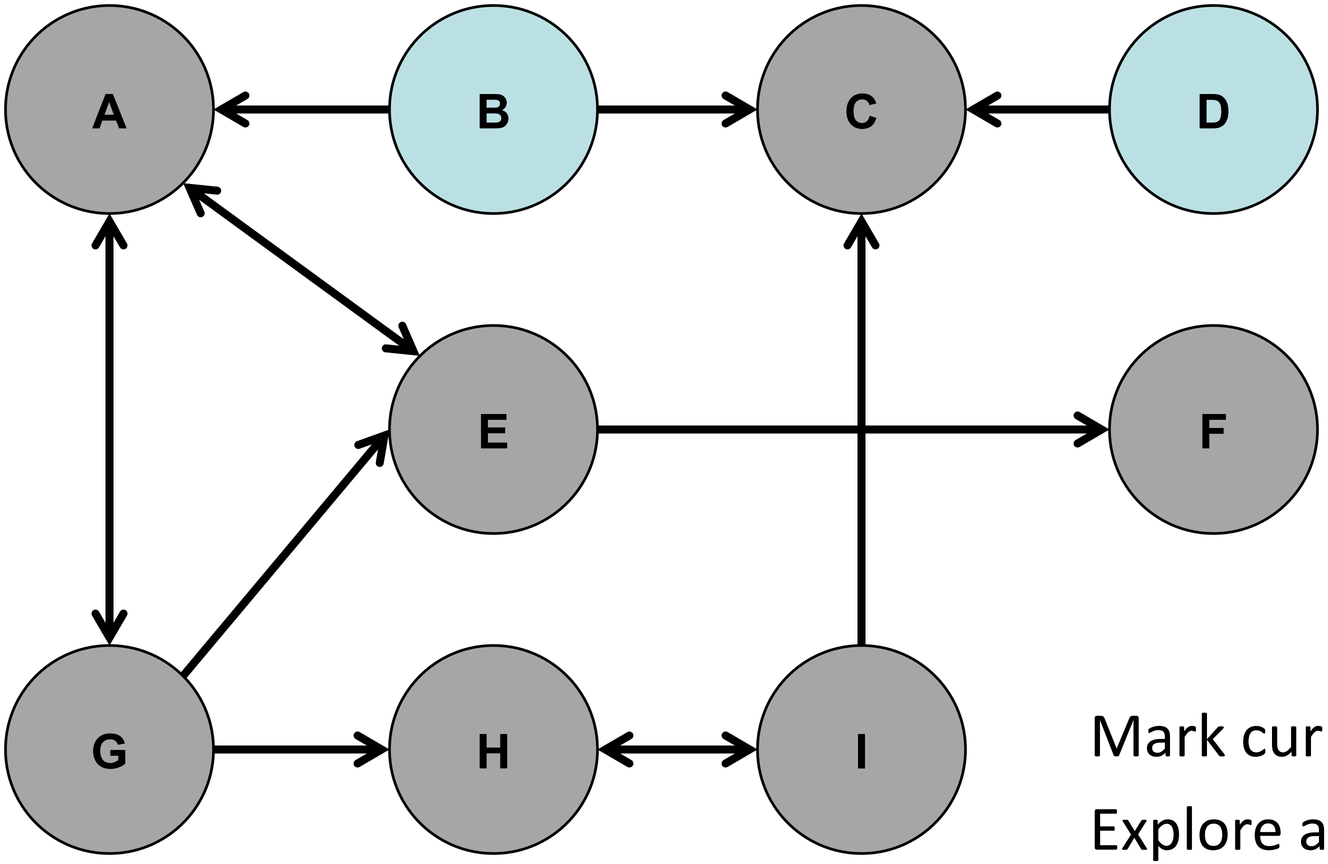
DFS



Mark current as visited

Explore all the unvisited nodes from this node

DFS



Mark current as visited

Explore all the unvisited nodes from this node

DFS Details

- In an n-node, m-edge graph, takes $O(m + n)$ time with an adjacency list
  - Visit each edge once, visit each node at most once
- Pseudocode:

```
dfs from v1:
    mark v1 as seen.
    for each of v1's unvisited neighbors n:
      dfs(n)
```

- How could we modify the pseudocode to look for a specific path?

DFS that finds path

dfs from v_1 to v_2:

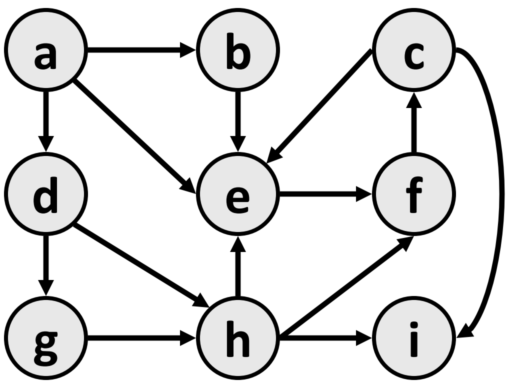
mark v_1 as **visited**, and **add to path**.

选对象

perform a **dfs** from each of v_1's unvisited neighbors n to v_2:

if **dfs**(n, v_2) succeeds: a path is found! yay!

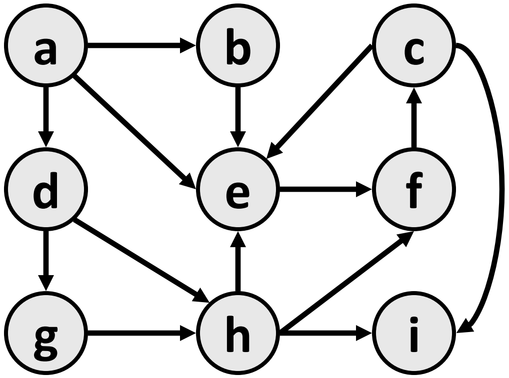
if all neighbors fail: **remove v_1 from path**.



- To retrieve the DFS path found, pass a collection parameter to each call and choose-explore-unchoose.

DFS observations

- *discovery*: DFS is guaranteed to find *a* path if one exists.



- *retrieval*: It is easy to retrieve exactly what the path is (the sequence of edges taken) if we find it
  - choose - explore - unchoose

- *optimality*: not optimal. DFS is guaranteed to find a path, not necessarily the best/shortest path
  - Example: dfs(a, i) returns {a, b, e, f, c, i} rather than {a, d, h, i}.

Plan For Today

- **Recap**: Graphs
- **Practice**: Twitter Influence
- Depth-First Search (DFS)
- Announcements
- Breadth-First Search (BFS)

Announcements

- Assignment 7 will go out this Friday, is due Wed. after break
  - Short graphs assignment (Google Maps!), implementing algorithms from this week
- Assignment 8 will go out the Wed. after break, is due the last day of class (Fri)
  - Graphs and inheritance assignment (Excel!)

Plan For Today

- **Recap**: Graphs
- **Practice**: Twitter Influence
- Depth-First Search (DFS)
- Announcements
- Breadth-First Search (BFS)

Finding *Shortest* Paths

- We can find paths between two nodes, but how can we find the **shortest** path?
  - Fewest number of steps to complete a task?
  - Least amount of edits between two words?
- When have we solved this problem before?

