EEL 4837
Programming for Electrical Engineers II

Ivan Ruchkin
Assistant Professor
Department of Electrical and Computer Engineering
University of Florida at Gainesville
iruchkin@ece.ufl.edu
http://ivan.ece.ufl.edu

Graphs

Readings:

- Weiss 9.1–9.3.1
- Horowitz 6.2
- Cormen 22

Graphs

A graph $G = (V,E)$ is composed of

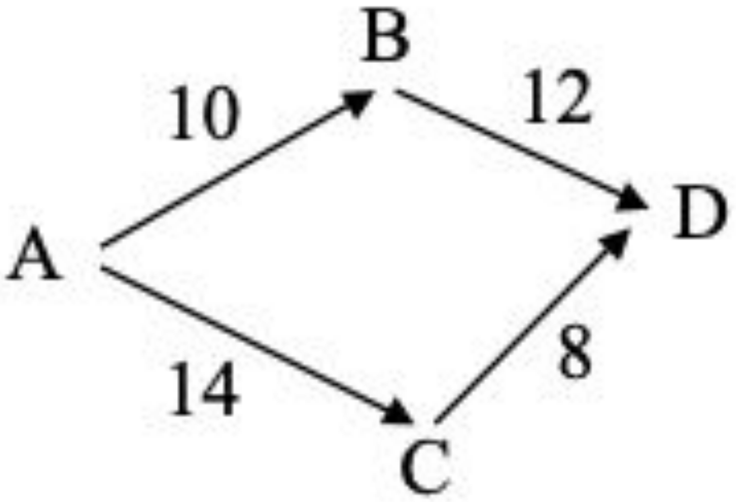
- a set of vertices V
- a set of *edges* $E \subset V \times V$ connecting the vertices

An edge $e = (u,v)$ is a pair of vertices

Weighted and Unweighted Graphs

Graphs can also be

- unweighted (as in the previous examples)
- weighted (edges have weights)



weighted graph

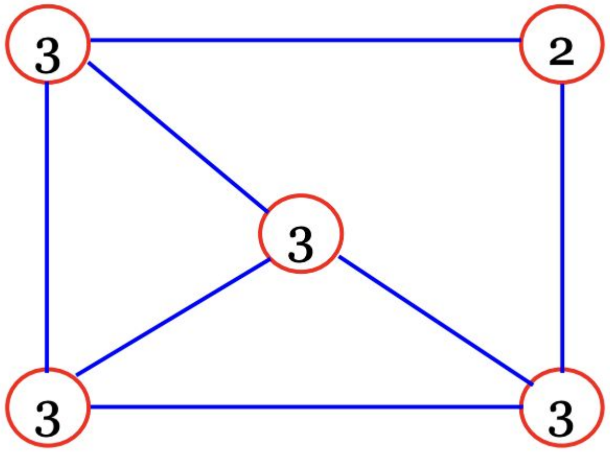
Graph Applications

- Electronic Circuits
- Transportation and Communication Networks
- Process flow charts
- Tasks in a project
  - Some should be completed before others, so edges represent task dependencies
- Any sort of relationships
  - Between people, programs, processes, concepts

Graph Terminology

A vertex v is adjacent to vertex u iff $(u,v) \in E$

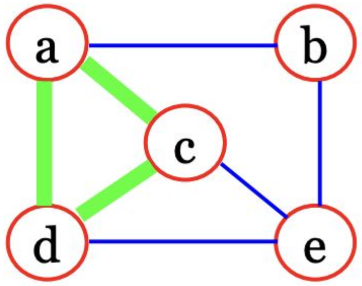
The degree of a vertex: # of adjacent vertices



Graph Terminology

Simple path – a path with no repeated vertices

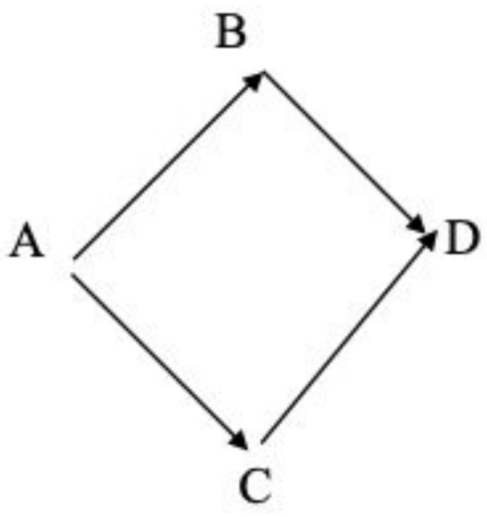
Cycle – a simple path, except that the last vertex is the same as the first vertex



Representation: Adjacency Matrix

Matrix M with entries for all pairs of vertices

- M[i][j] = 1 $\Leftrightarrow$ there is an edge (i, j)
- M[i][j] = 0 $\Leftrightarrow$ there is no edge (i, j)



A	B	C	D
0	1	2	3

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	0	0	1
3	0	0	0	0

Weighted Graphs

For weighted graphs, place weights in matrix (if there is no edge we use a value which can't be confused with a weight, e.g., -1 or `Integer.MAX_VALUE`)

Deficiencies of Adjacency Matrix

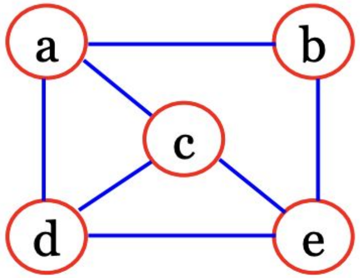
- Sparse graphs with few edges for number of vertices result in many zero entries in adjacency matrix—this wastes space and makes many algorithms less efficient (e.g., to find nodes adjacent to a given node, we have to iterate through the whole row even if there are few 1s there).

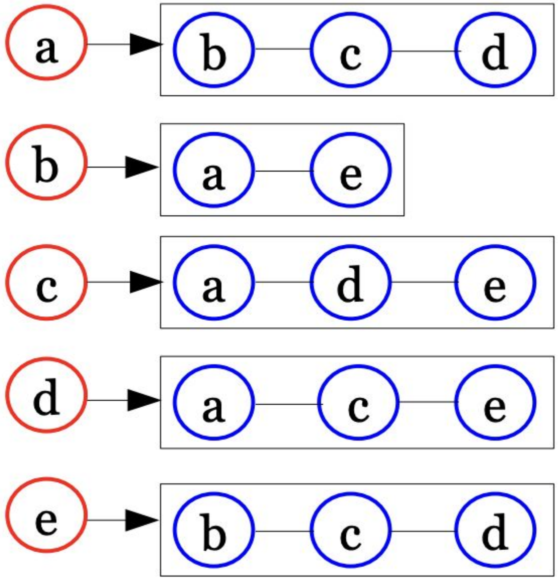
- Also, if the number of nodes in the graph may change, matrix representation is too inflexible (especially if we don't know the maximal size of the graph).

- Also, an *array of node indices* requires looking through the array each time to find the node's position in the adjacency matrix
- We will fix this later with a better data structure for looking up nodes *(hash table)*

Adjacency List Representation

- The adjacency list of a vertex *v*:
  sequence of vertices adjacent to *v*
- A graph is represented by the adjacency lists of all its vertices





- o Space required for adjacency matrix for a graph with m vertices and n edges: $O(m^2)$
- o Space required for an adjacency list for a graph with m vertices and n edges: $O(m+n)$

Traversing a Graph

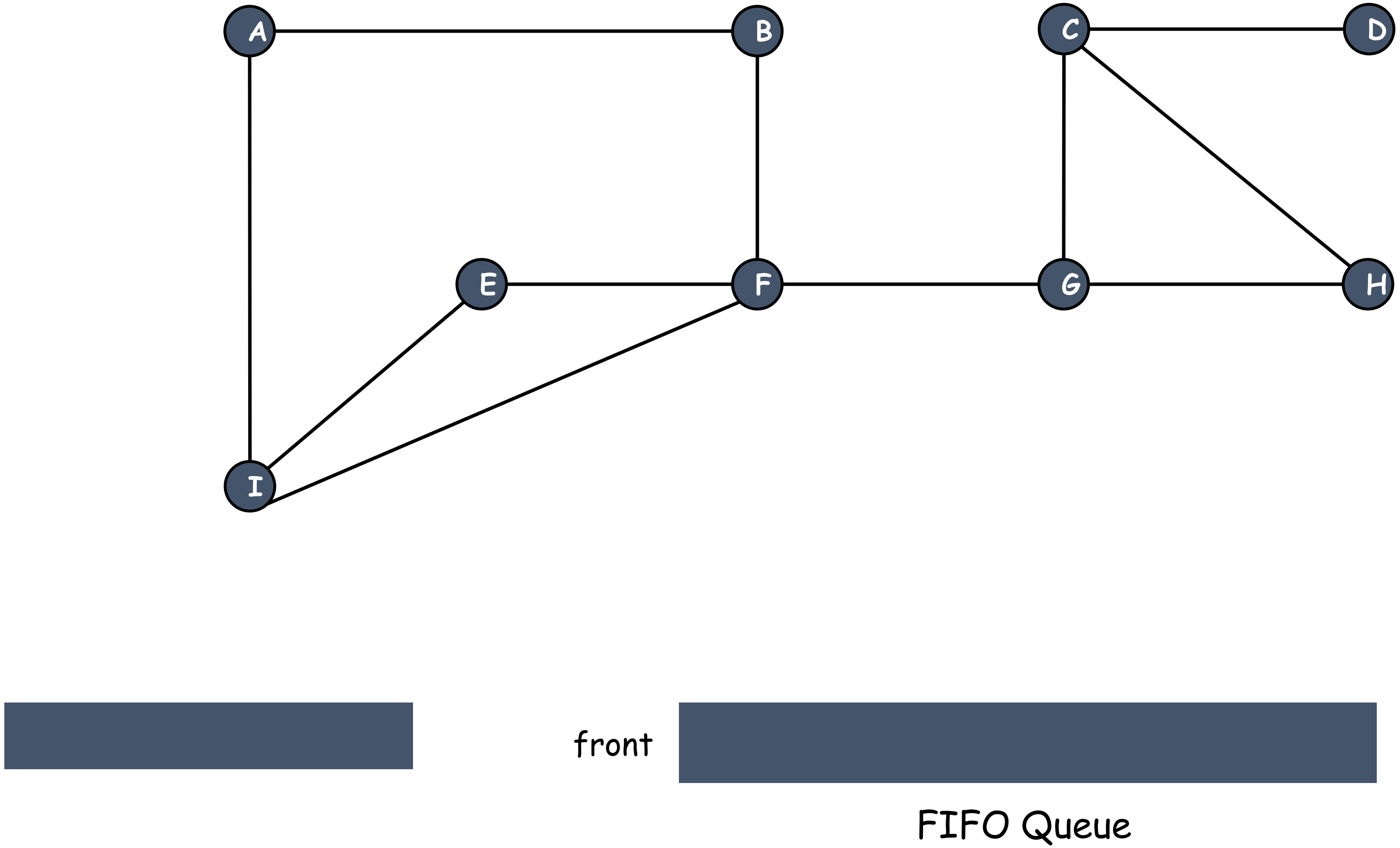
Given a graph G = (V, E), write an algorithm to **traverse** (i.e., visit) **each node** of the G

Issue: getting tangled up in cycles

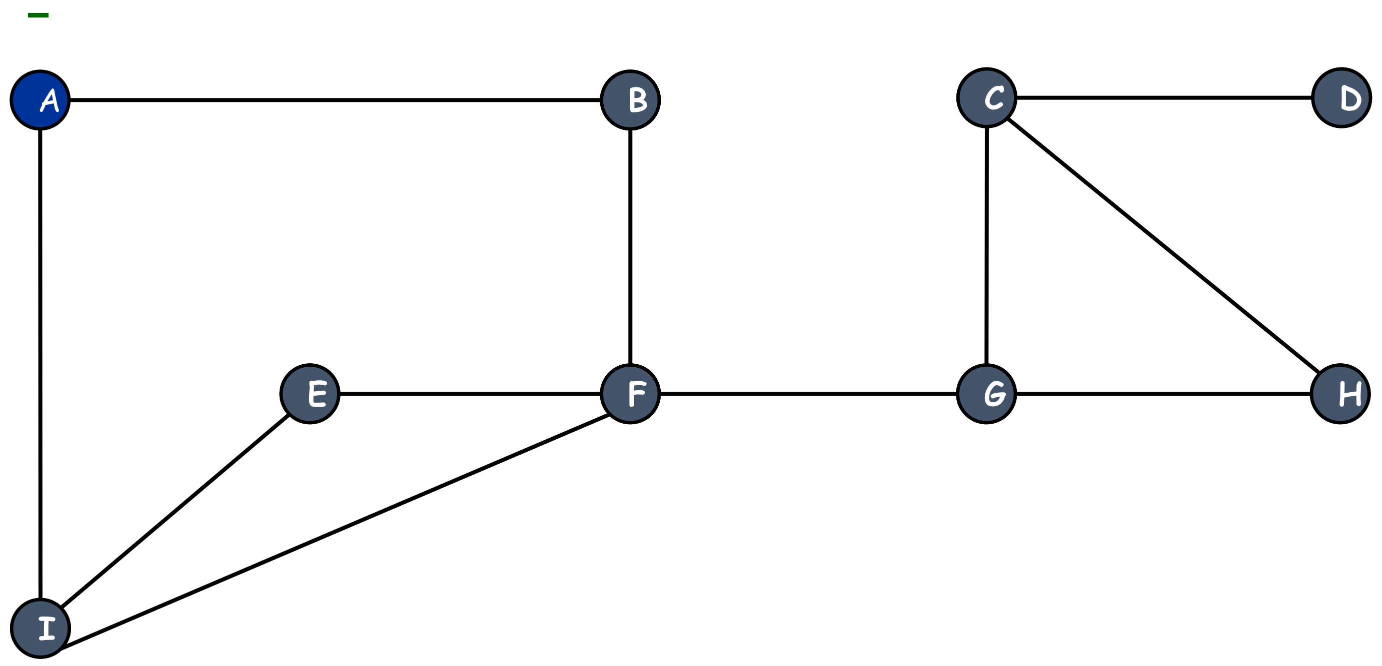
Breadth-First Search (BFS)

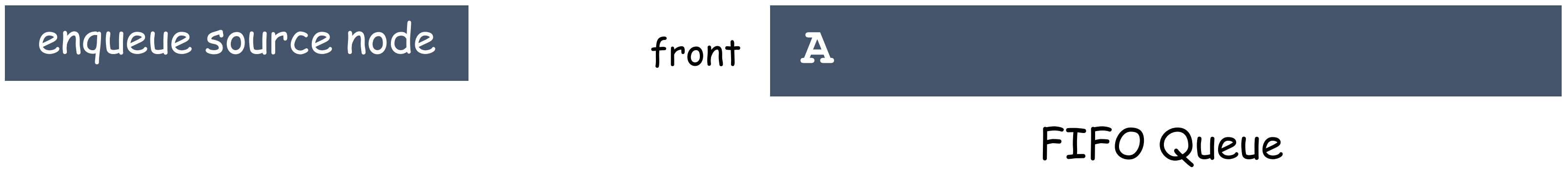
```
Create a queue Q
Mark initial node v as visited and enqueue v in Q
While Q is non-empty
   Dequeue u from Q
   For each unvisited neighbor n of u:
      Mark n as visited
      Enqueue n into Q
```

Breadth-First Search

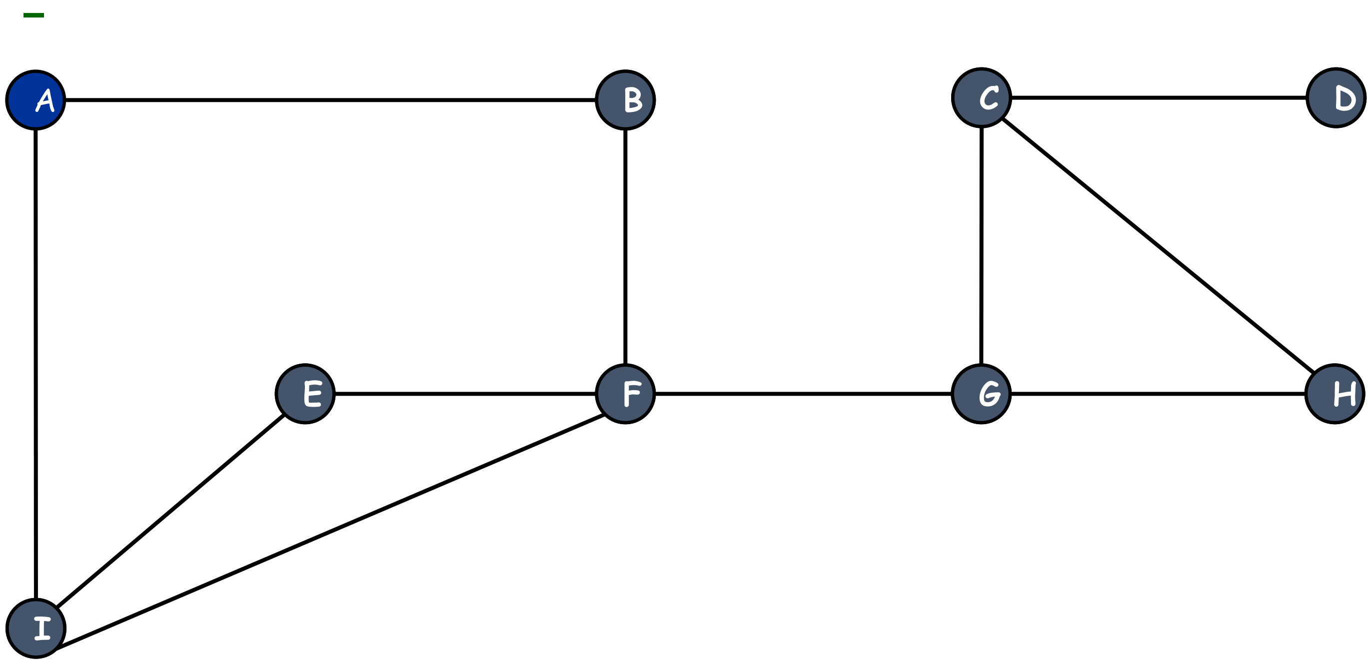


Breadth-First Search





Breadth-First Search

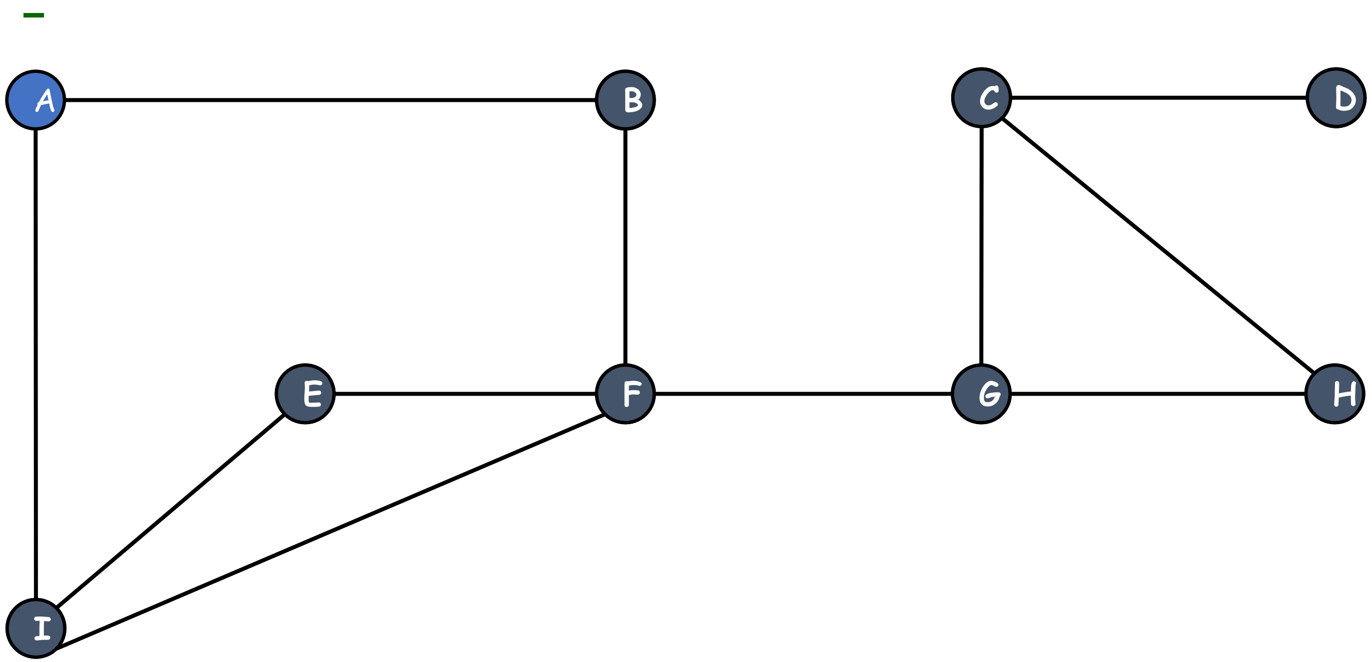


dequeue next node

front A

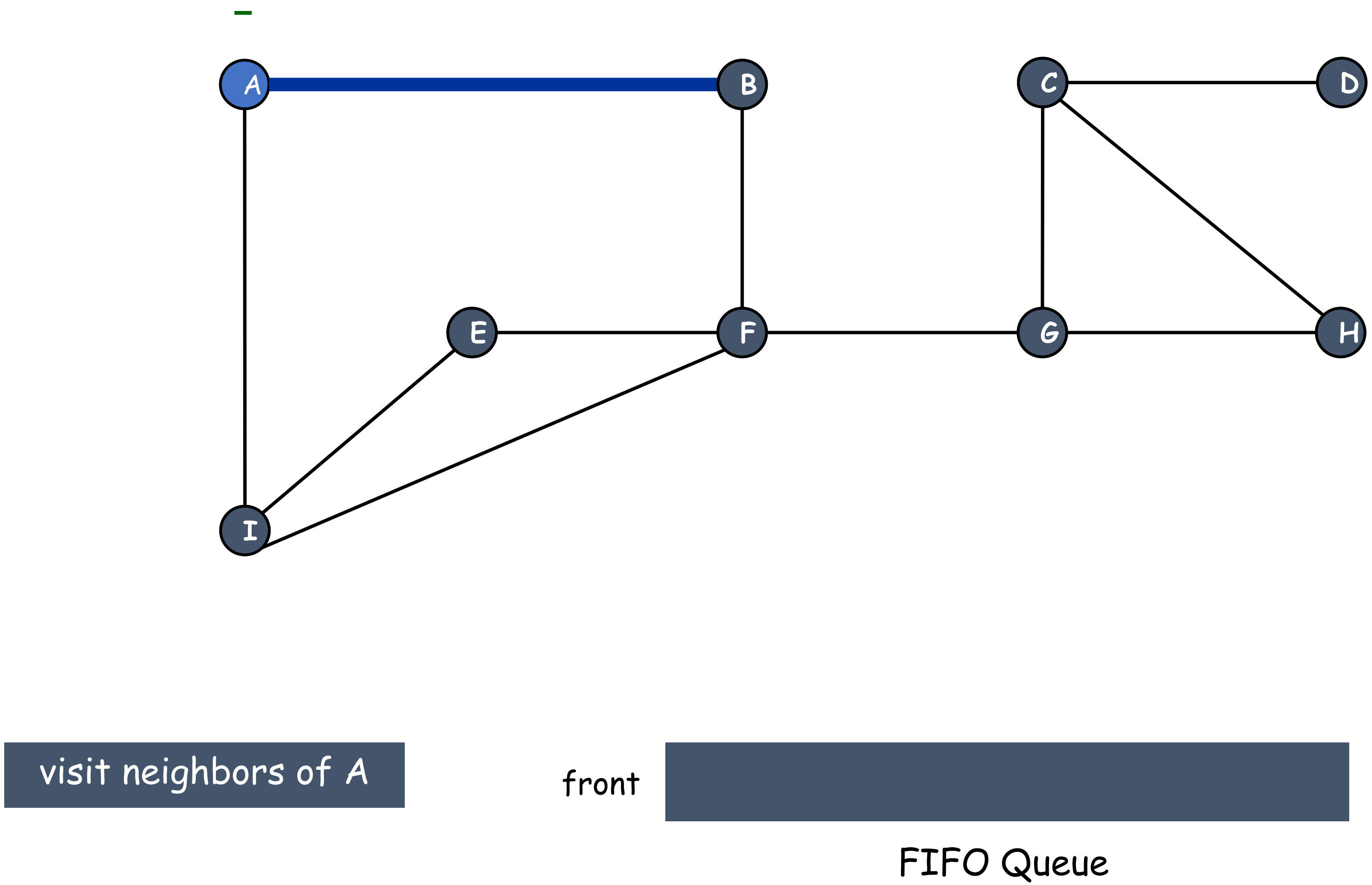
FIFO Queue

Breadth-First Search

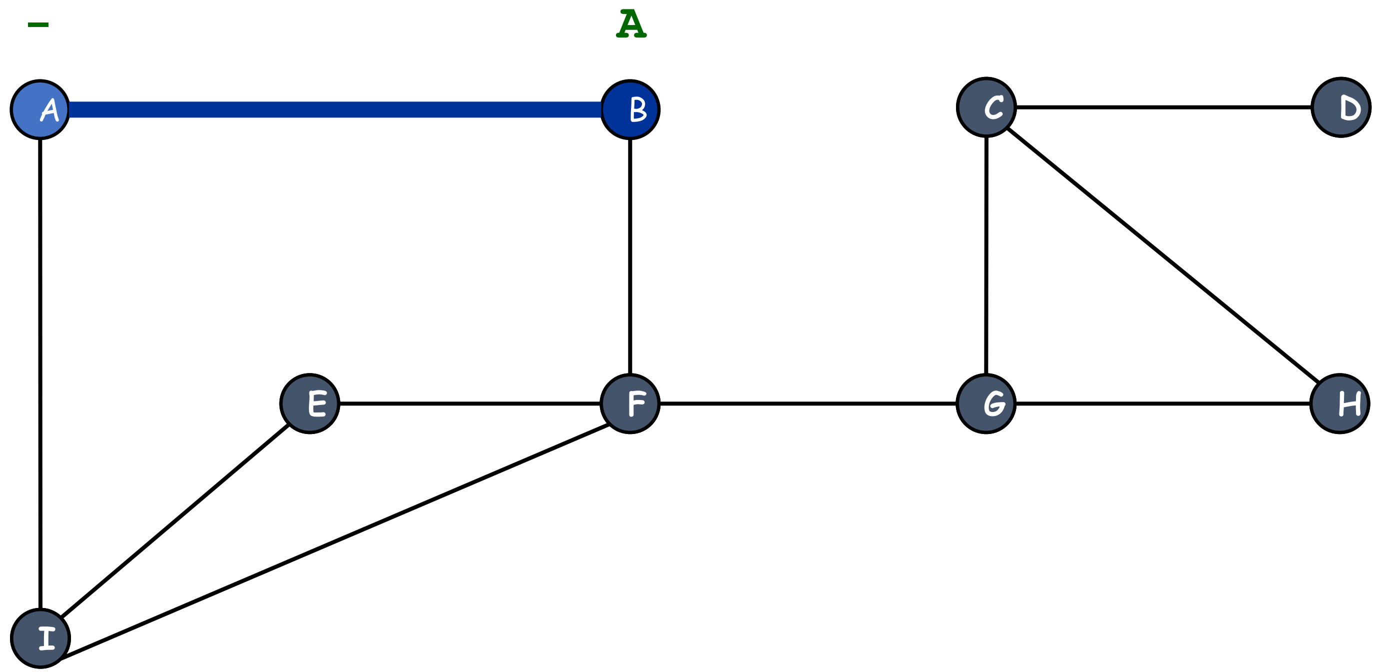


visit neighbors of A

Breadth-First Search



Breadth-First Search

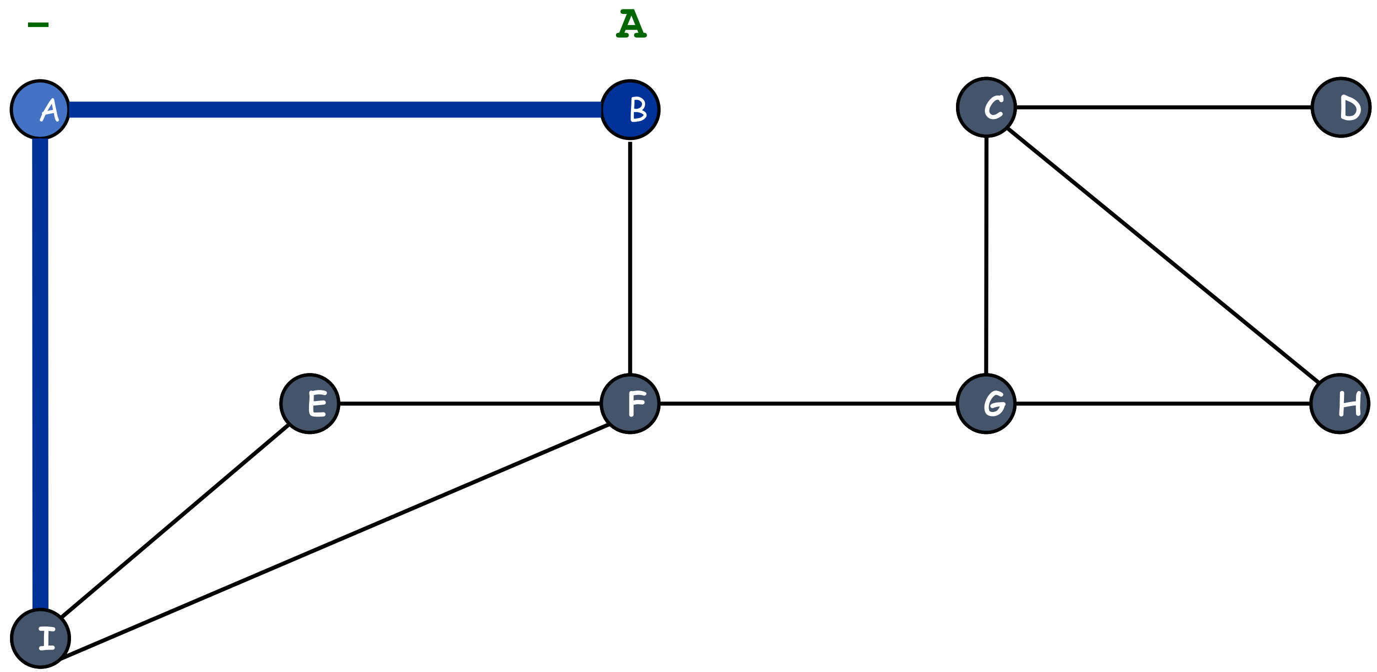


B discovered

front B

FIFO Queue

Breadth-First Search

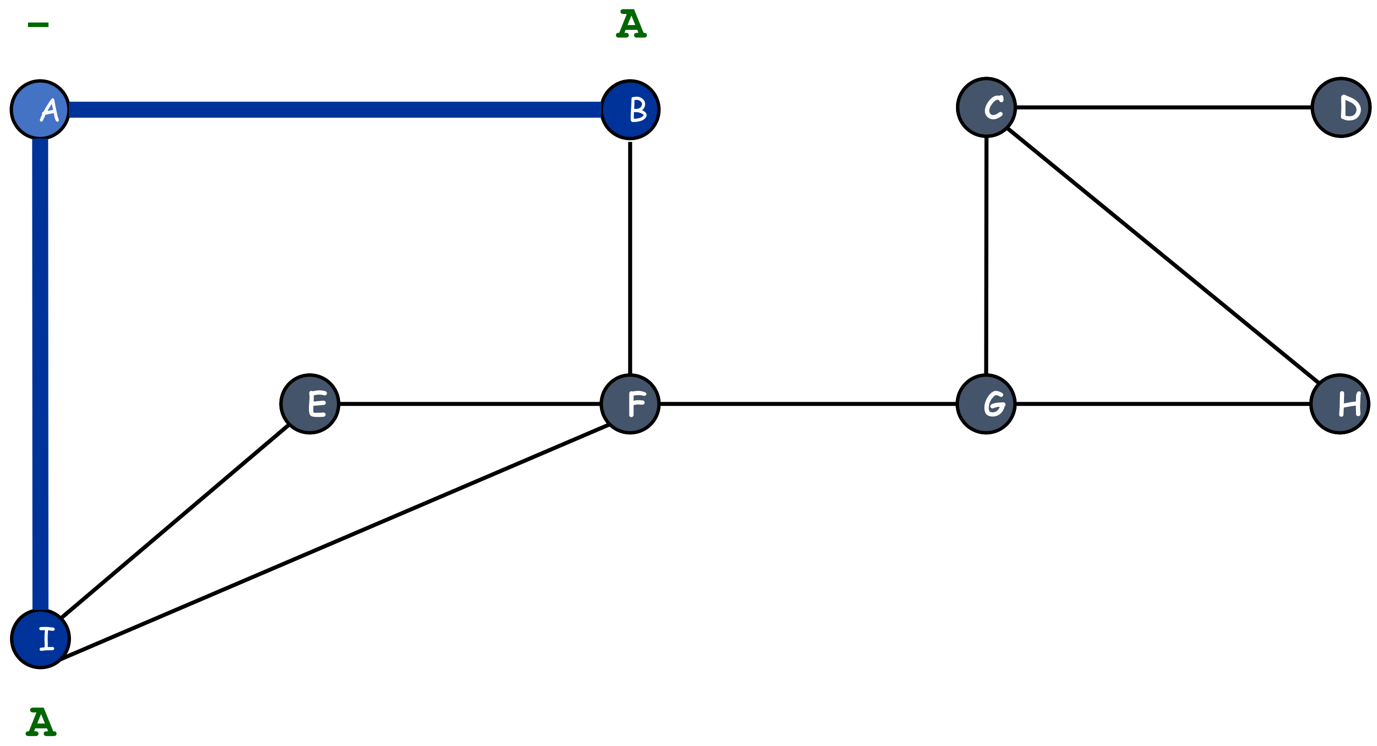


visit neighbors of A

front B

FIFO Queue

Breadth-First Search

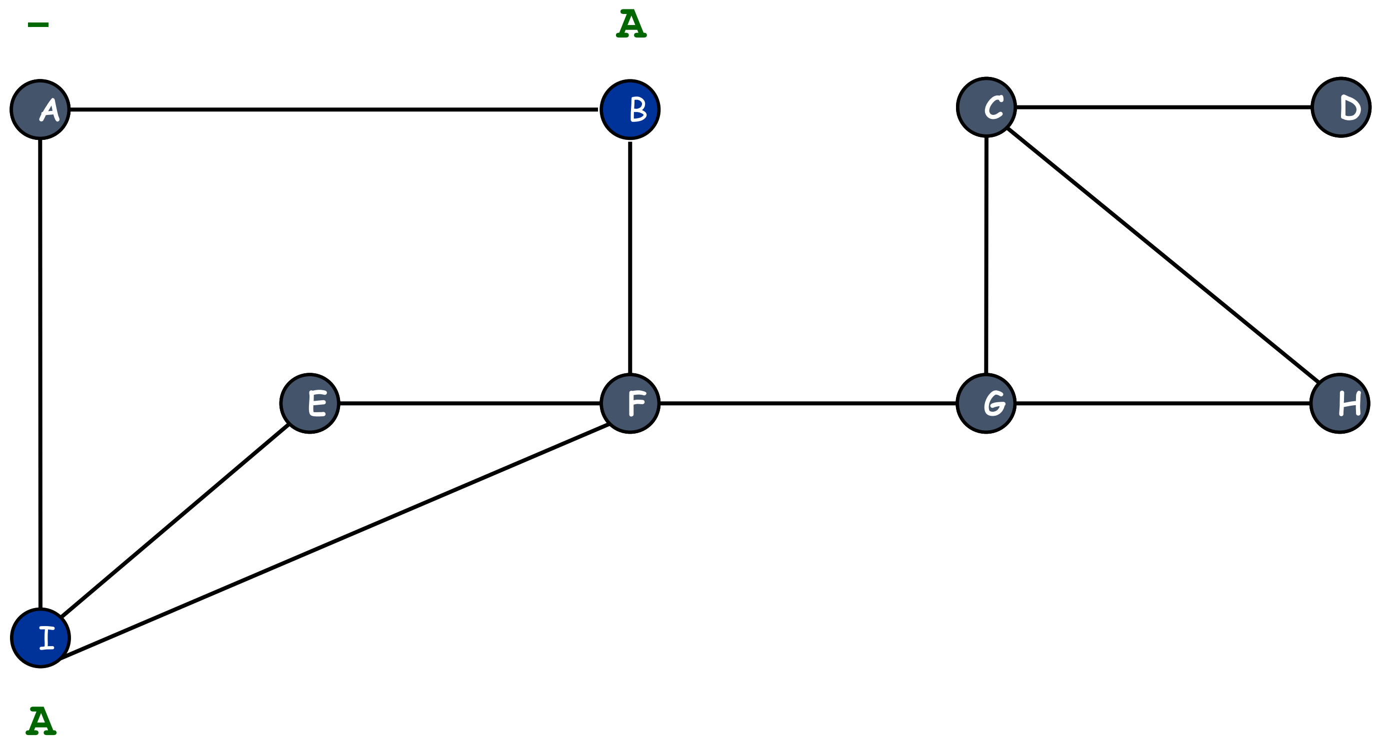


I discovered

front B I

FIFO Queue

Breadth-First Search