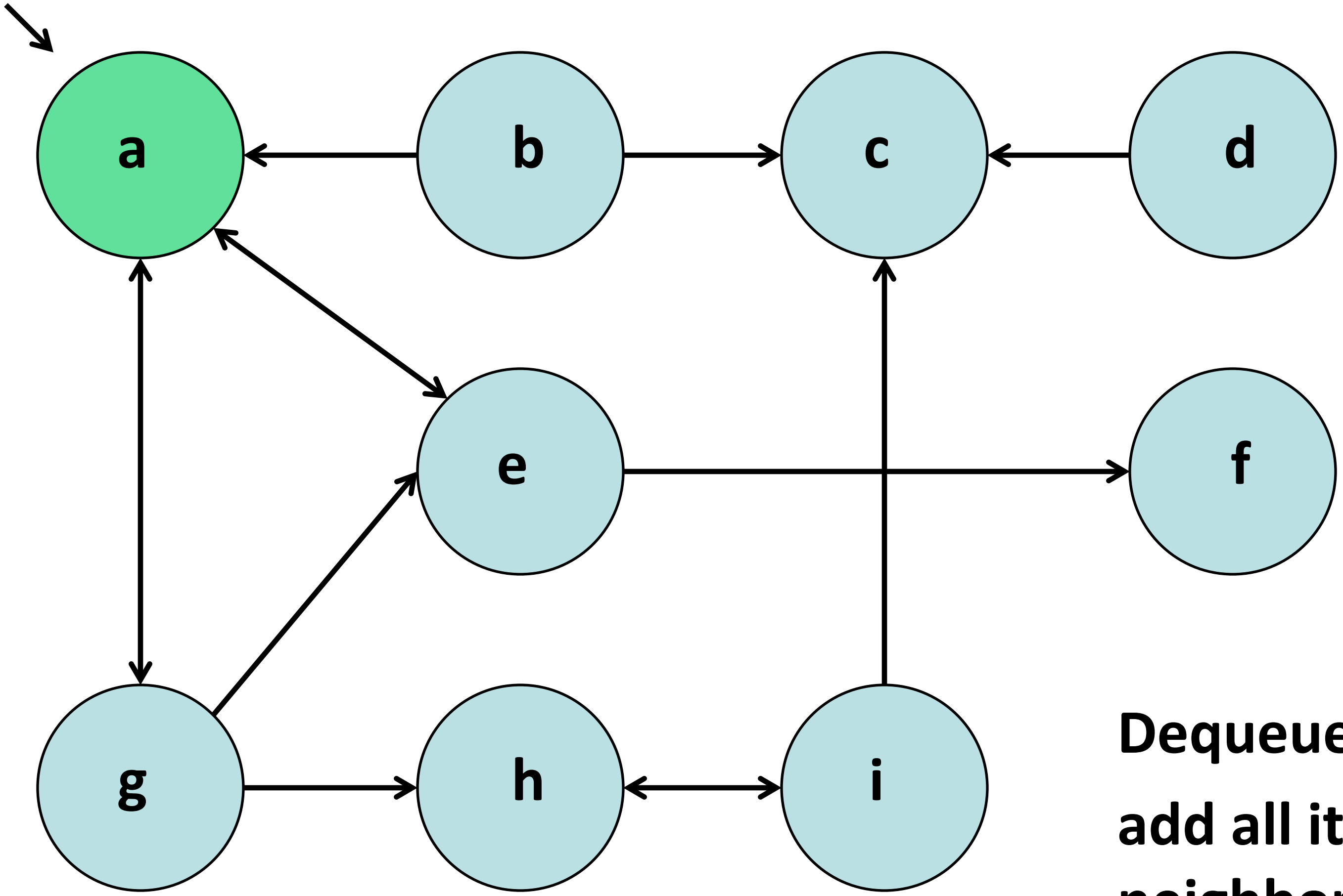
Breadth-First Search (BFS)

- Idea: processing a node involves knowing we need to visit all its neighbors (just like DFS)
- Need to keep a TODO list of nodes to process

Breadth-First Search (BFS)

- Keep a Queue of nodes as our TODO list
- Idea: dequeue a node, enqueue all its neighbors
- Still will return the same nodes as reachable, just might have shorter paths

BFS

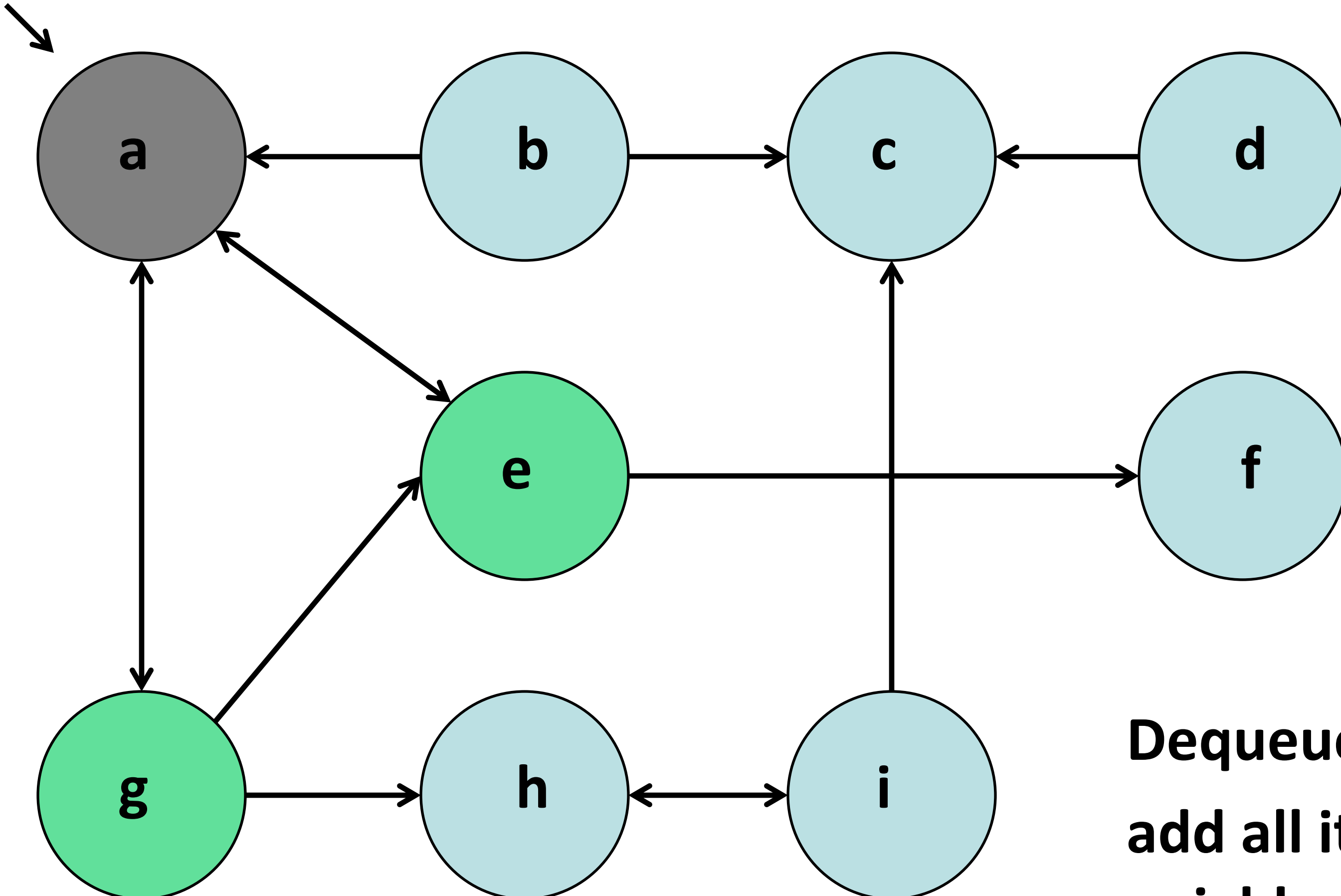


Dequeue a node

add all its unseen neighbors to the queue

queue: a

BFS

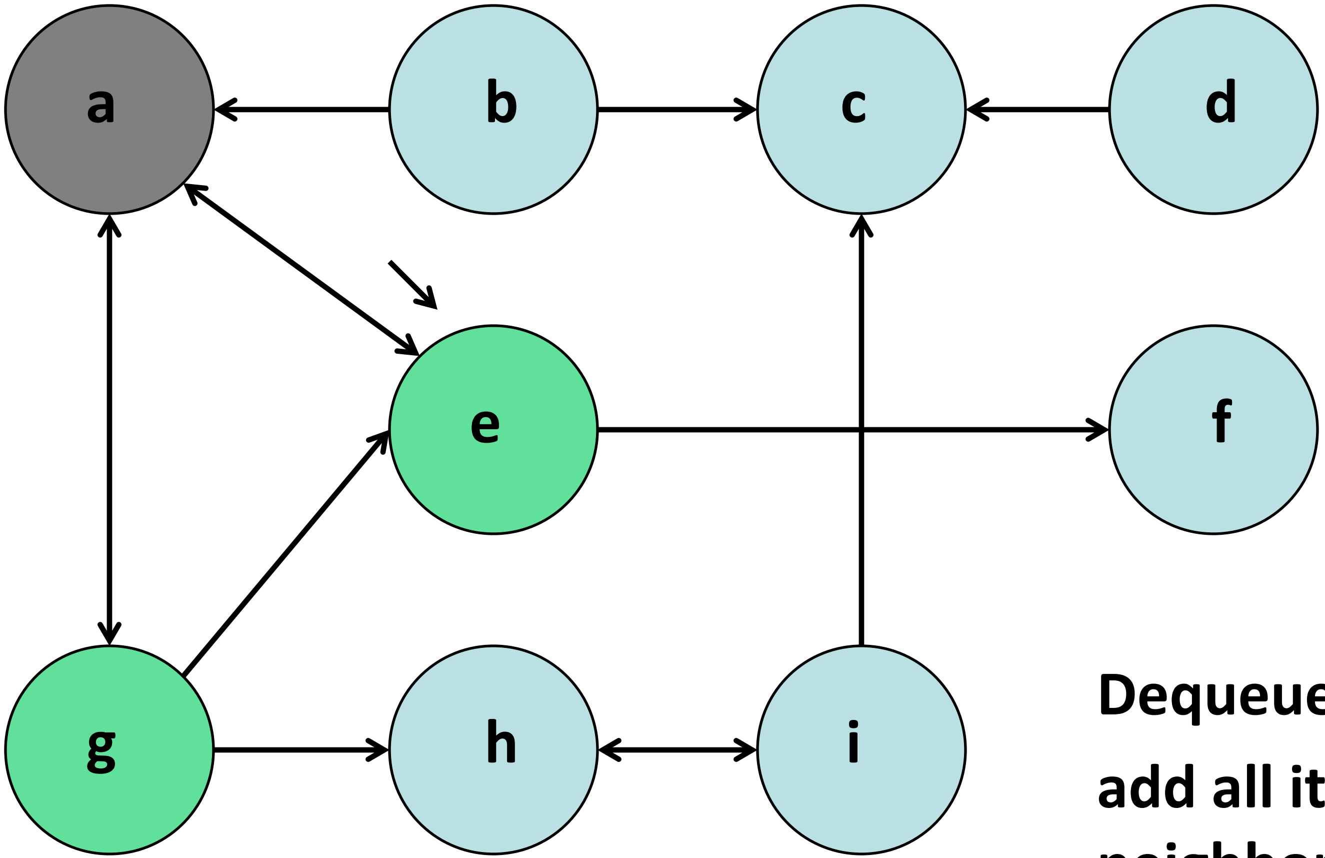


Dequeue a node

add all its unseen neighbors to the queue

queue: e, g

BFS

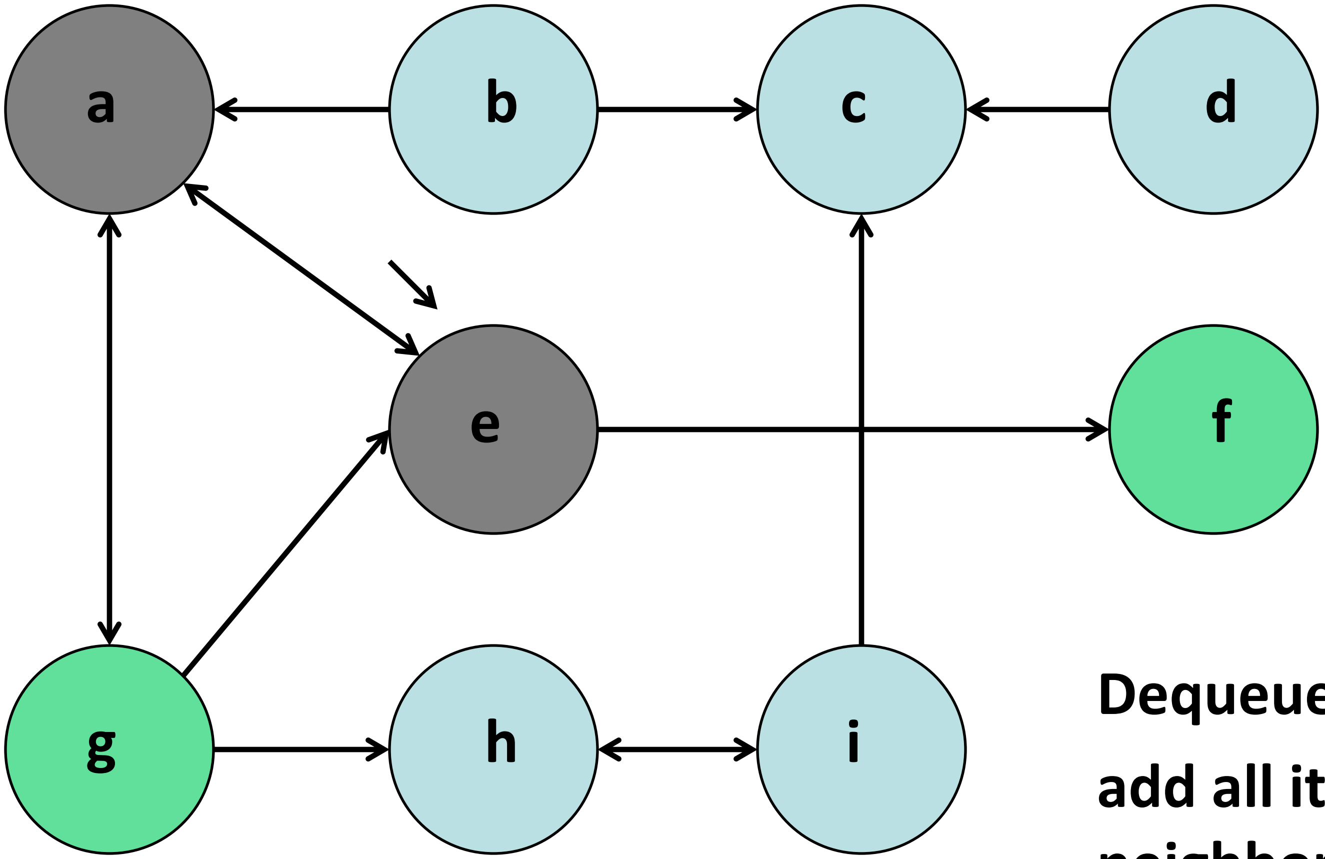


Dequeue a node

add all its unseen neighbors to the queue

queue: e, g

BFS

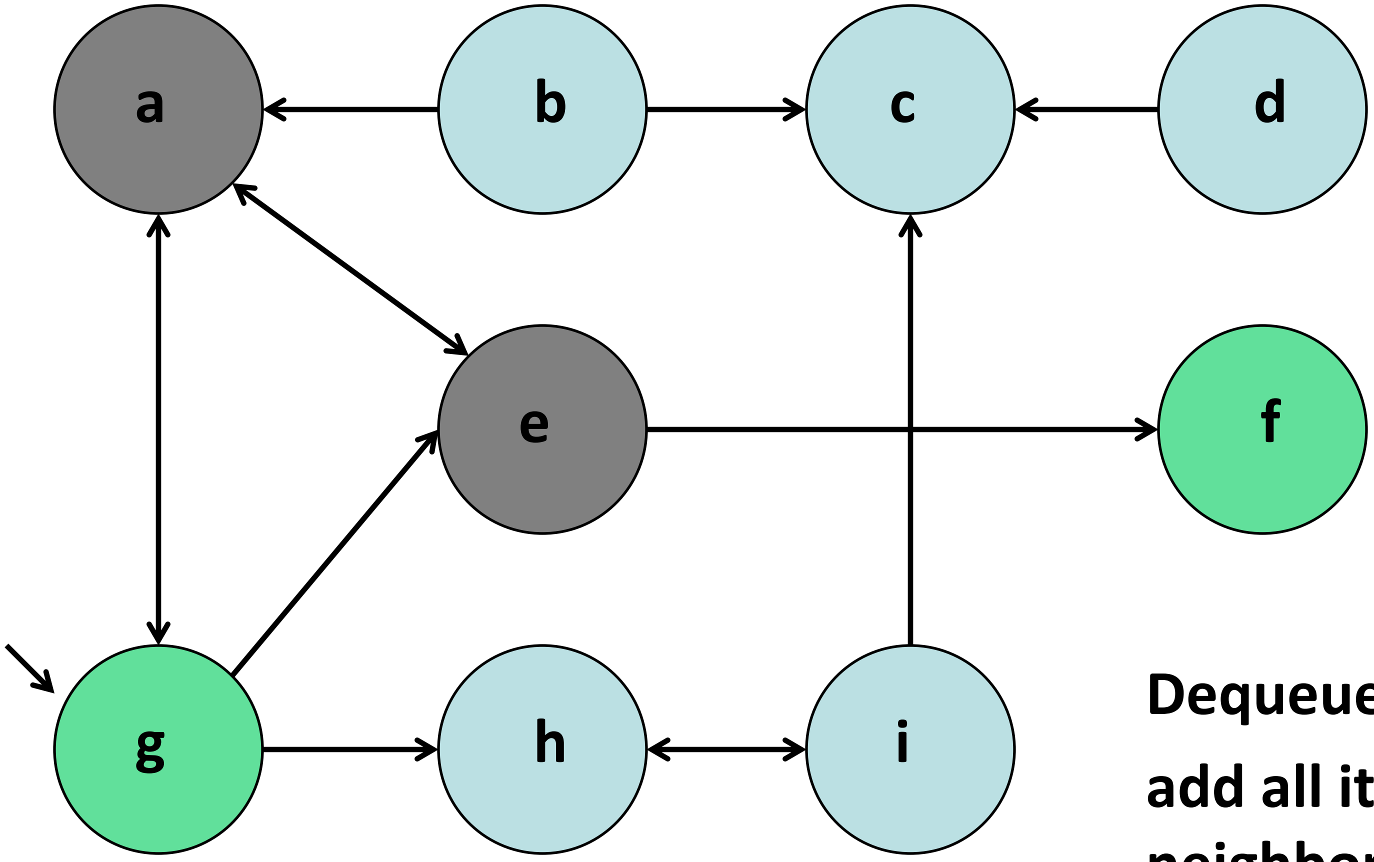


Dequeue a node

add all its unseen neighbors to the queue

queue: g, f

BFS

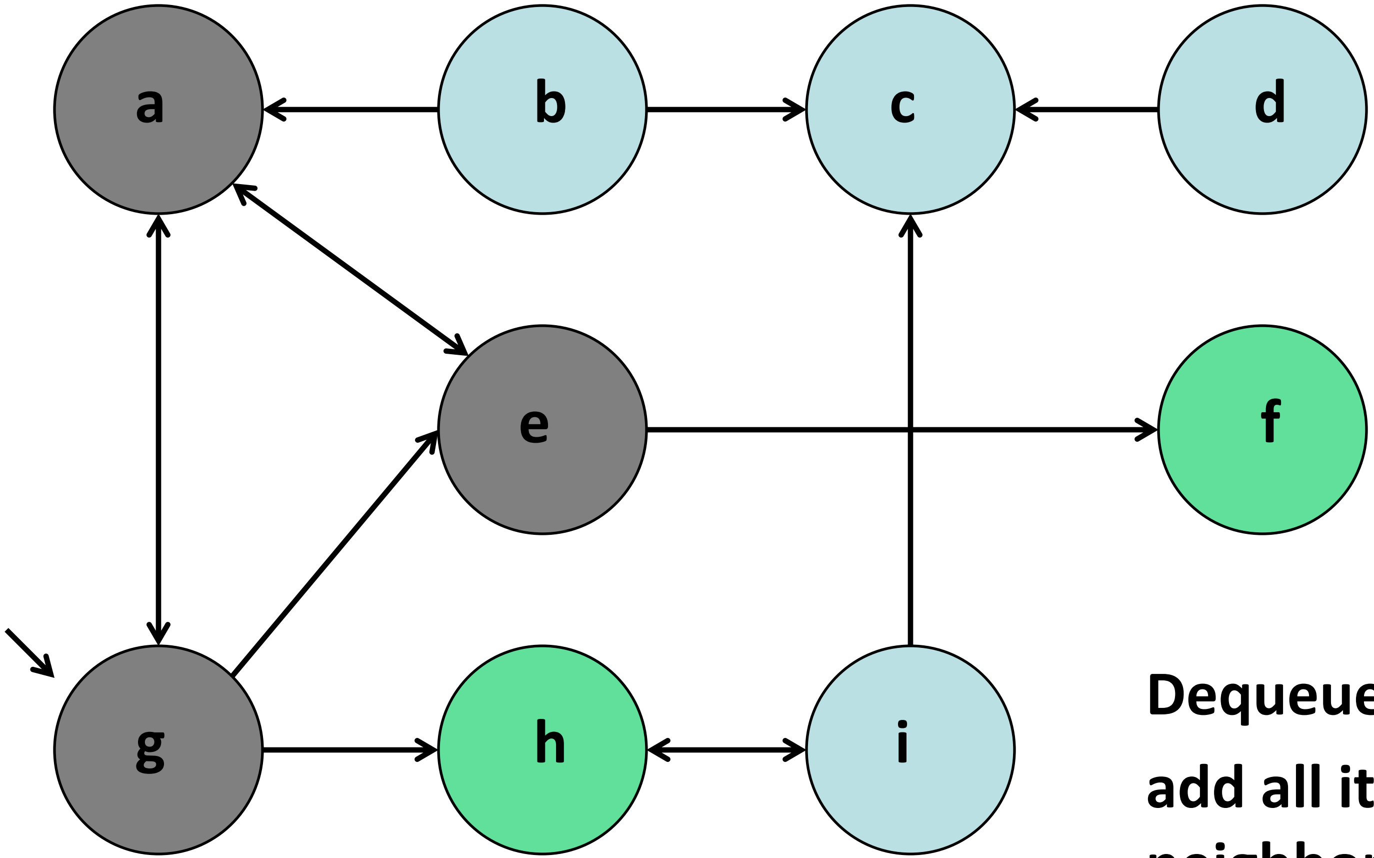


Dequeue a node

add all its unseen neighbors to the queue