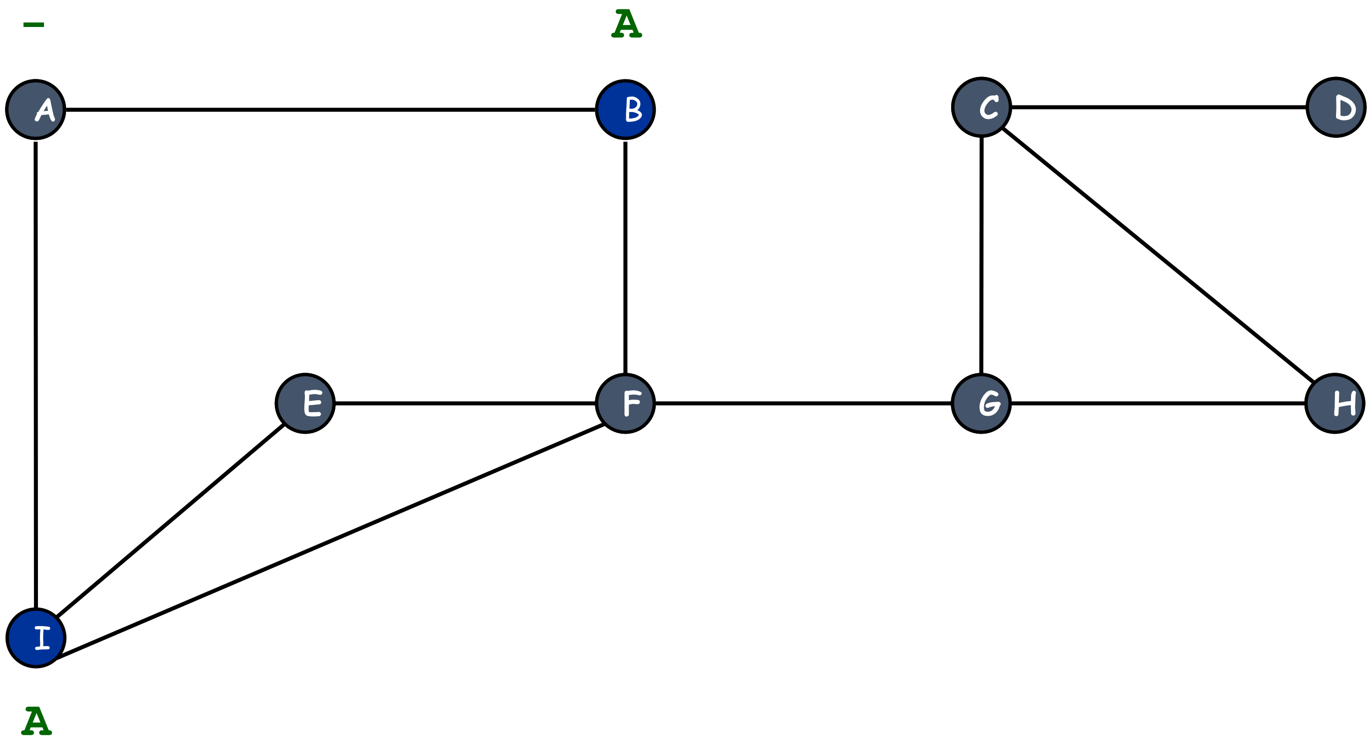


finished with A

front | B I

FIFO Queue

Breadth-First Search

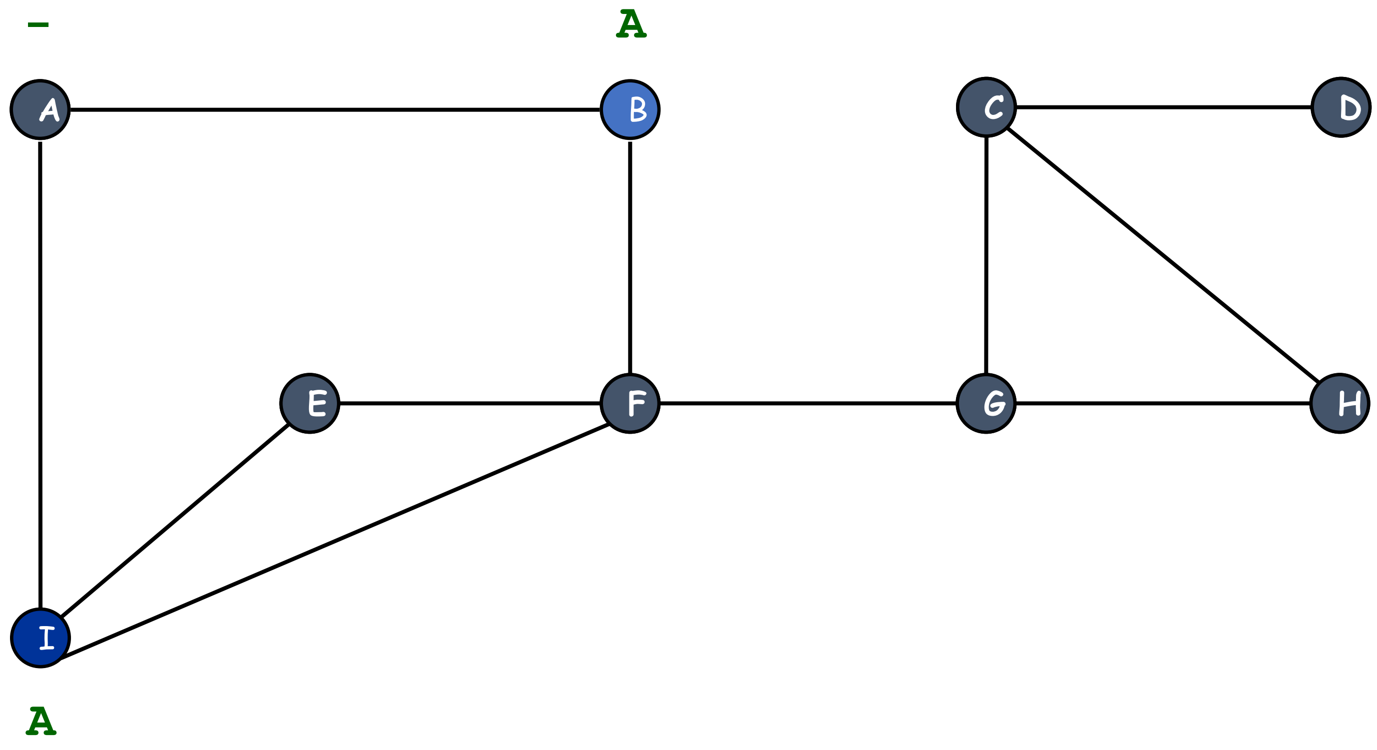


dequeue next vertex

front B I

FIFO Queue

Breadth-First Search

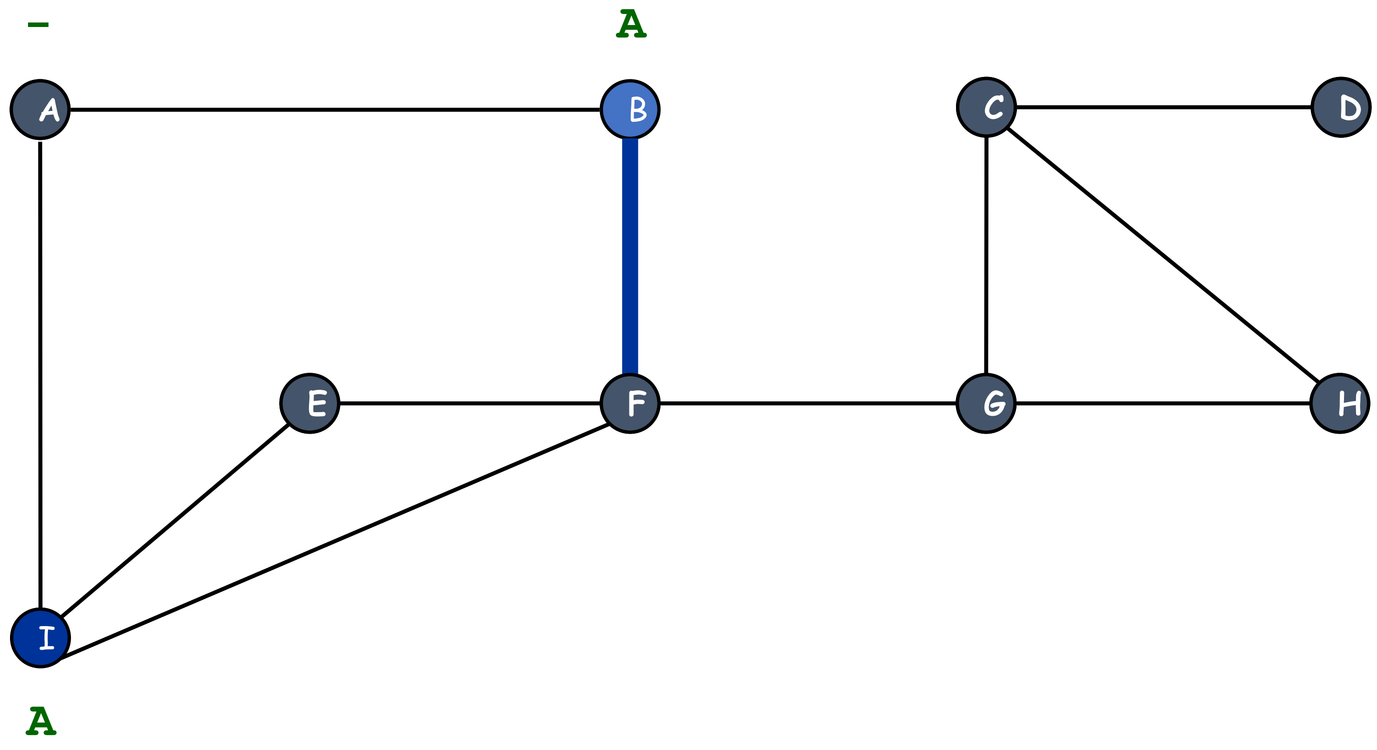


visit neighbors of B

front I

FIFO Queue

Breadth-First Search

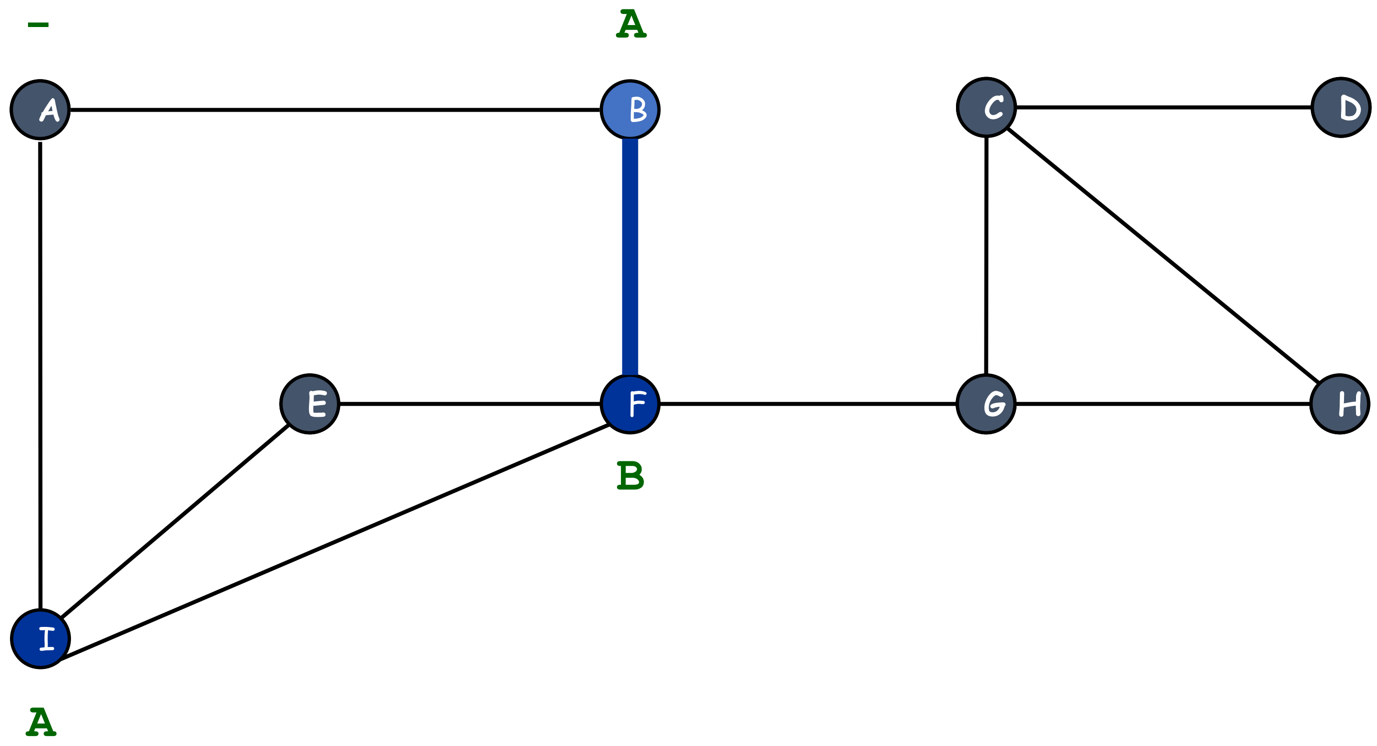


visit neighbors of B

front I

FIFO Queue

Breadth-First Search

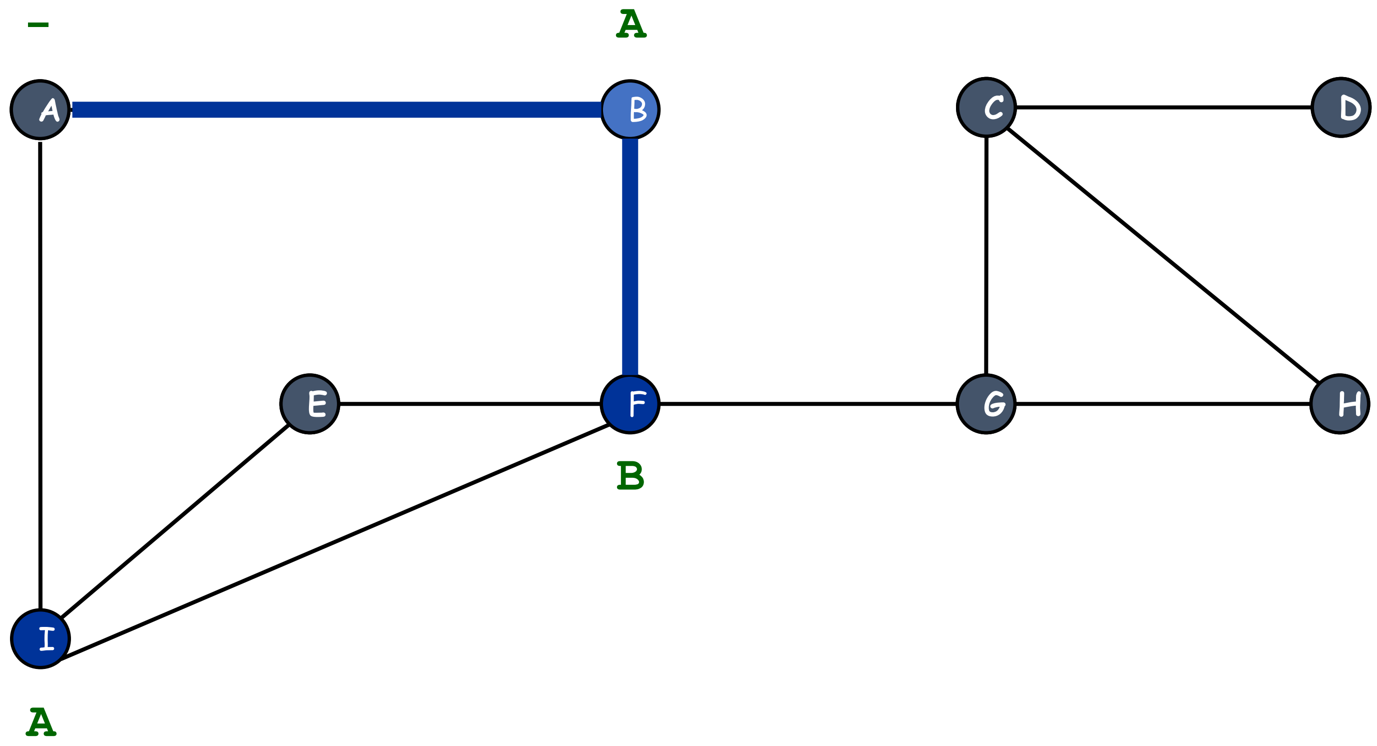


F discovered

front | I F

FIFO Queue

Breadth-First Search

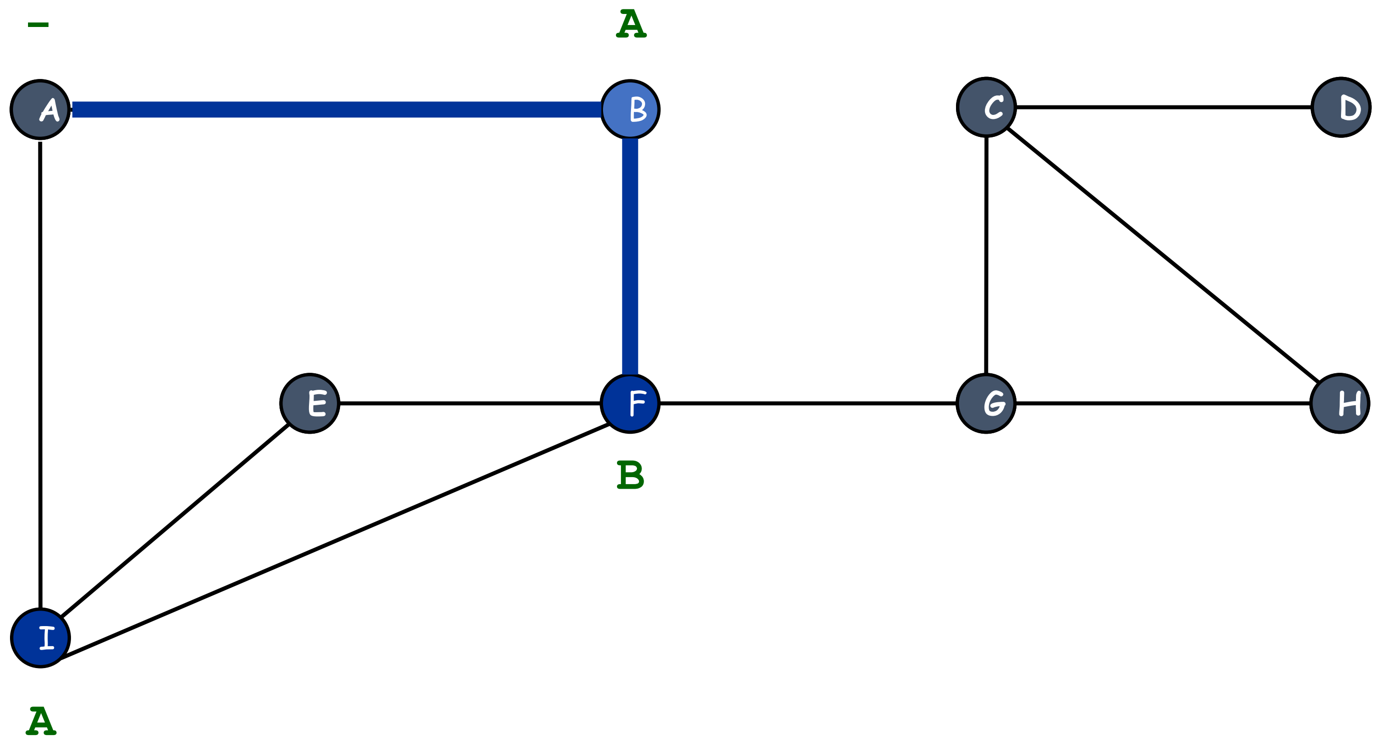


visit neighbors of B

front I F

FIFO Queue

Breadth-First Search

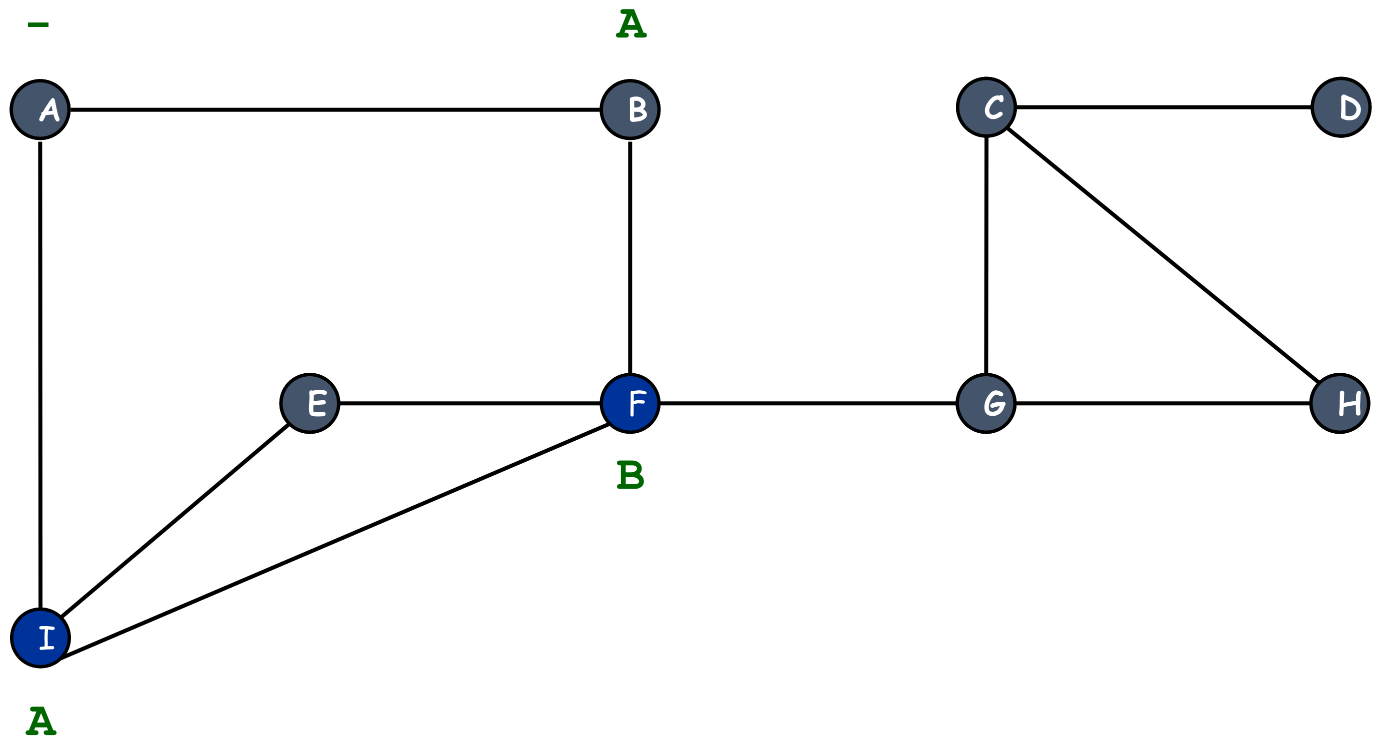