queue: g, f

BFS

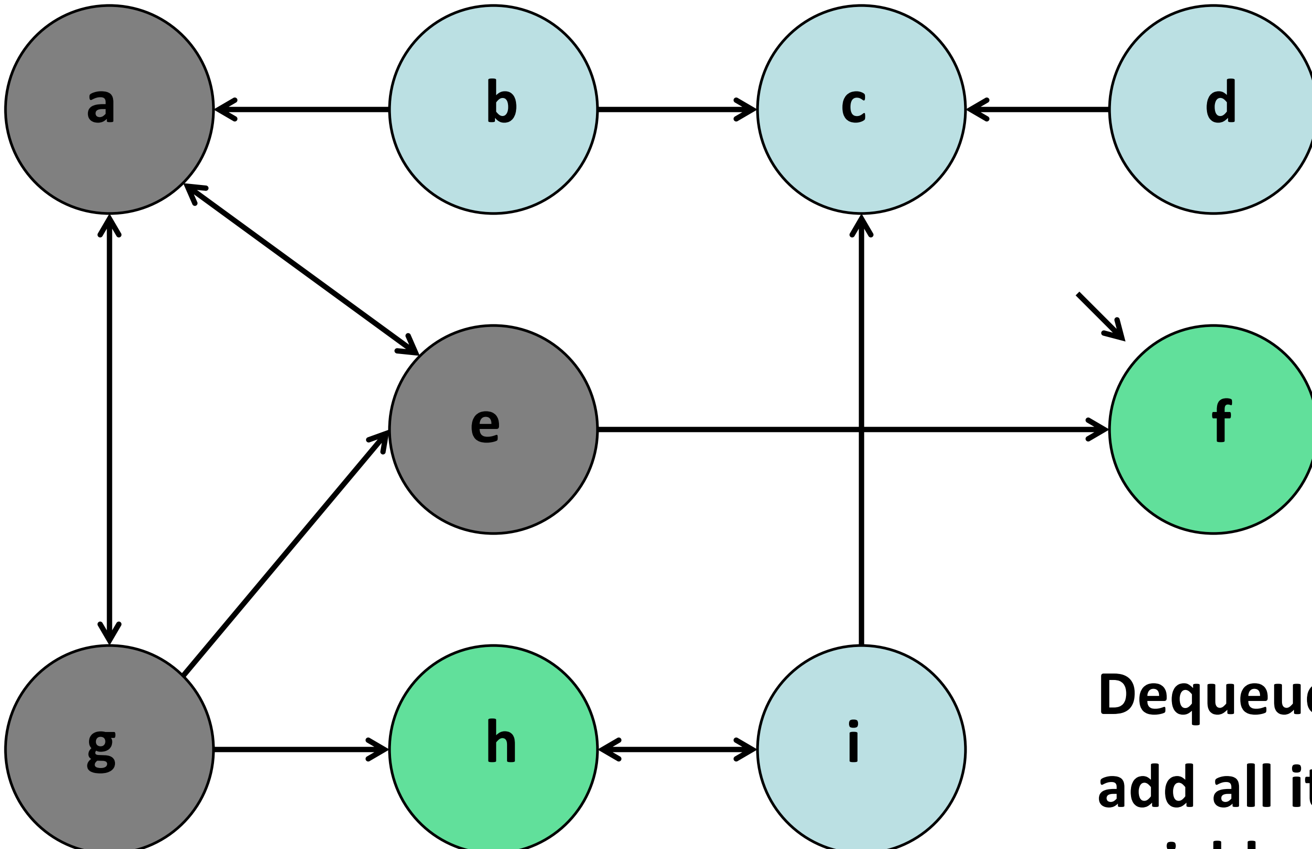


Dequeue a node

add all its unseen neighbors to the queue

queue: f, h

BFS

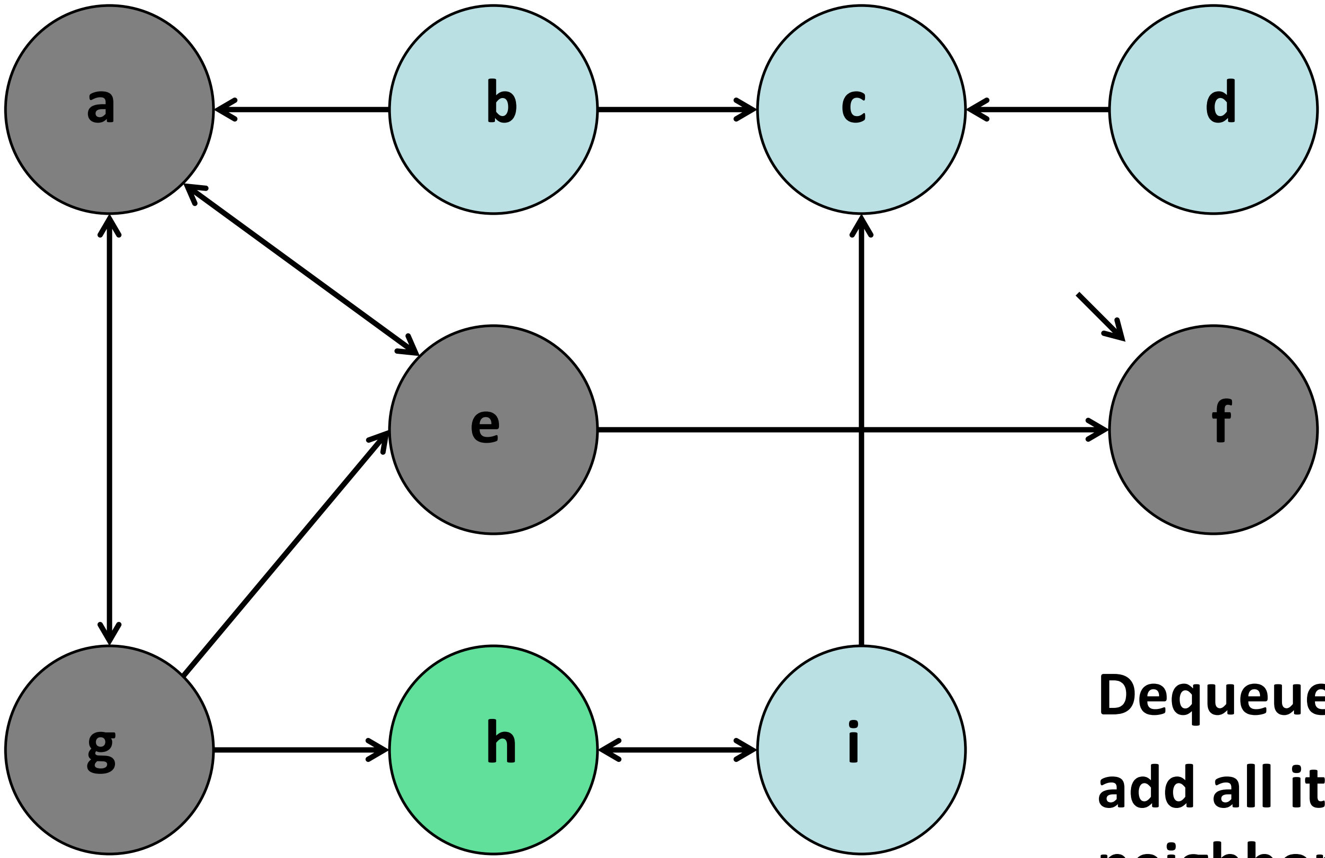


Dequeue a node

add all its unseen neighbors to the queue

queue: f, h

BFS

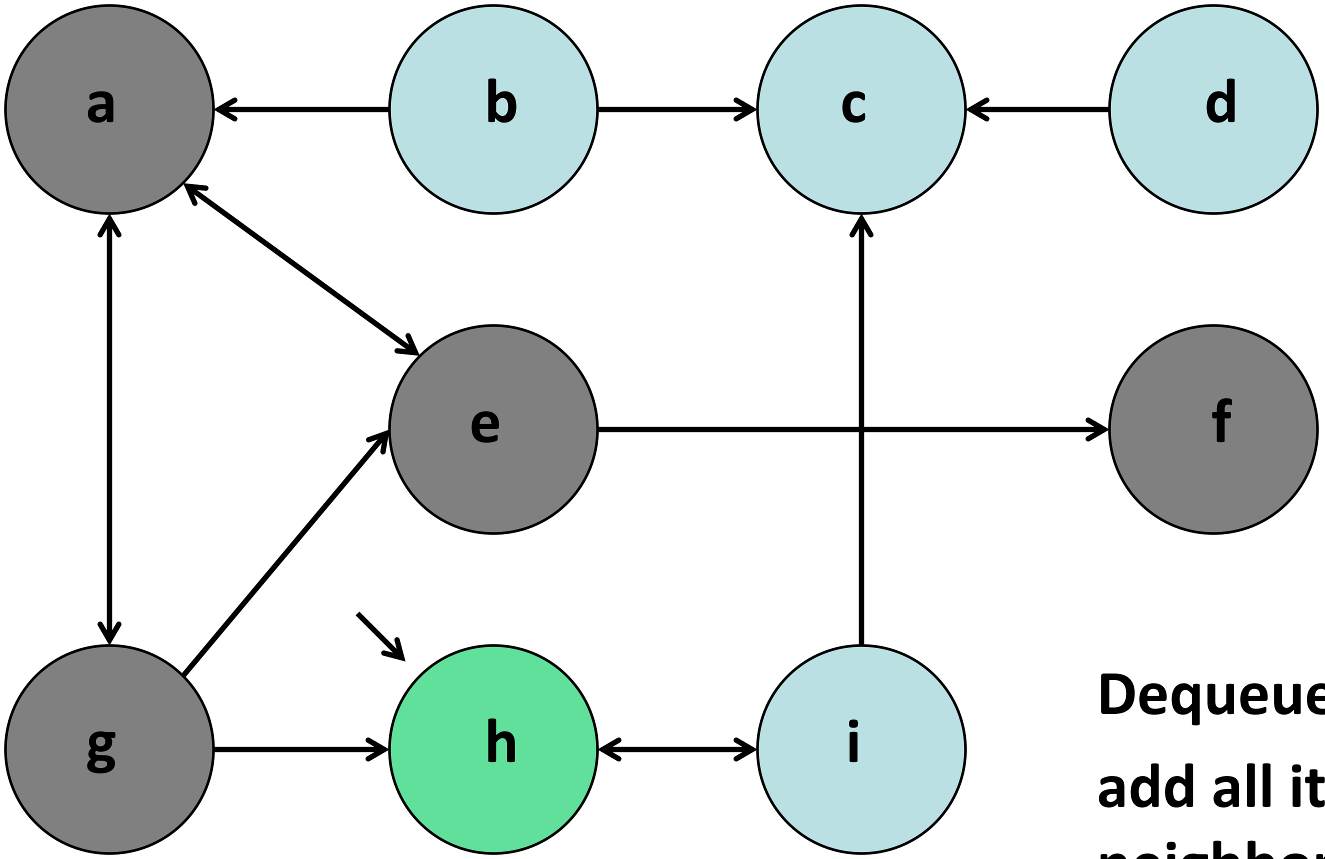


Dequeue a node

add all its unseen neighbors to the queue

queue: h

BFS

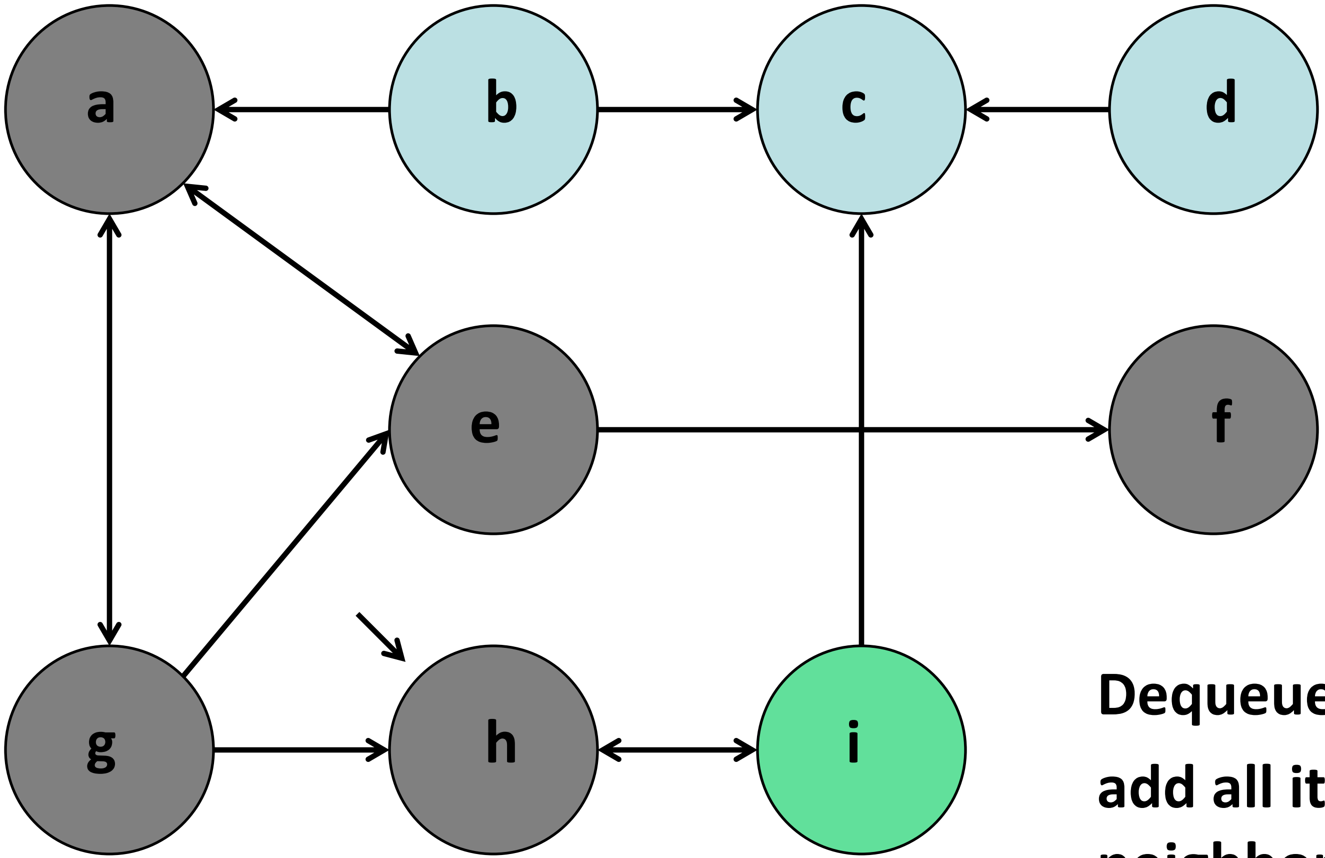


Dequeue a node

add all its unseen neighbors to the queue

queue: h

BFS

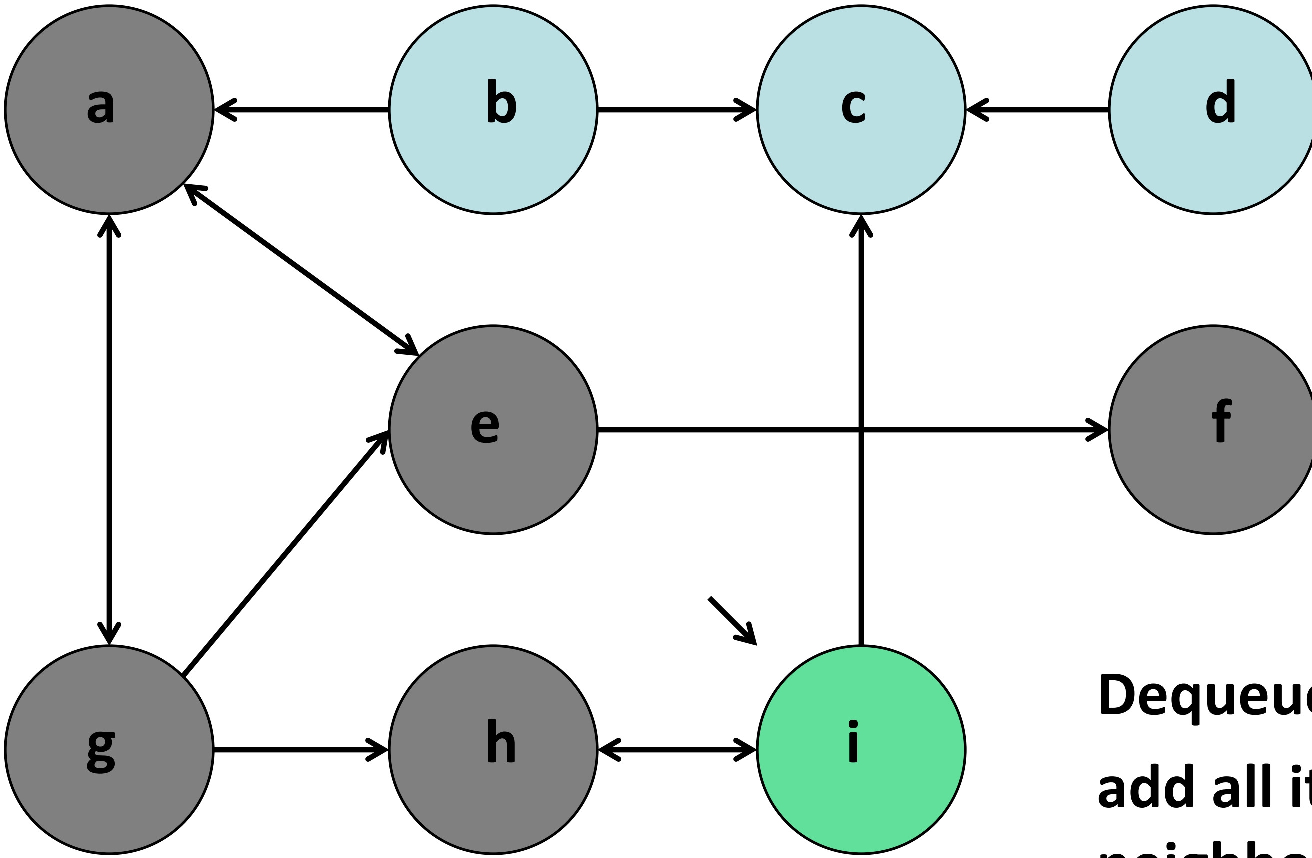


Dequeue a node