A already discovered

front I F

FIFO Queue

Breadth-First Search

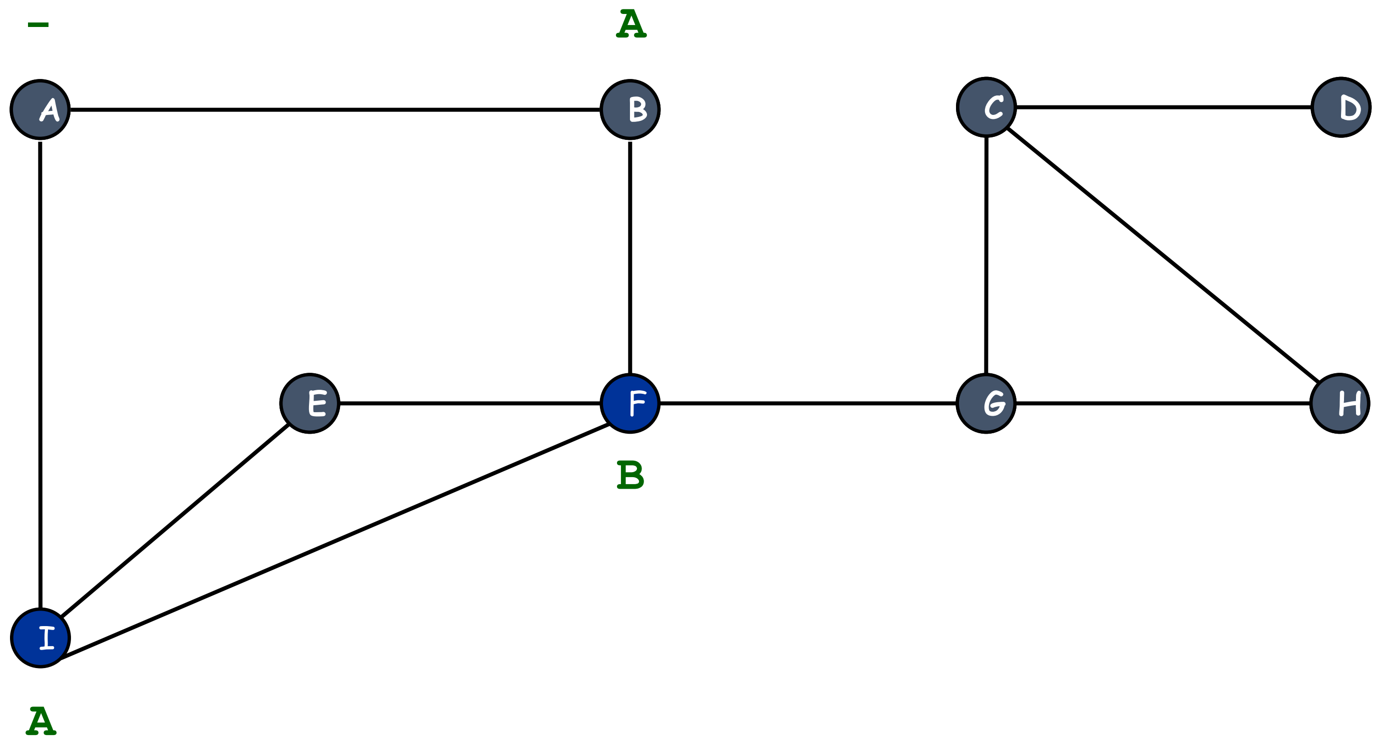


finished with B

front I F

FIFO Queue

Breadth-First Search

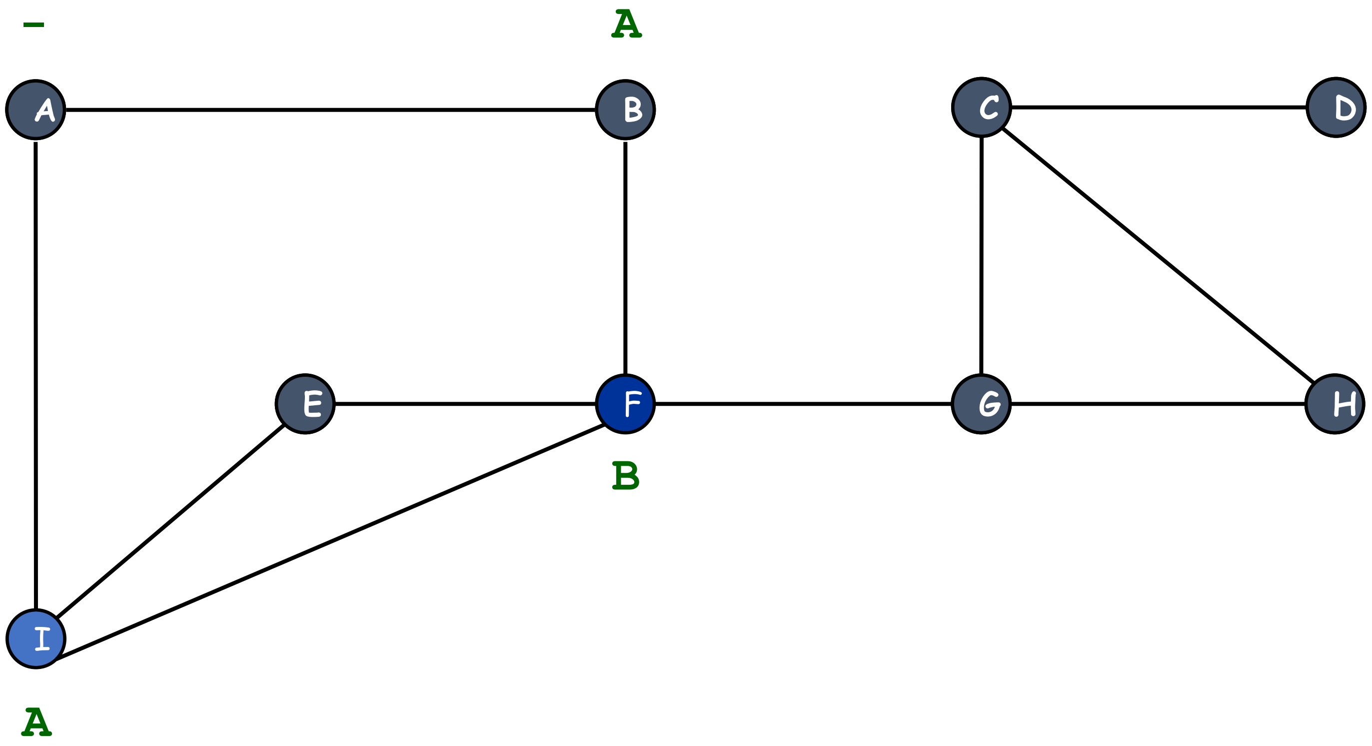


dequeue next vertex

front I F

FIFO Queue

Breadth-First Search

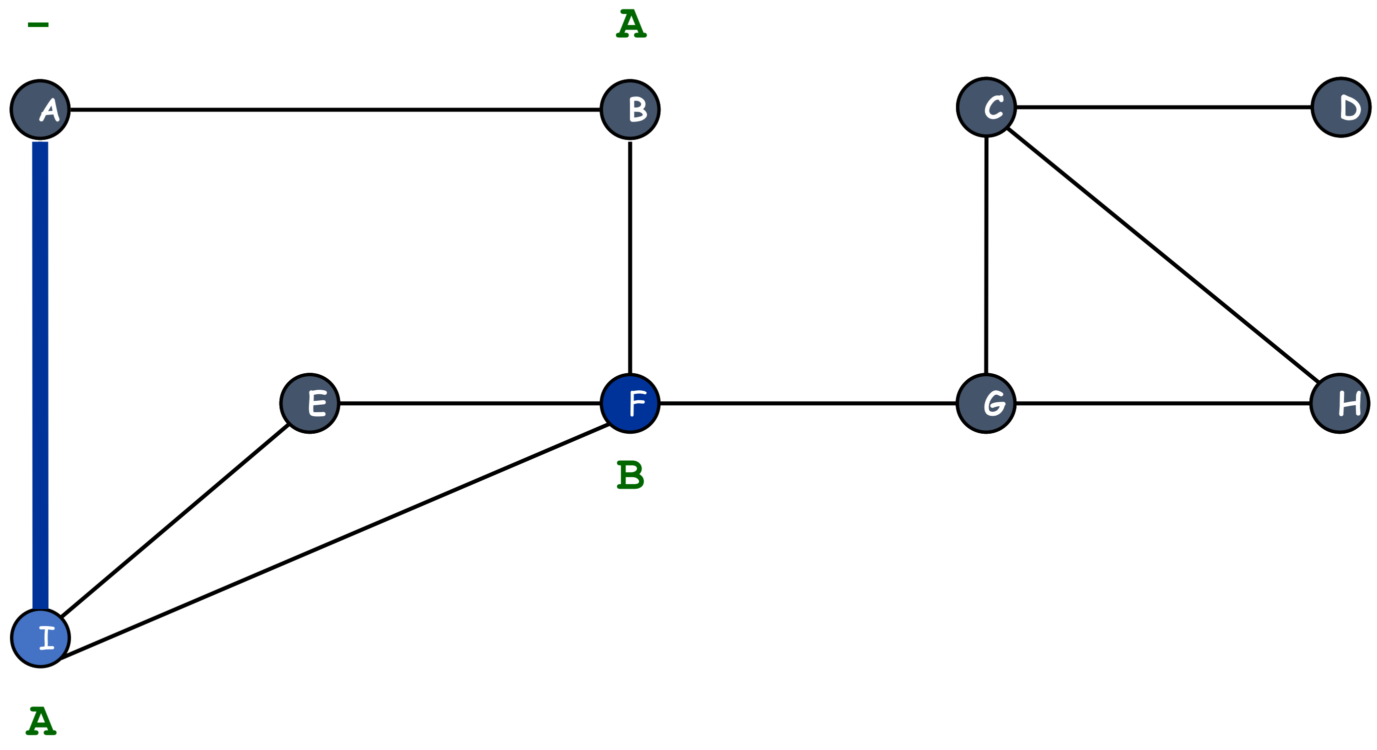


visit neighbors of I

front **F**

FIFO Queue

Breadth-First Search

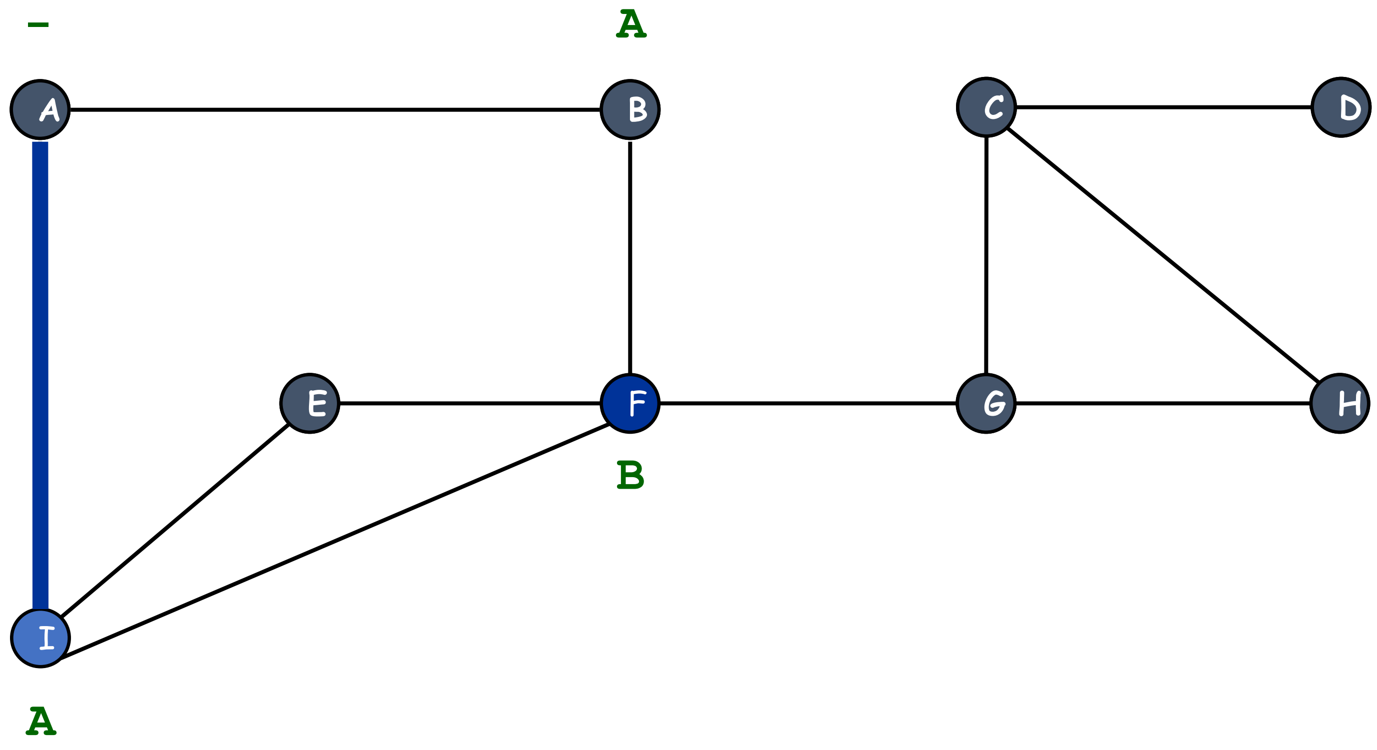


visit neighbors of I

front F

FIFO Queue

Breadth-First Search

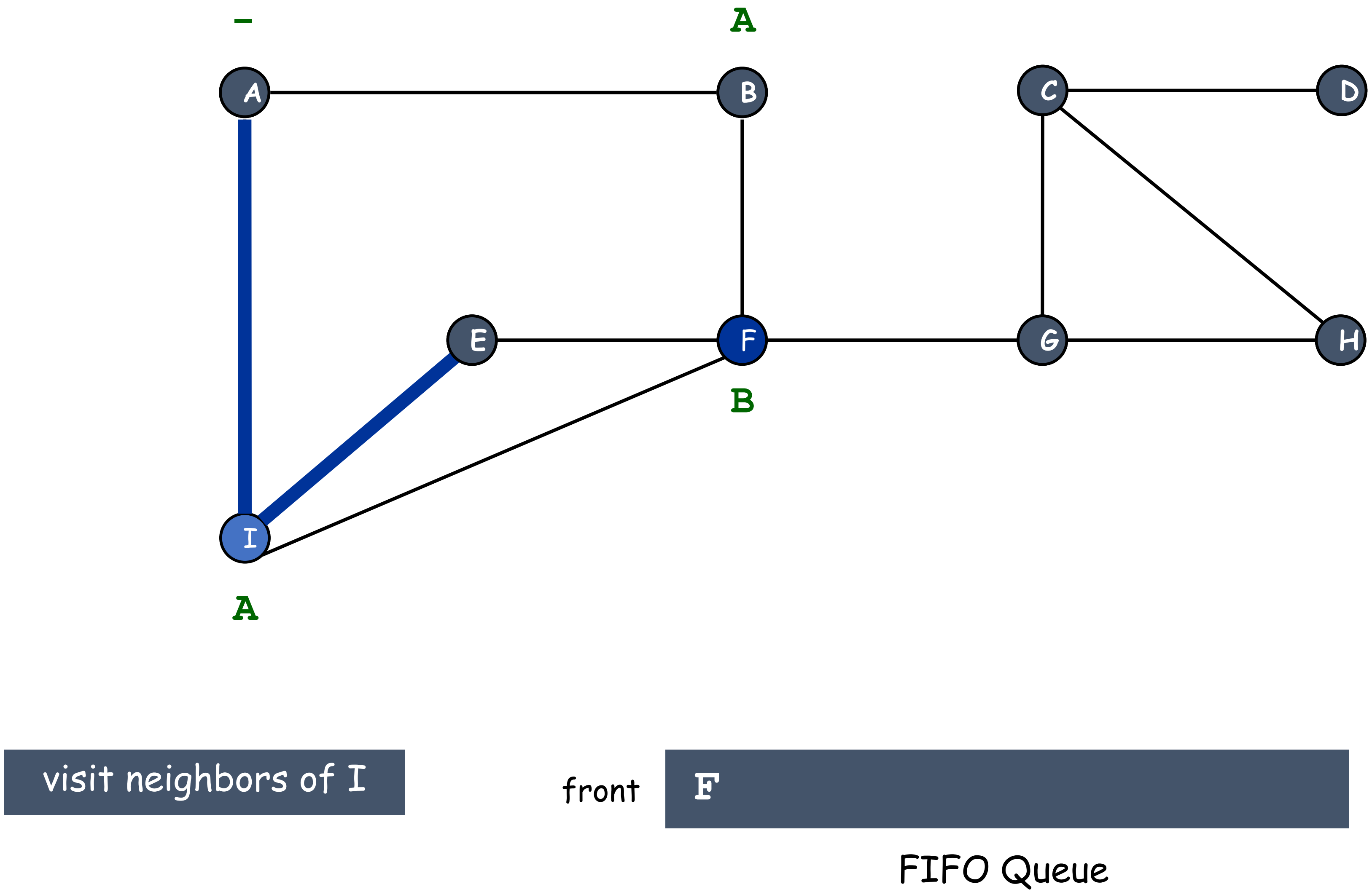


A already discovered

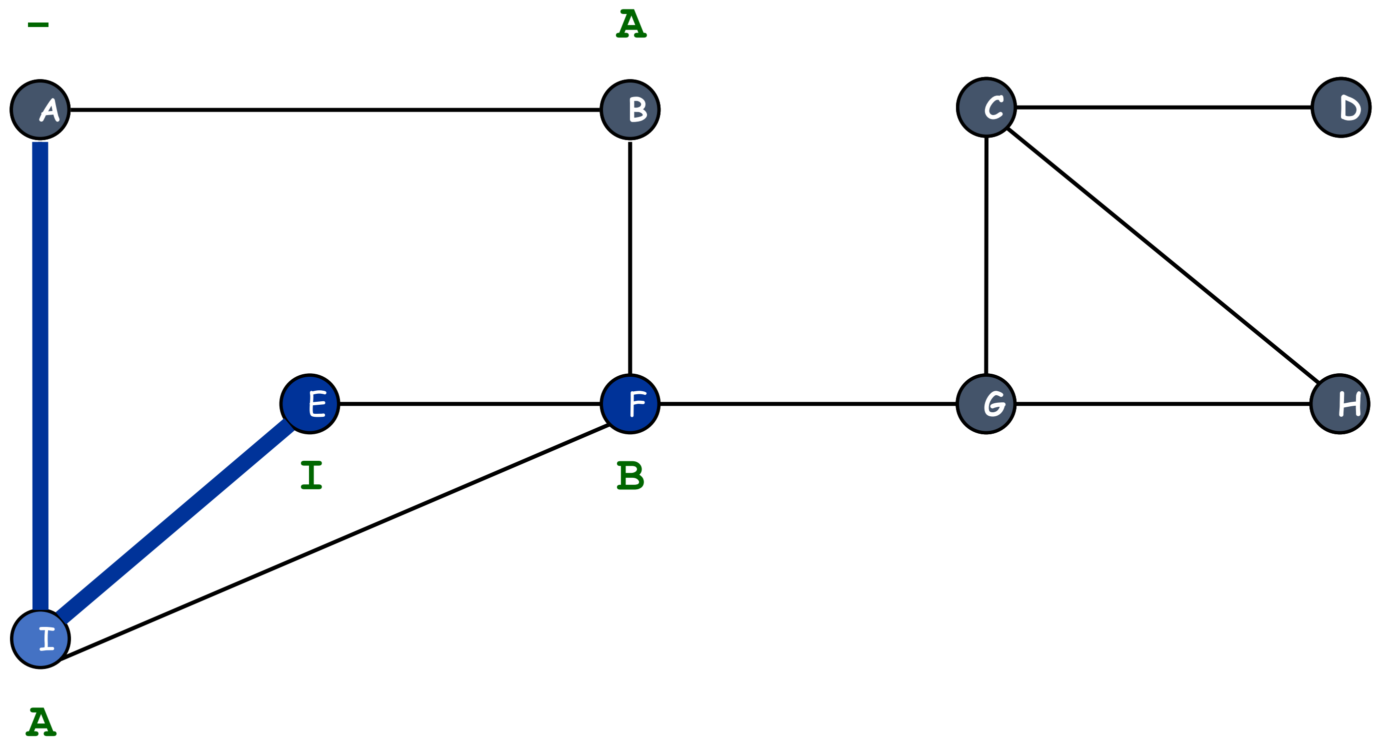
front F

FIFO Queue

Breadth-First Search



Breadth-First Search

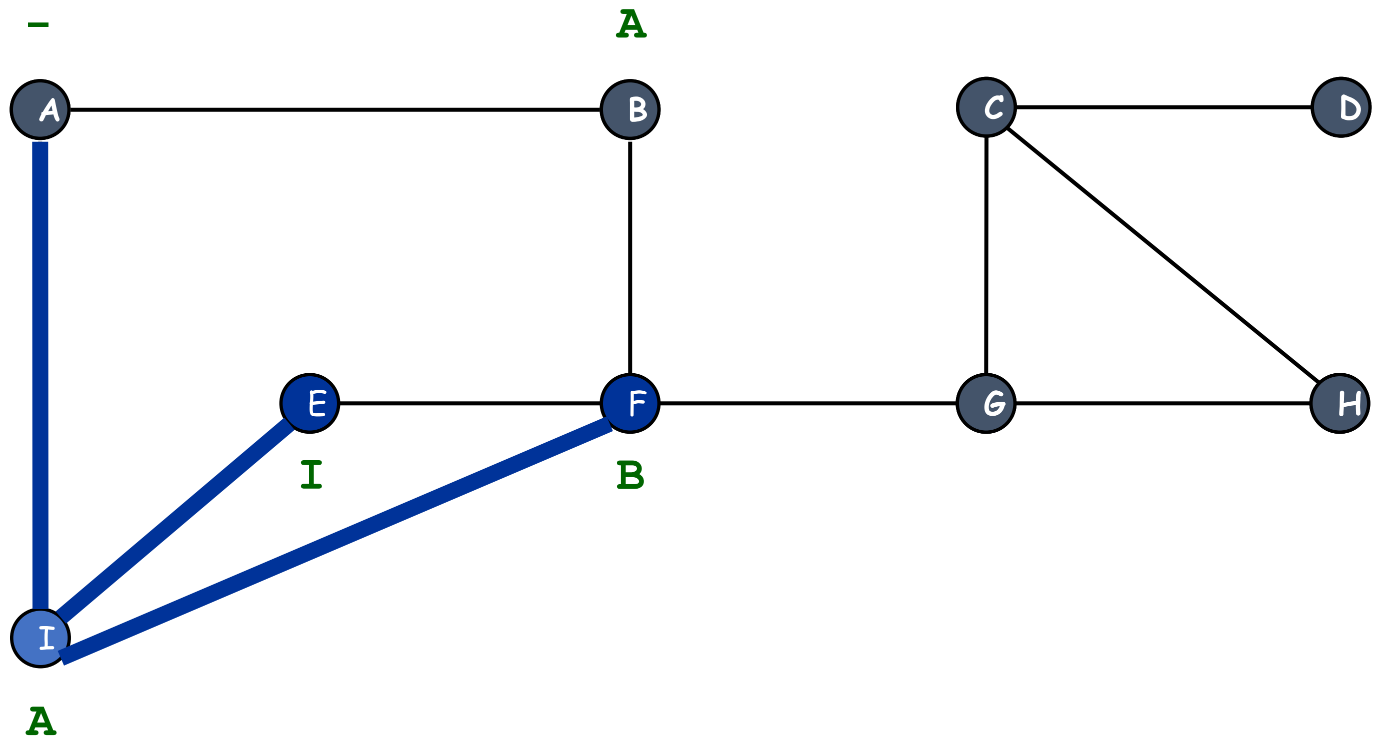


E discovered

front F E

FIFO Queue

Breadth-First Search

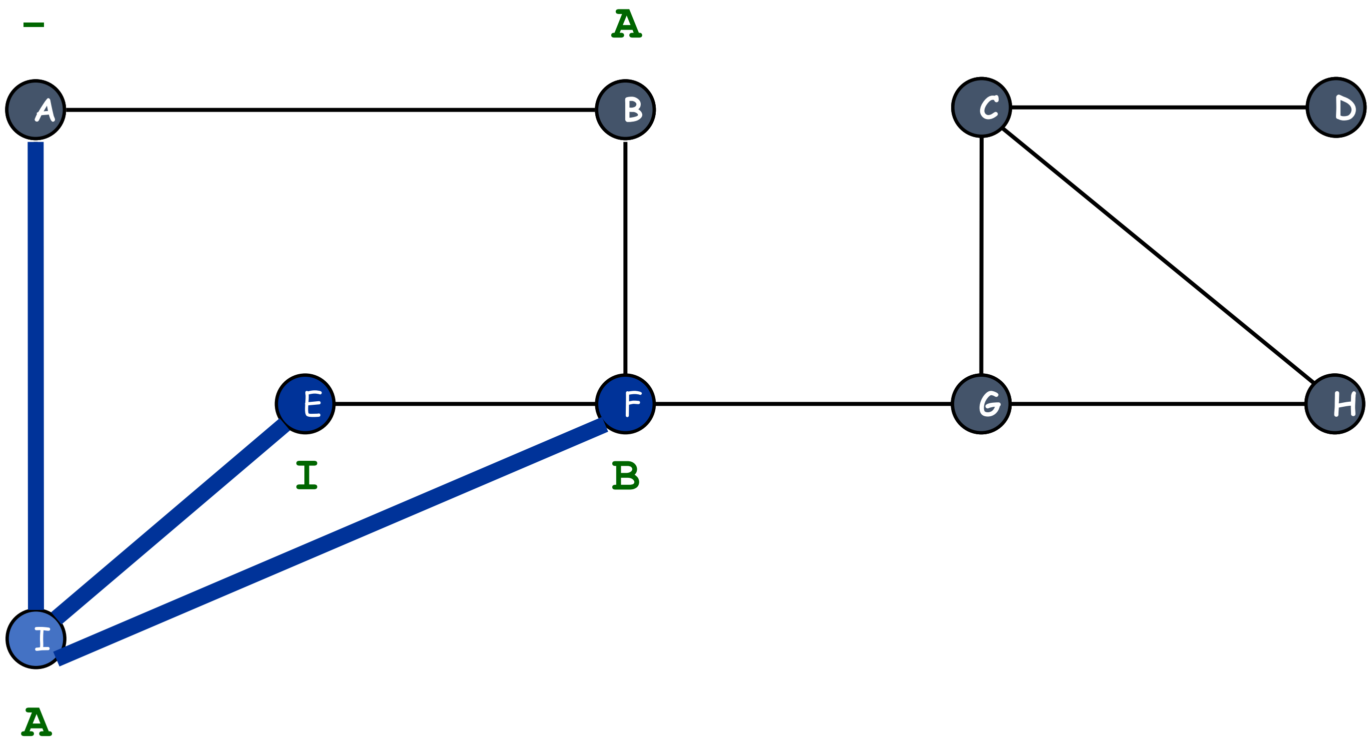


visit neighbors of I

front F E

FIFO Queue

Breadth-First Search

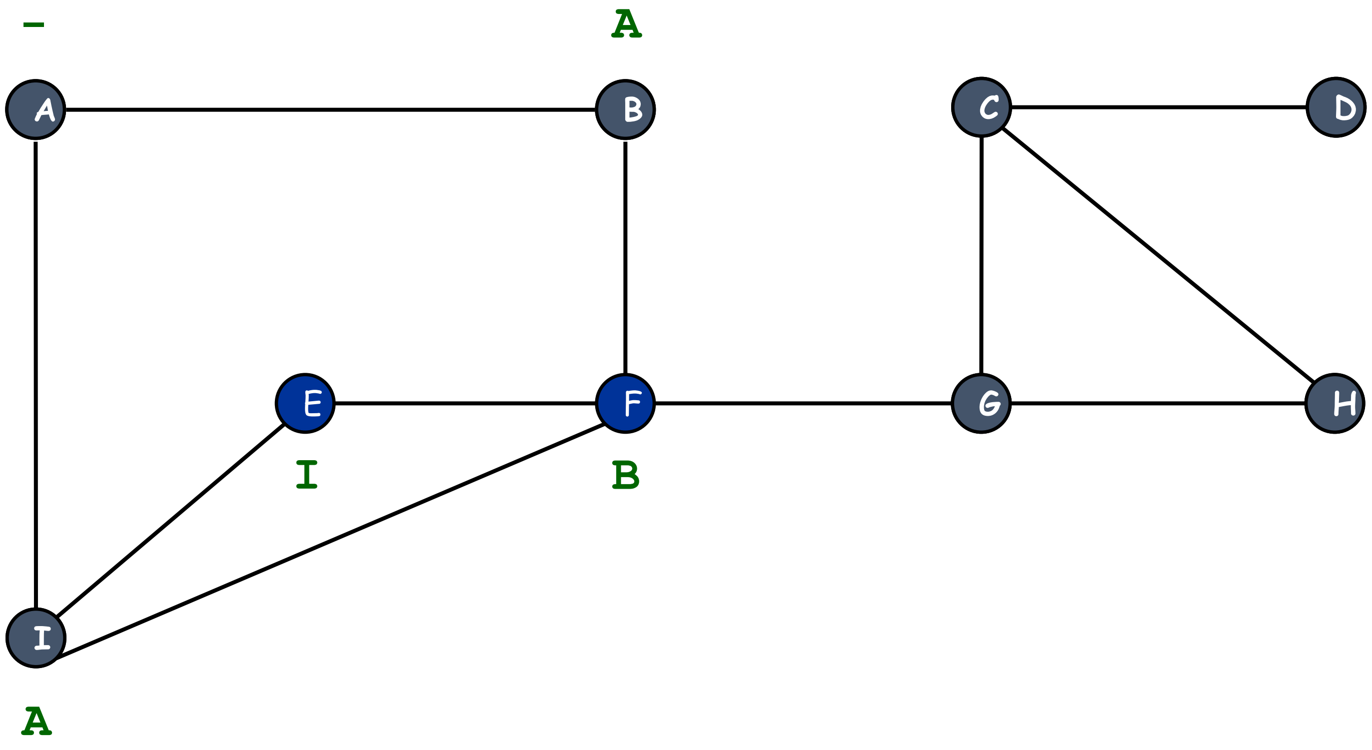


F already discovered

front F E

FIFO Queue

Breadth-First Search

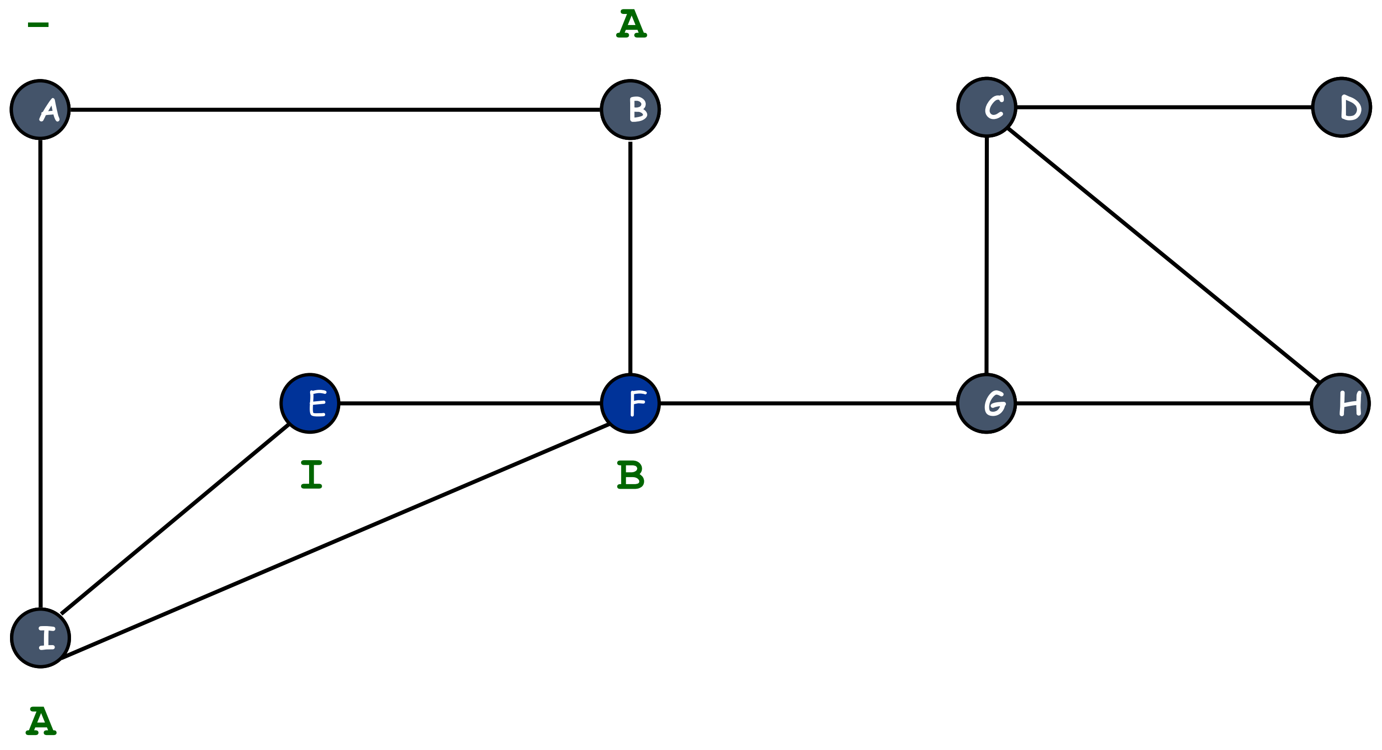


I finished

front F E

FIFO Queue

Breadth-First Search

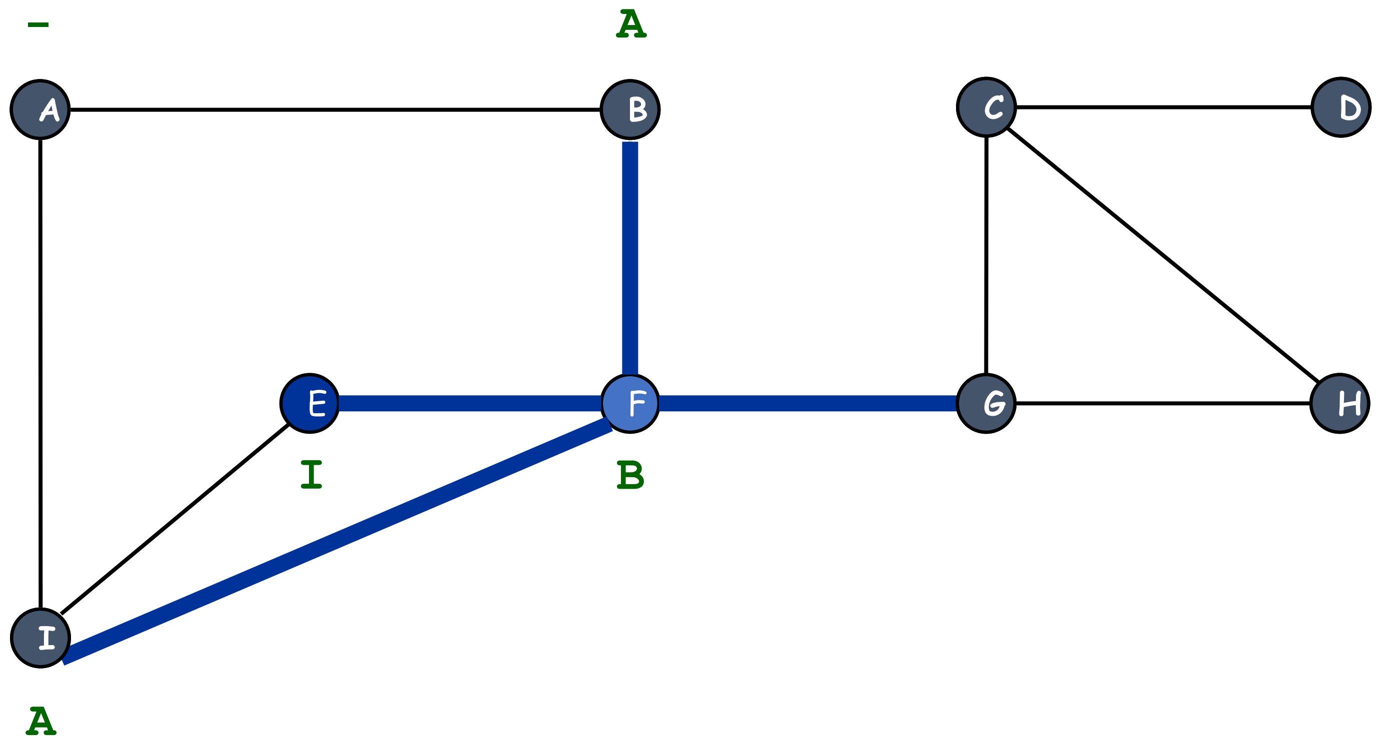


dequeue next vertex

front **F E**

FIFO Queue

Breadth-First Search

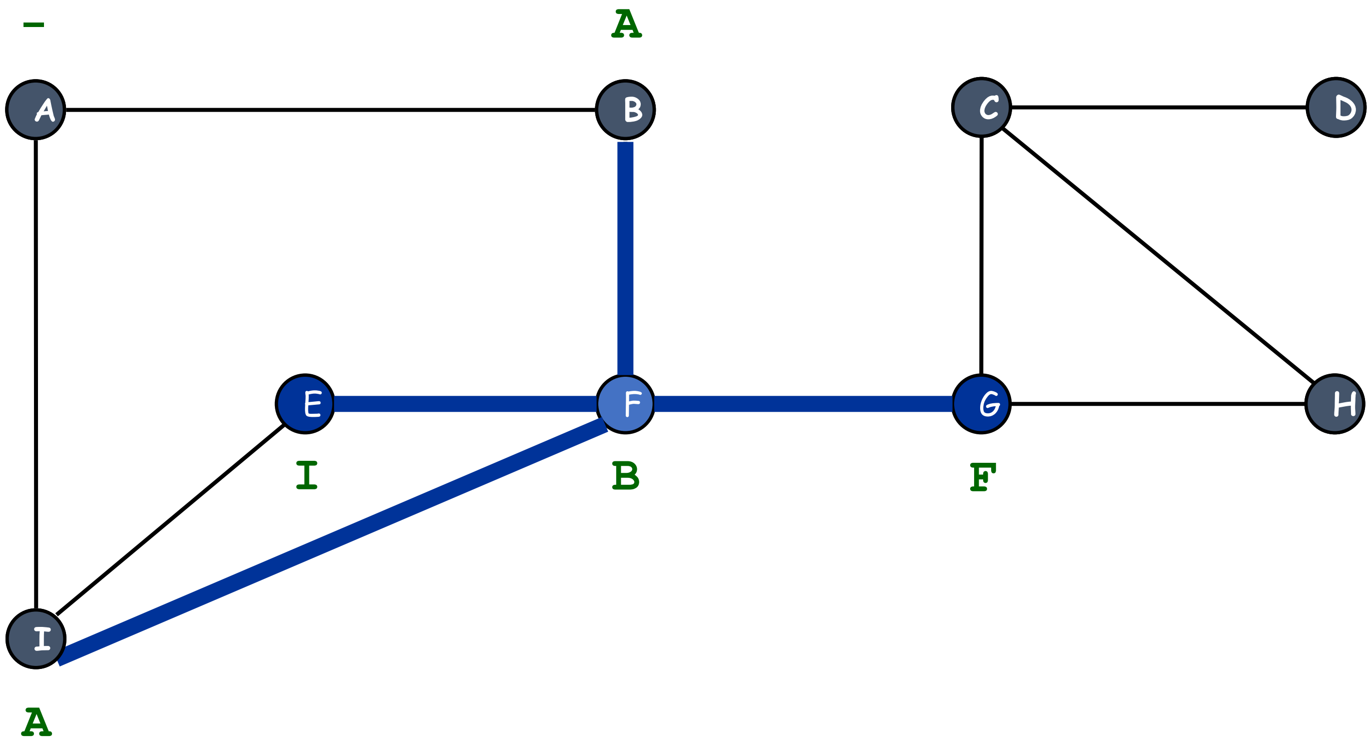


visit neighbors of F

front E

FIFO Queue

Breadth-First Search

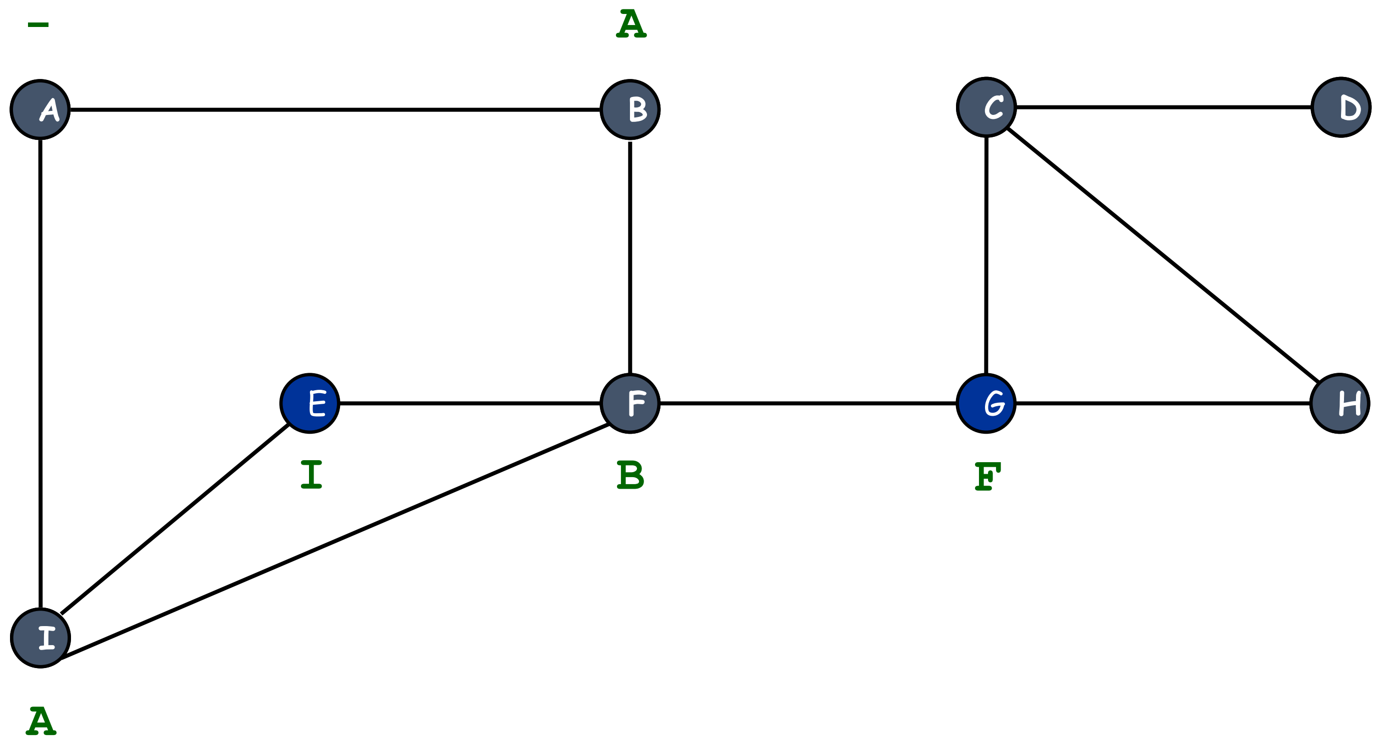