add all its unseen neighbors to the queue

queue: i

BFS

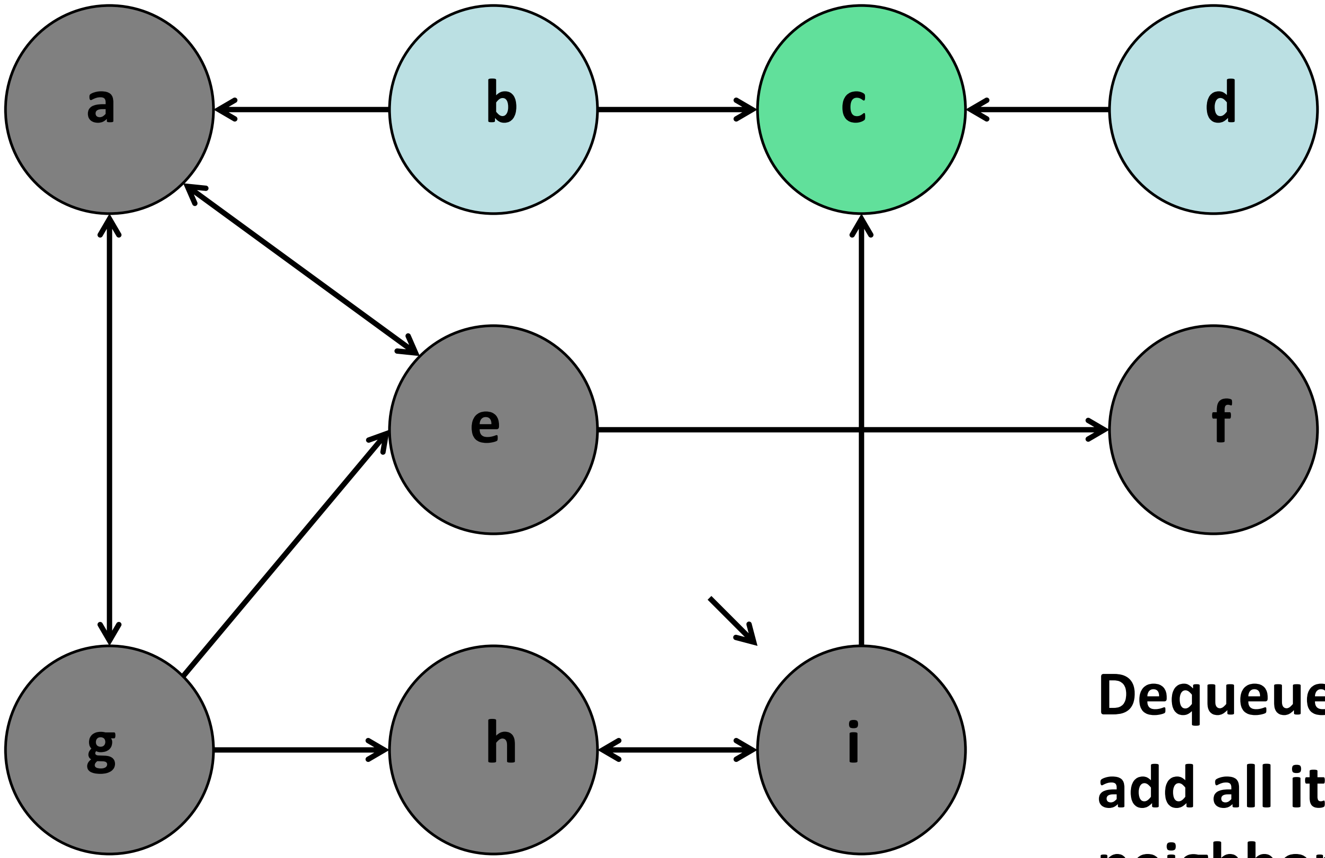


Dequeue a node

add all its unseen neighbors to the queue

queue: i

BFS

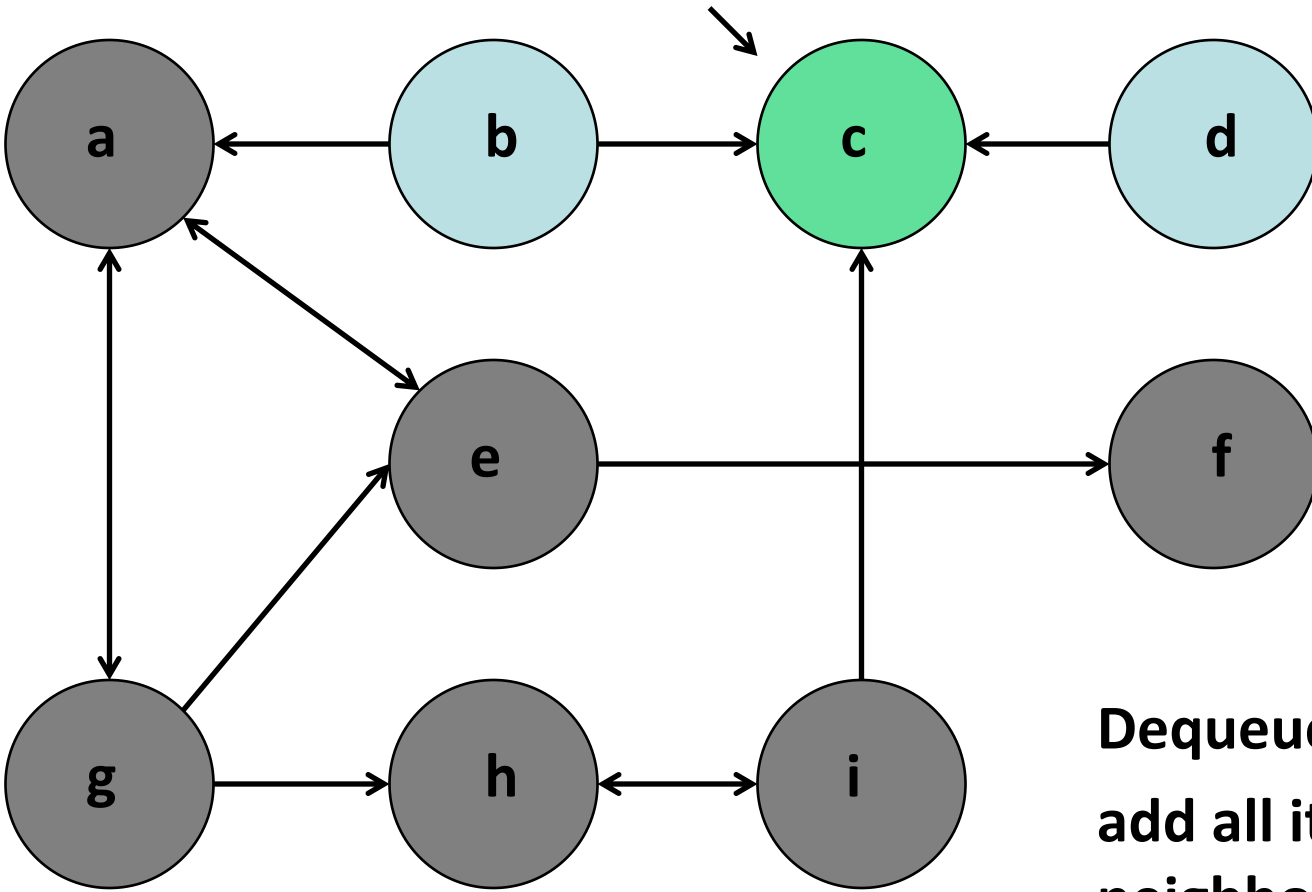


Dequeue a node

add all its unseen neighbors to the queue

queue: c

BFS

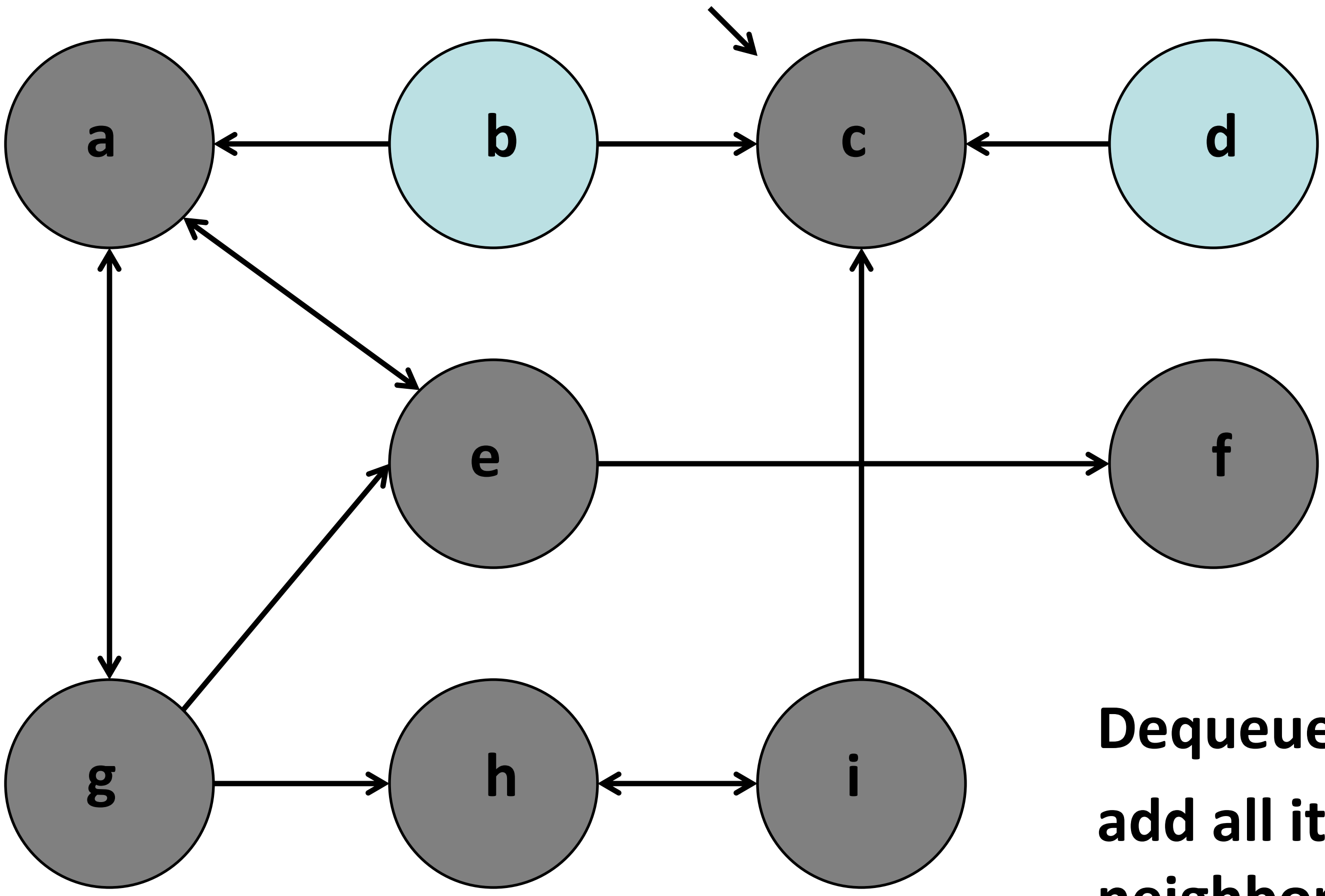


Dequeue a node

add all its unseen neighbors to the queue

queue: c

BFS



Dequeue a node

add all its unseen neighbors to the queue

queue: c

BFS Details

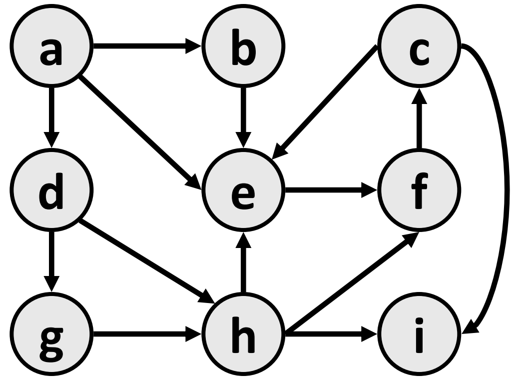