G discovered

front E G

FIFO Queue

Breadth-First Search

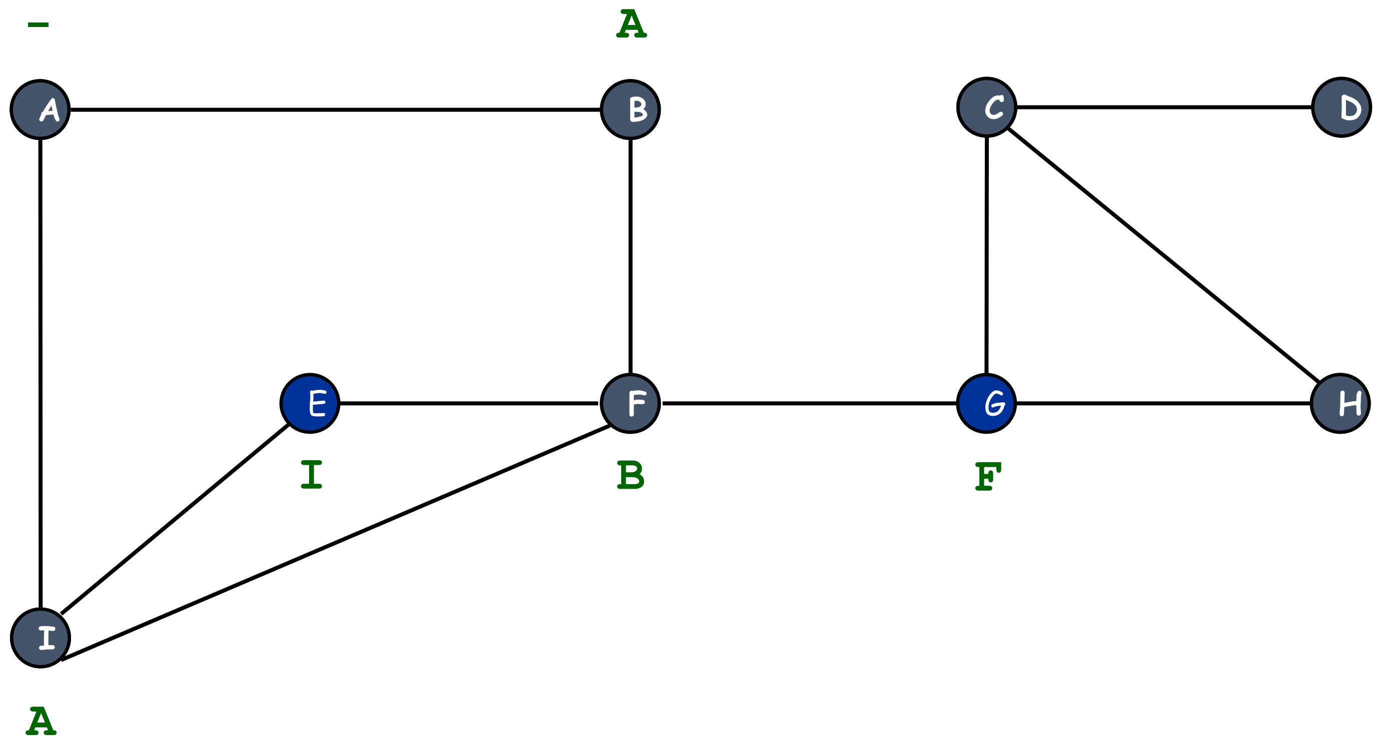


F finished

front E G

FIFO Queue

Breadth-First Search

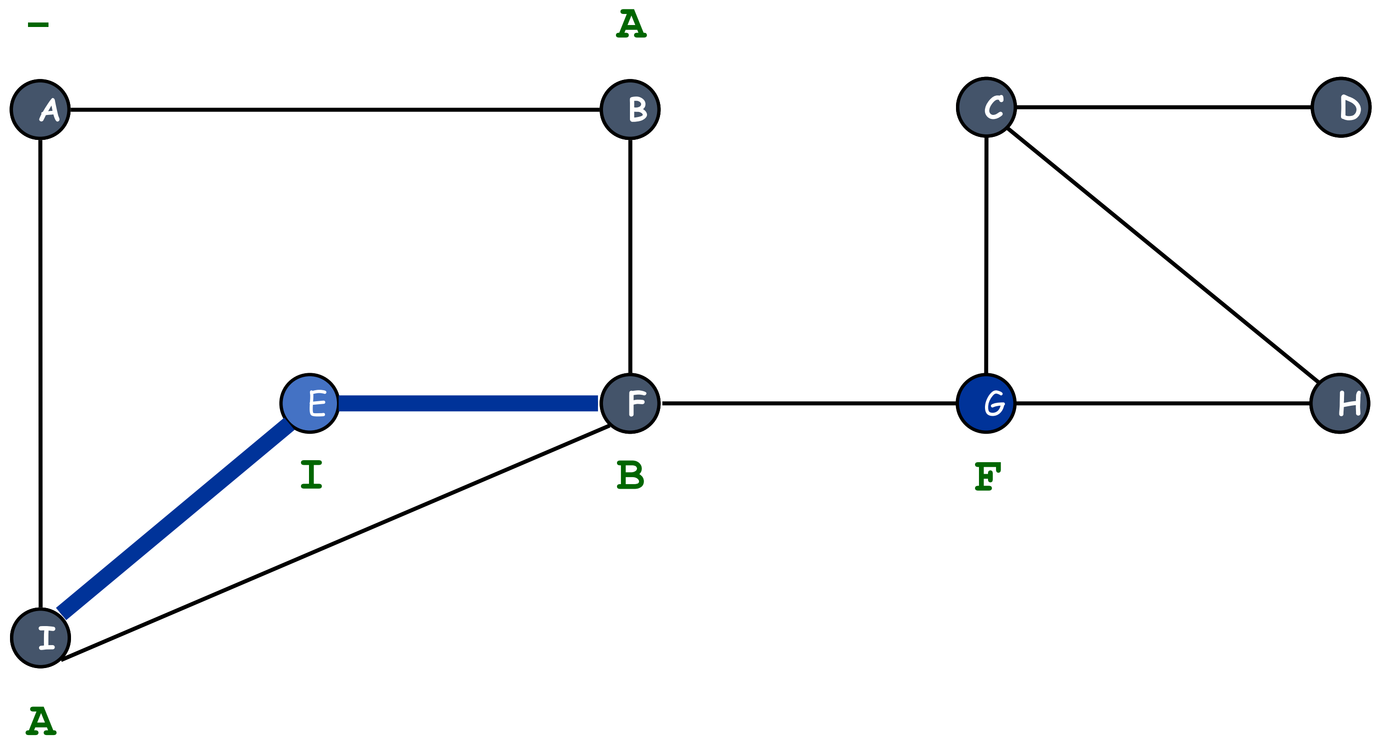


dequeue next vertex

front E G

FIFO Queue

Breadth-First Search

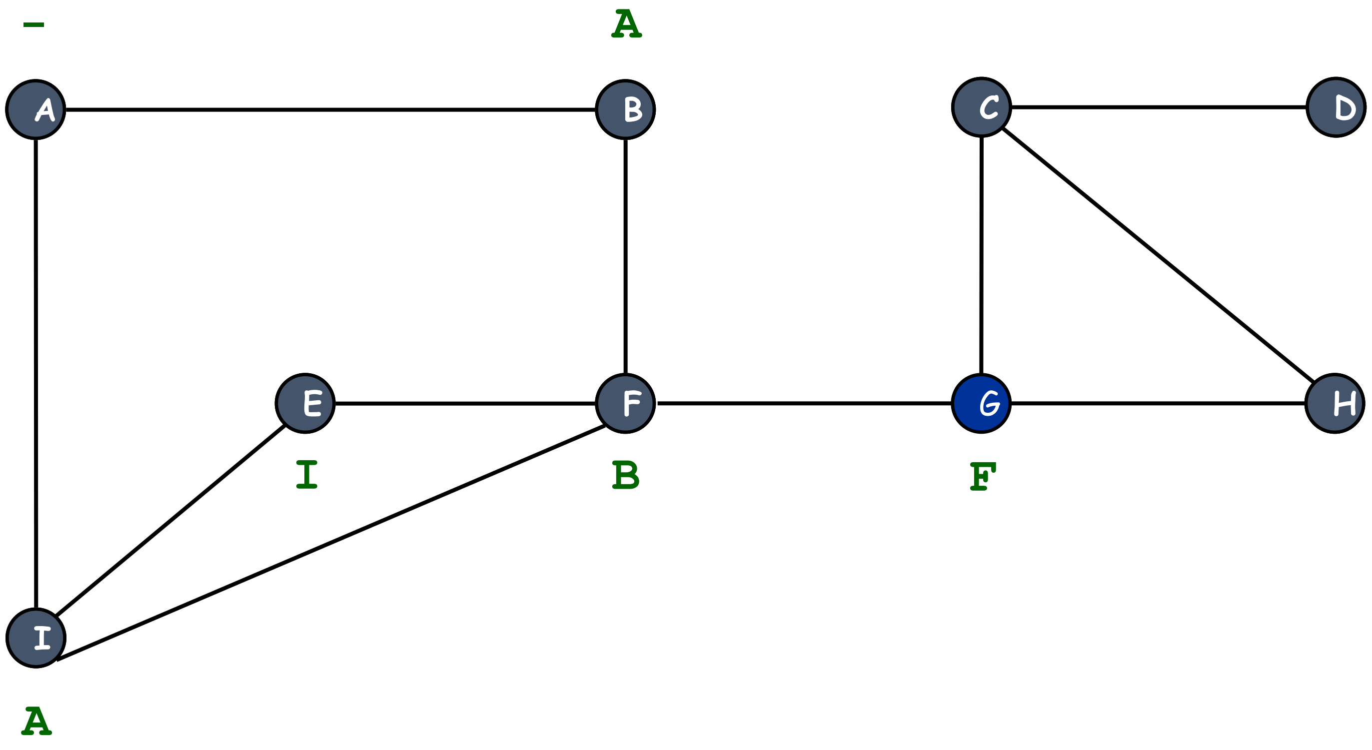


visit neighbors of E

front G

FIFO Queue

Breadth-First Search

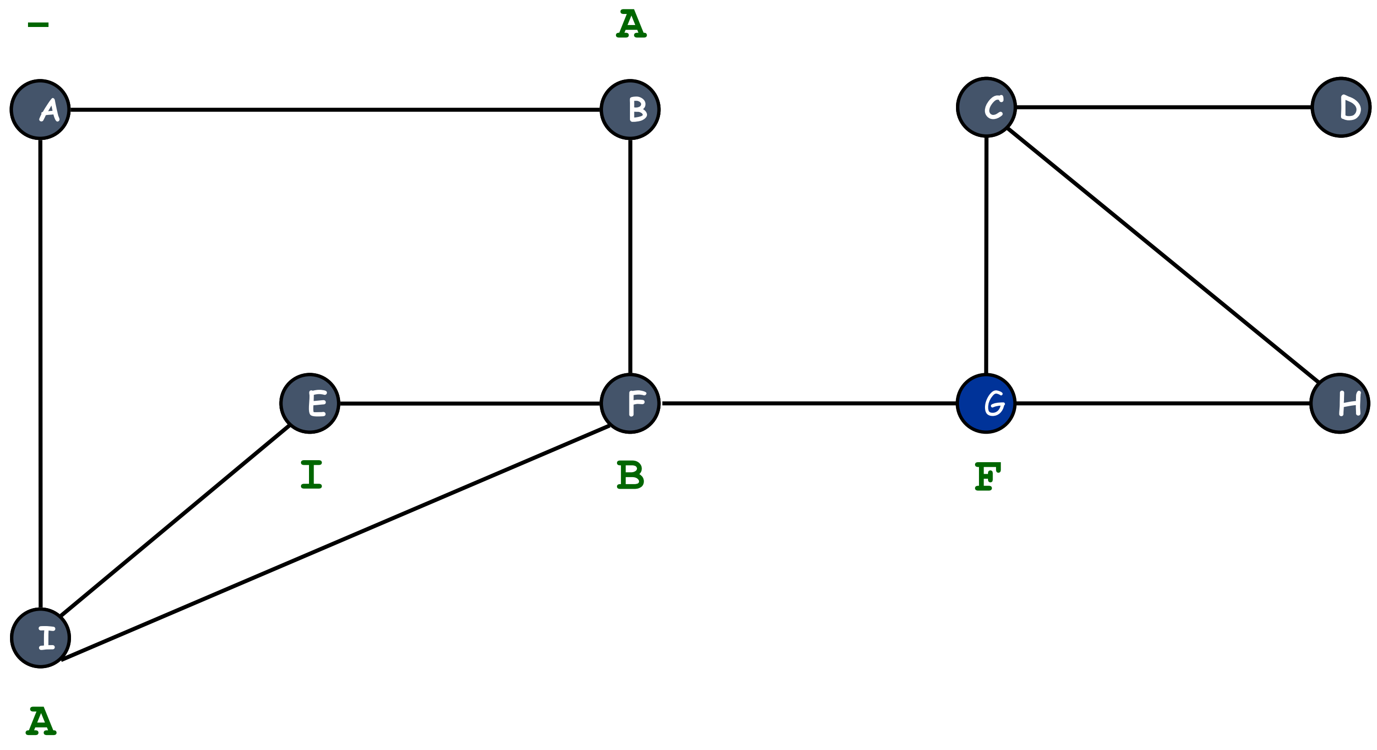


E finished

front G

FIFO Queue

Breadth-First Search

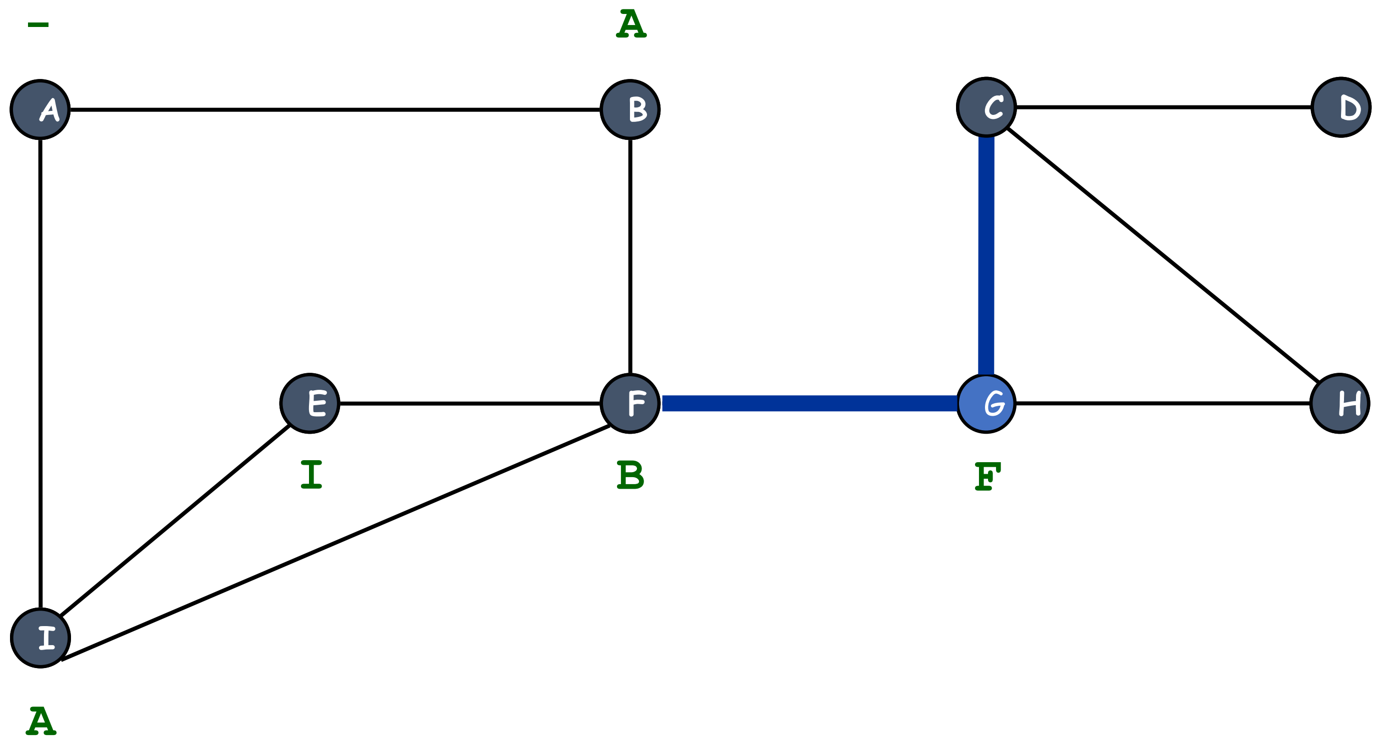


dequeue next vertex

front G

FIFO Queue

Breadth-First Search

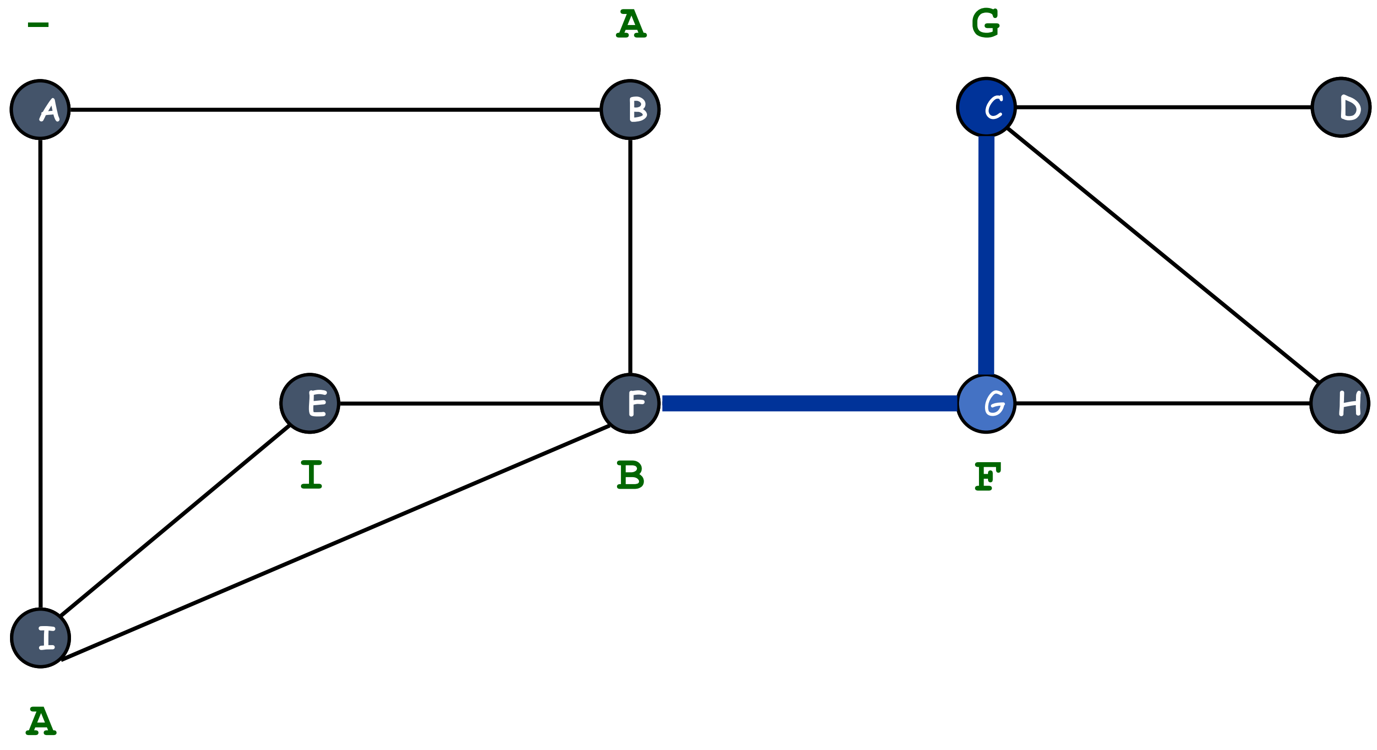


visit neighbors of G

front

FIFO Queue

Breadth-First Search

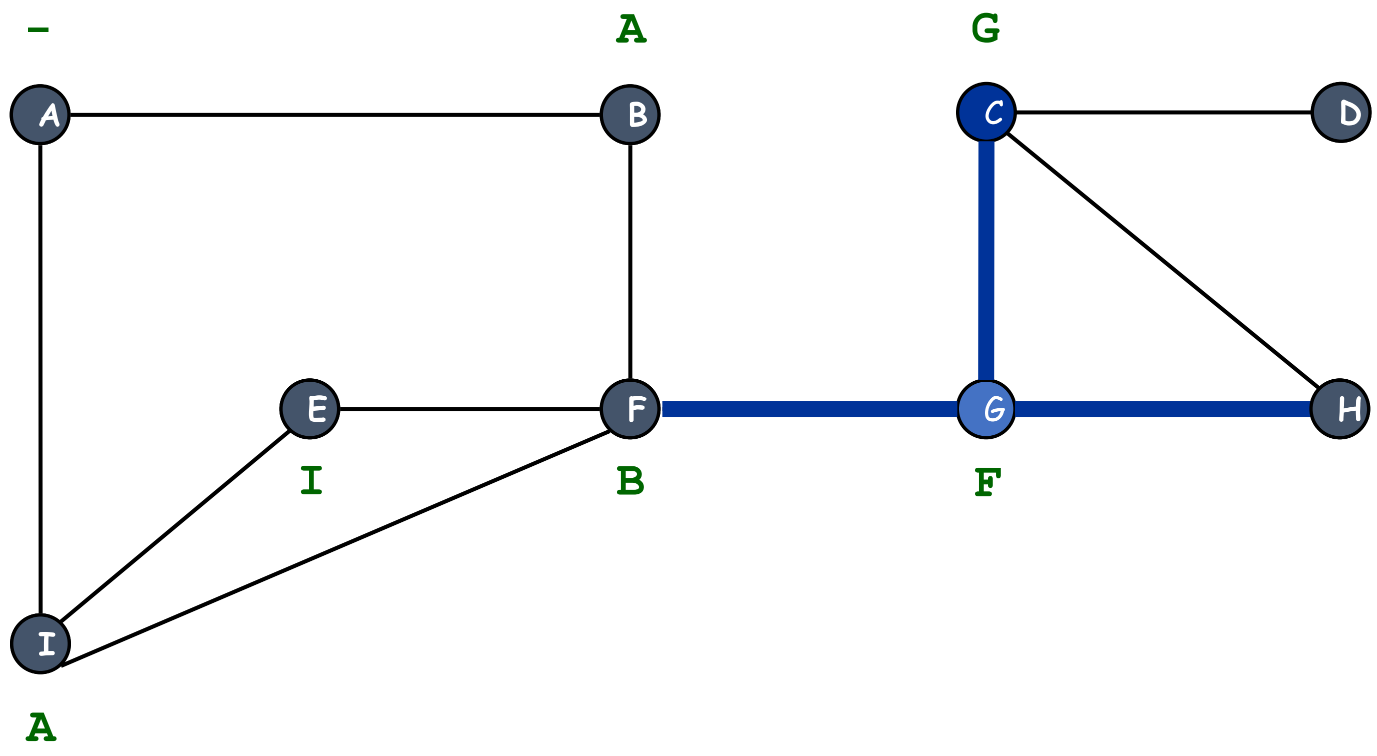


C discovered

front C

FIFO Queue

Breadth-First Search

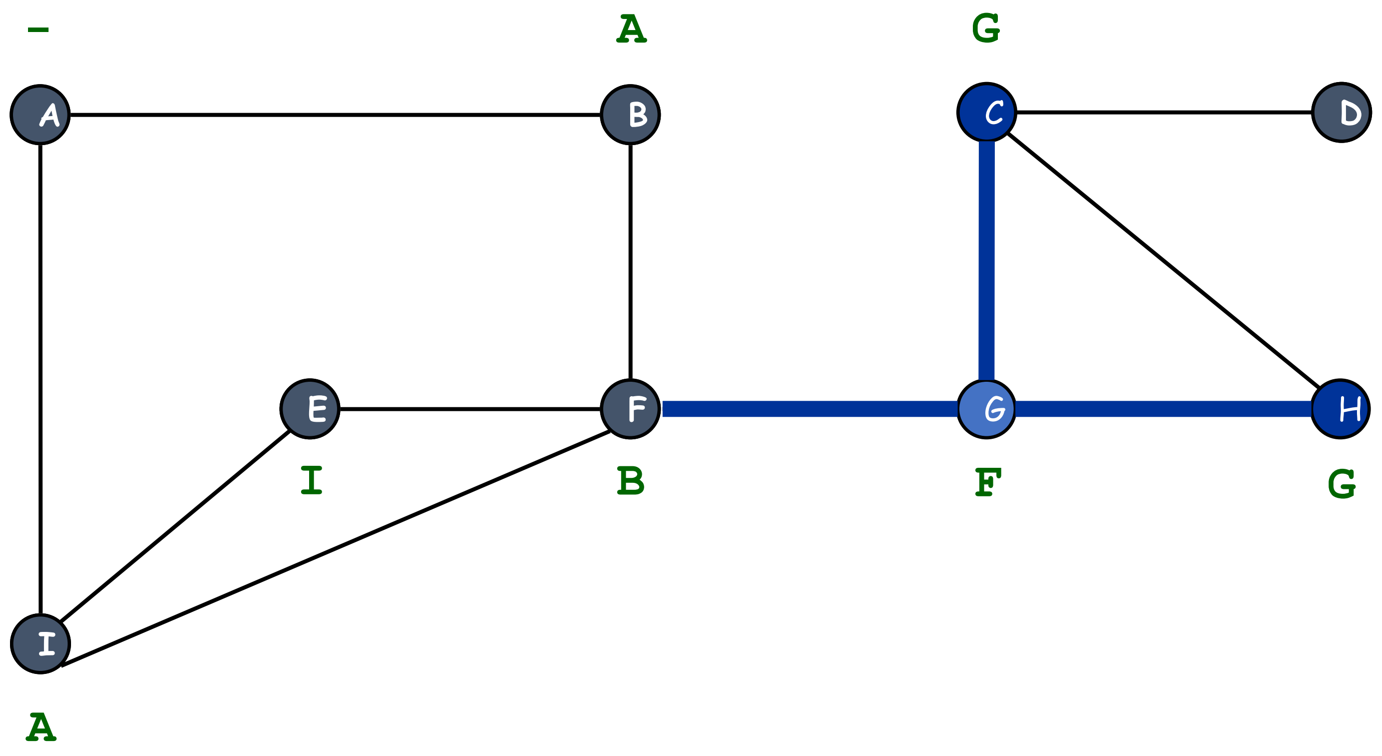


visit neighbors of G

front | C

FIFO Queue

Breadth-First Search

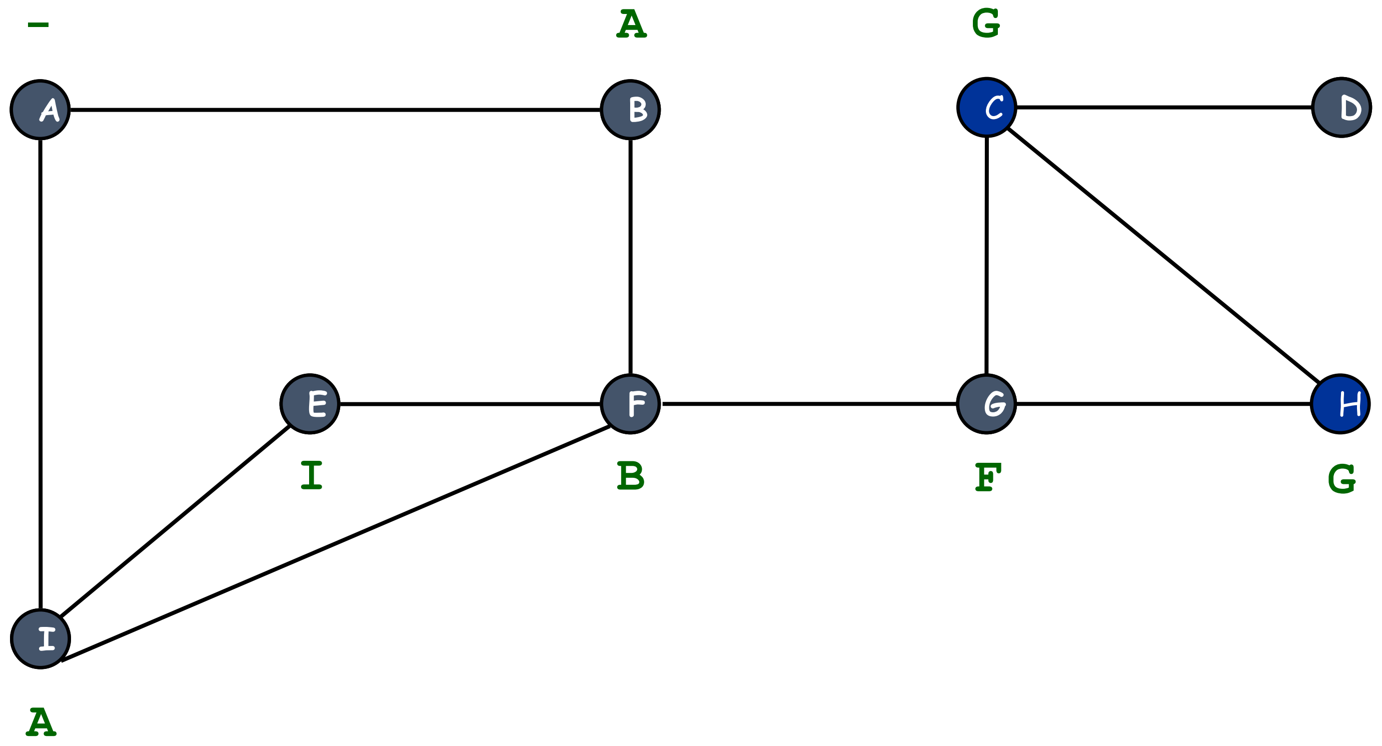


H discovered

front | C H

FIFO Queue

Breadth-First Search

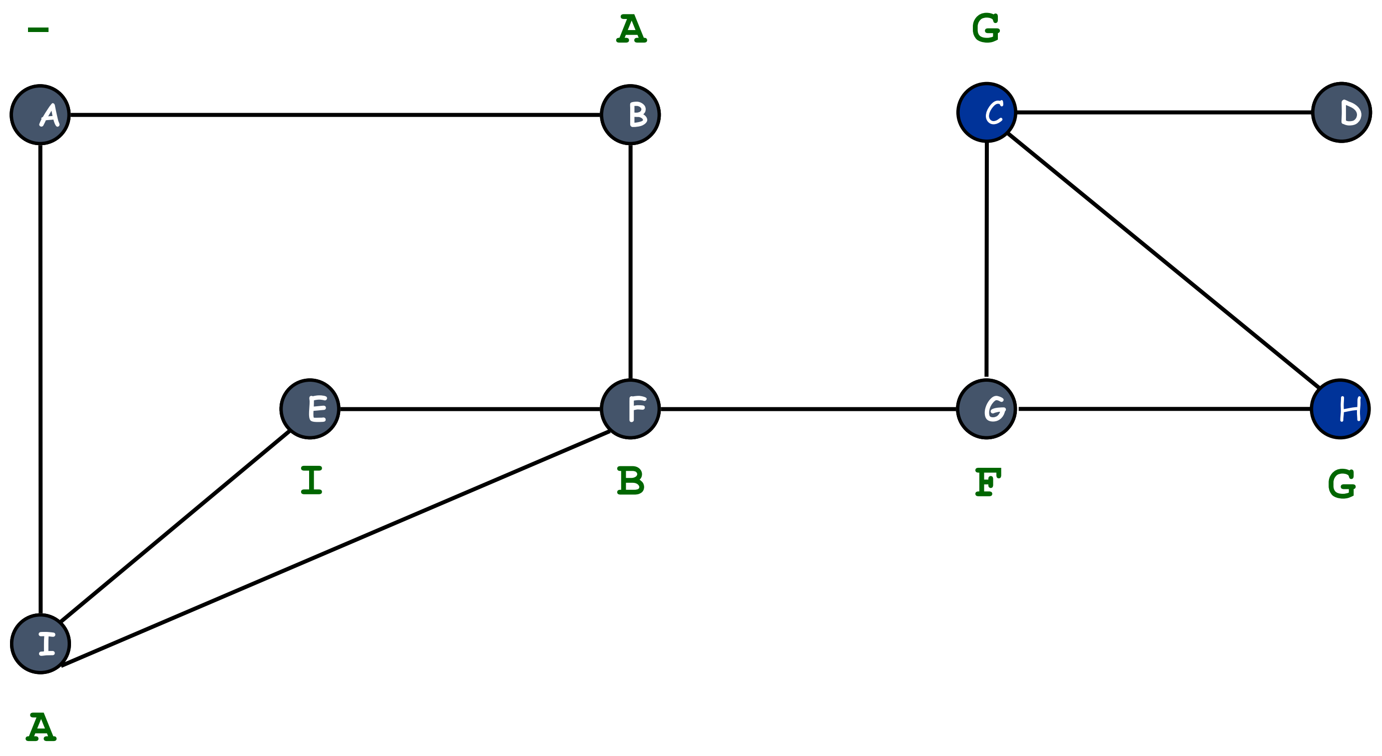


G finished

front C H

FIFO Queue

Breadth-First Search

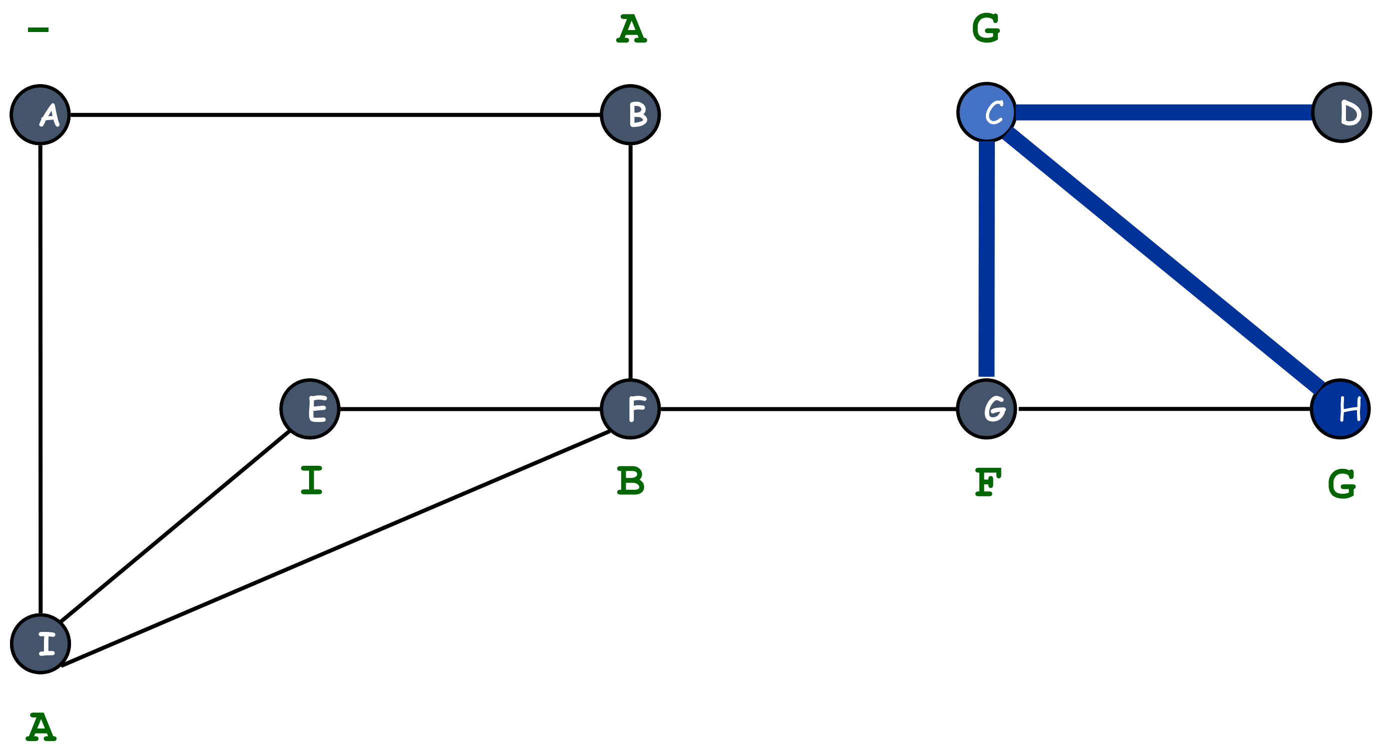


dequeue next vertex

front C H

FIFO Queue

Breadth-First Search

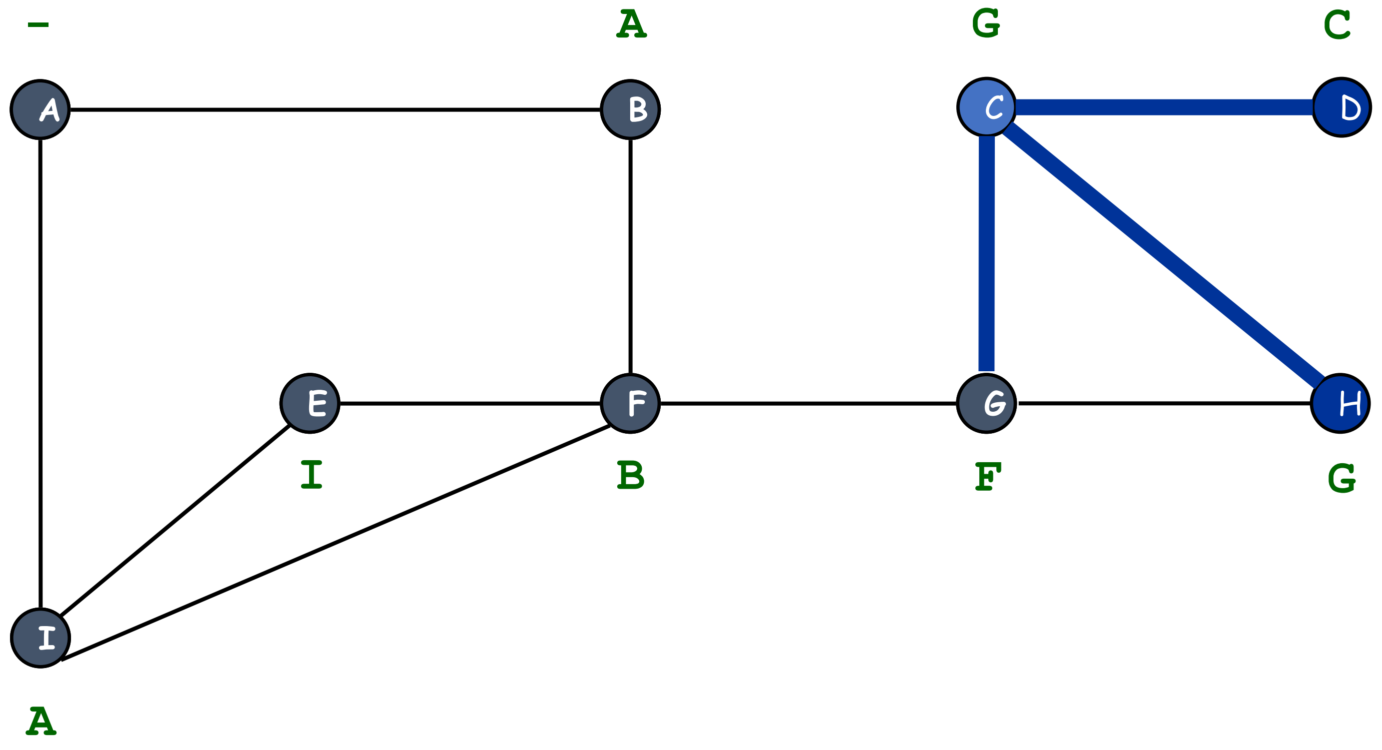


visit neighbors of C

front H

FIFO Queue

Breadth-First Search

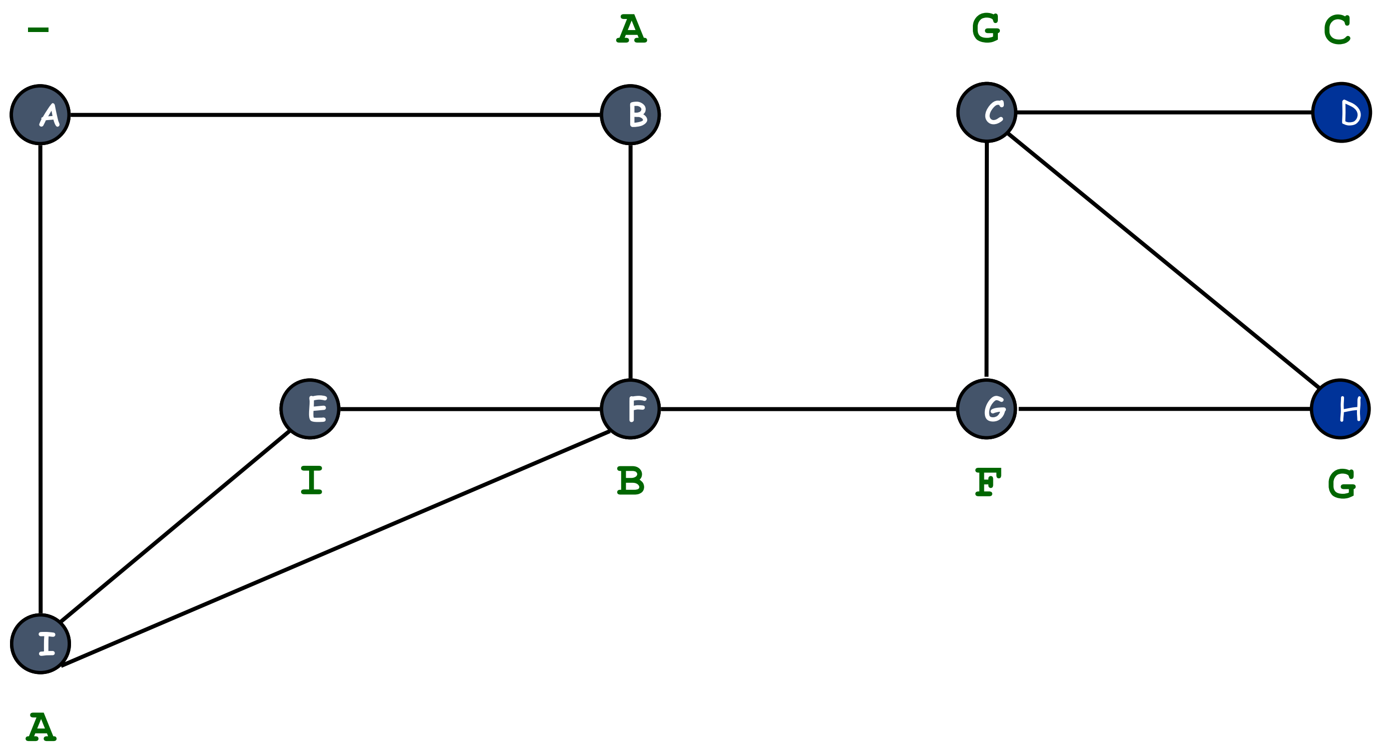