- In an n-node, m-edge graph, takes $O(m + n)$ time with an adjacency list
  - Visit each edge once, visit each node at most once

bfs from v_1 to v_2:
    create a *queue* of vertexes to visit,
        initially storing just v_1.
    mark v_1 as **visited**.

    while *queue* is not empty and v_2 is not seen:
        dequeue a vertex v from it,
        mark that vertex v as **visited**,
        and add each unvisited neighbor ***n*** of v to the *queue*.

- How could we modify the pseudocode to look for a specific path?

BFS observations



- *optimality*:
  - always finds the shortest path (fewest edges).
  - in unweighted graphs, finds optimal cost path.
  - In weighted graphs, *not* always optimal cost.
- *retrieval*: harder to reconstruct the actual sequence of vertices or edges in the path once you find it
  - conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a path array/list in progress
  - solution: We can keep track of the path by storing predecessors for each vertex (each vertex can store a reference to a *previous* vertex).
- DFS uses less memory than BFS, easier to reconstruct the path once found; but DFS does not always find shortest path. BFS does.

Recap

- **Recap**: Graphs
- **Practice**: Twitter Influence
- Depth-First Search (DFS)
- Announcements
- Breadth-First Search (BFS)

Next time: more graph searching algorithms

Overflow

BFS that finds path

bfs from v_1 to v_2:
create a *queue* of vertexes to visit,
initially storing just v_1.
mark v_1 as **visited**.

while *queue* is not empty and v_2 is not seen:
dequeue a vertex *v* from it,
mark that vertex *v* as **visited**,
and add each unvisited neighbor ***n*** of *v* to the *queue,*
while setting *n*'s previous to *v*.