D discovered

front
H D

FIFO Queue

Breadth-First Search



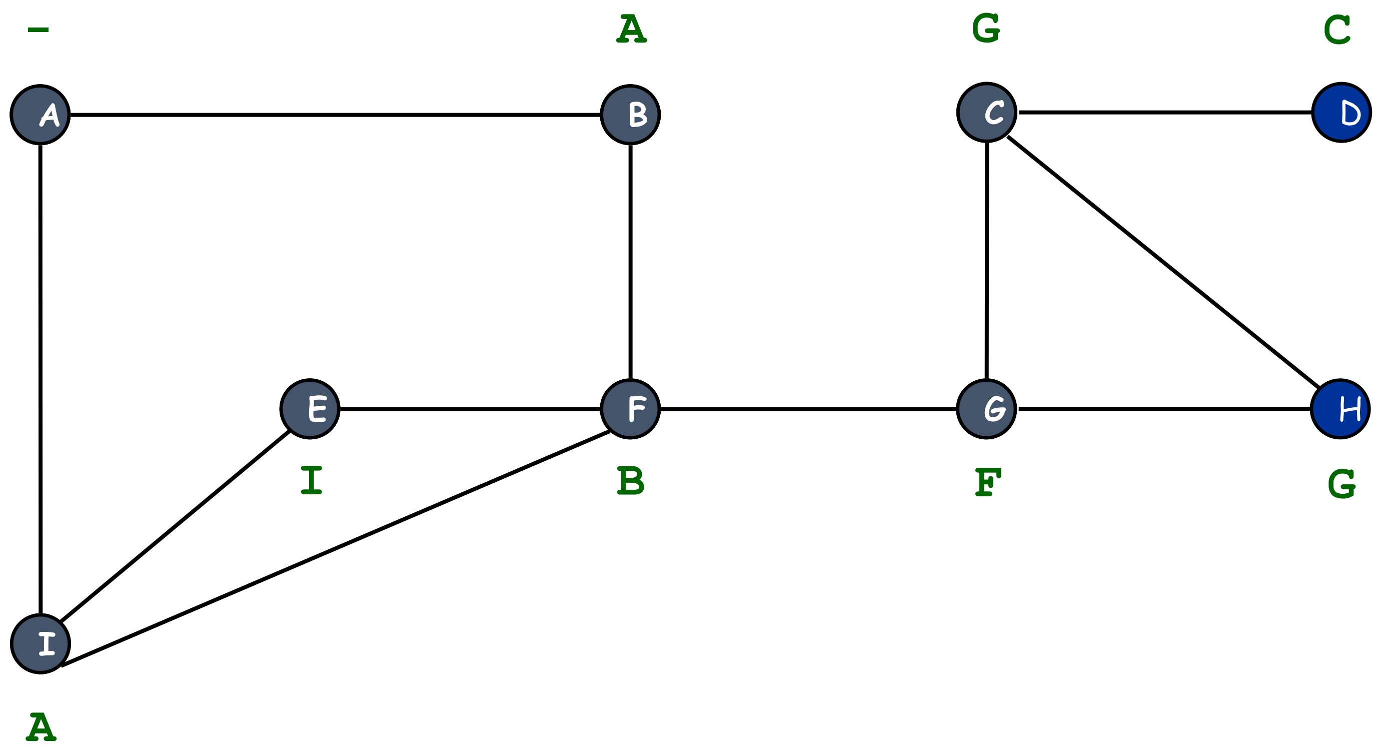
C finished

front

H D

FIFO Queue

Breadth-First Search

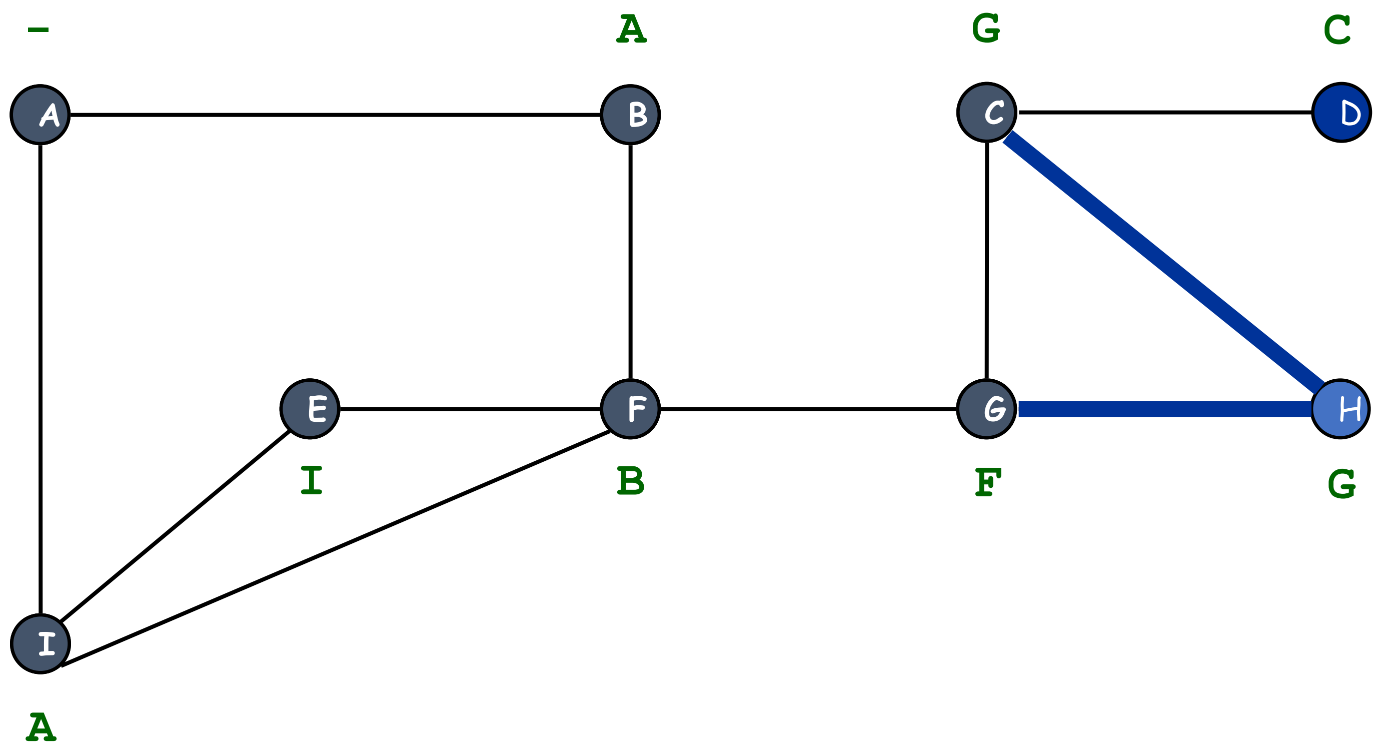


get next vertex

front H D

FIFO Queue

Breadth-First Search

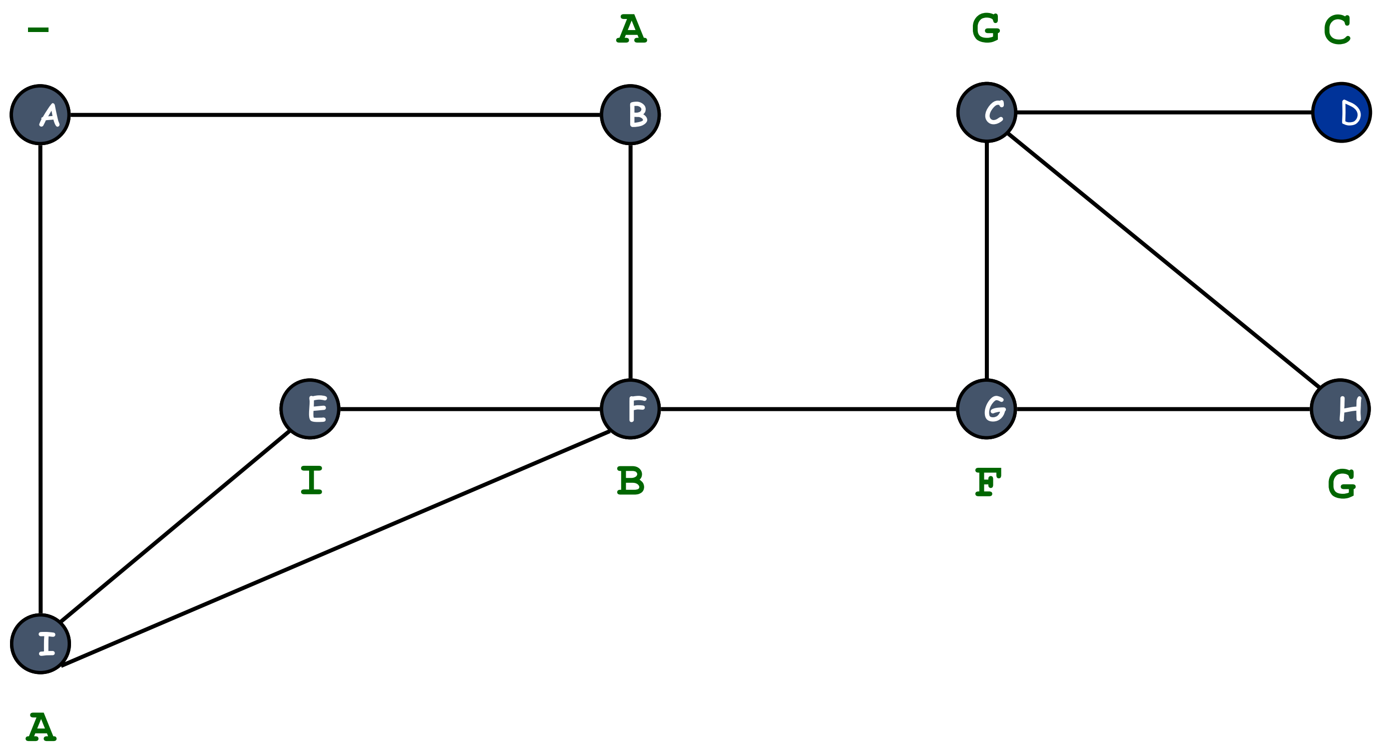


visit neighbors of H

front D

FIFO Queue

Breadth-First Search

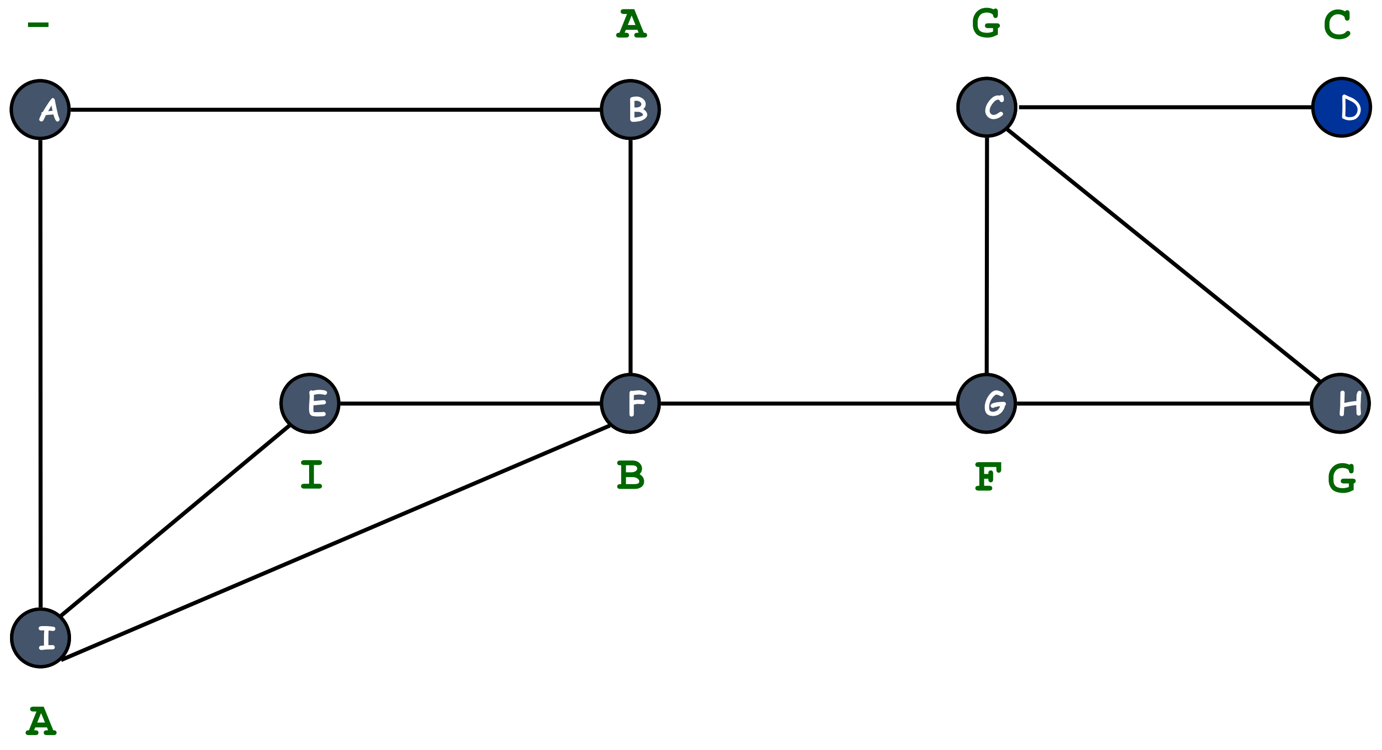


finished H

front D

FIFO Queue

Breadth-First Search

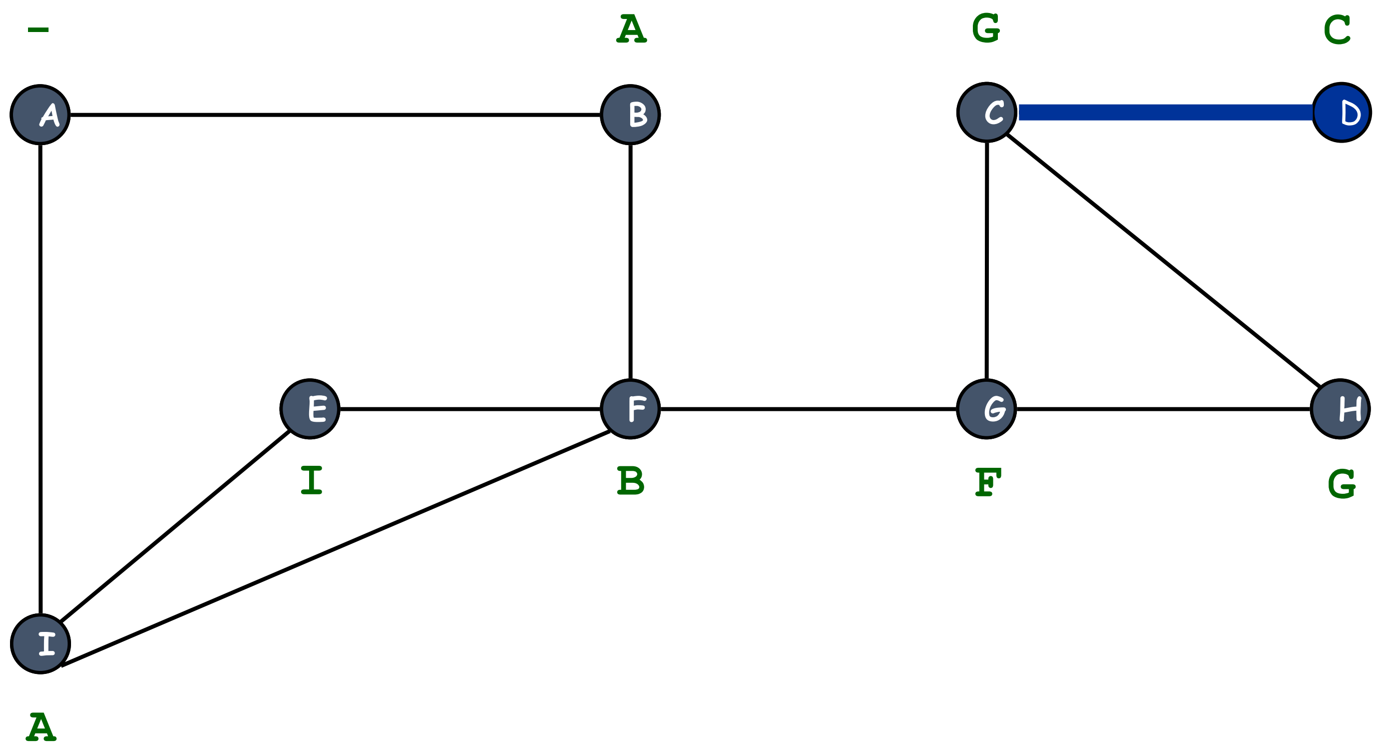


dequeue next vertex

front D

FIFO Queue

Breadth-First Search

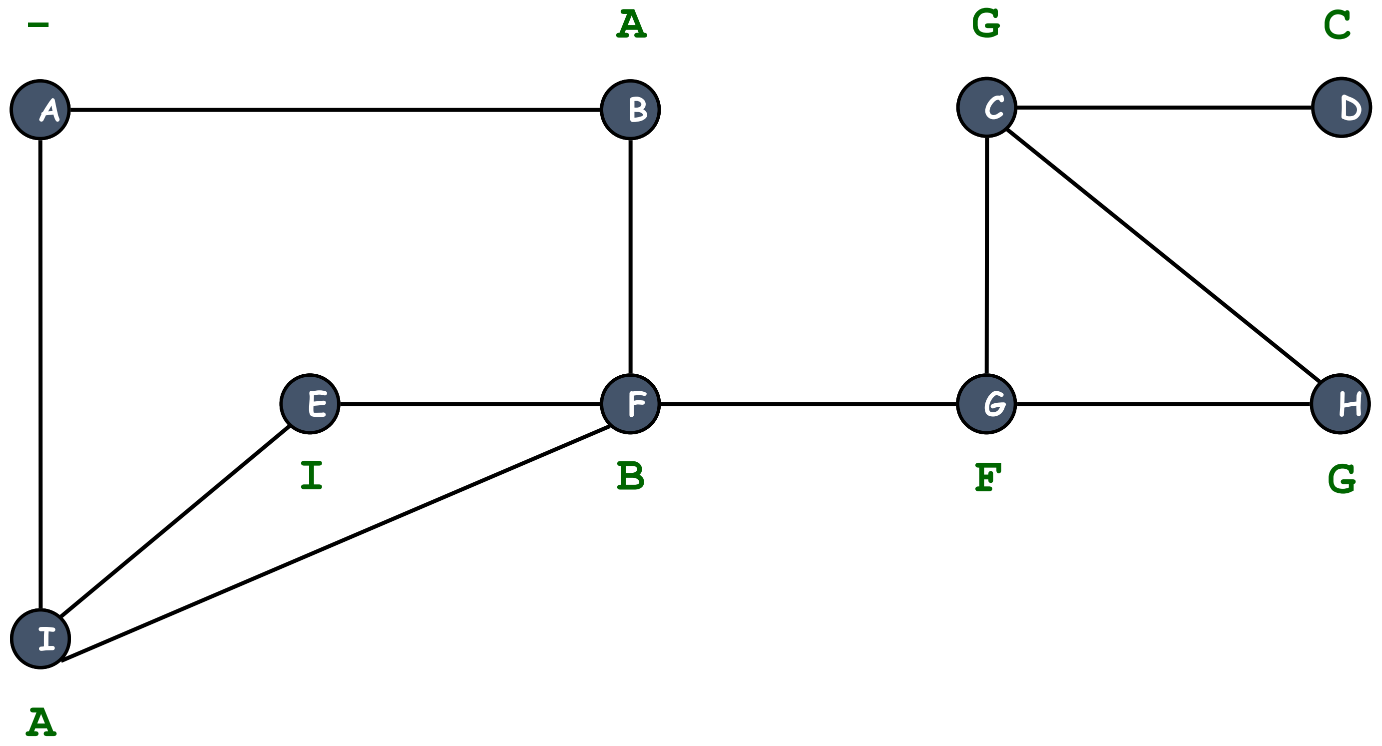


visit neighbors of D

front

FIFO Queue

Breadth-First Search

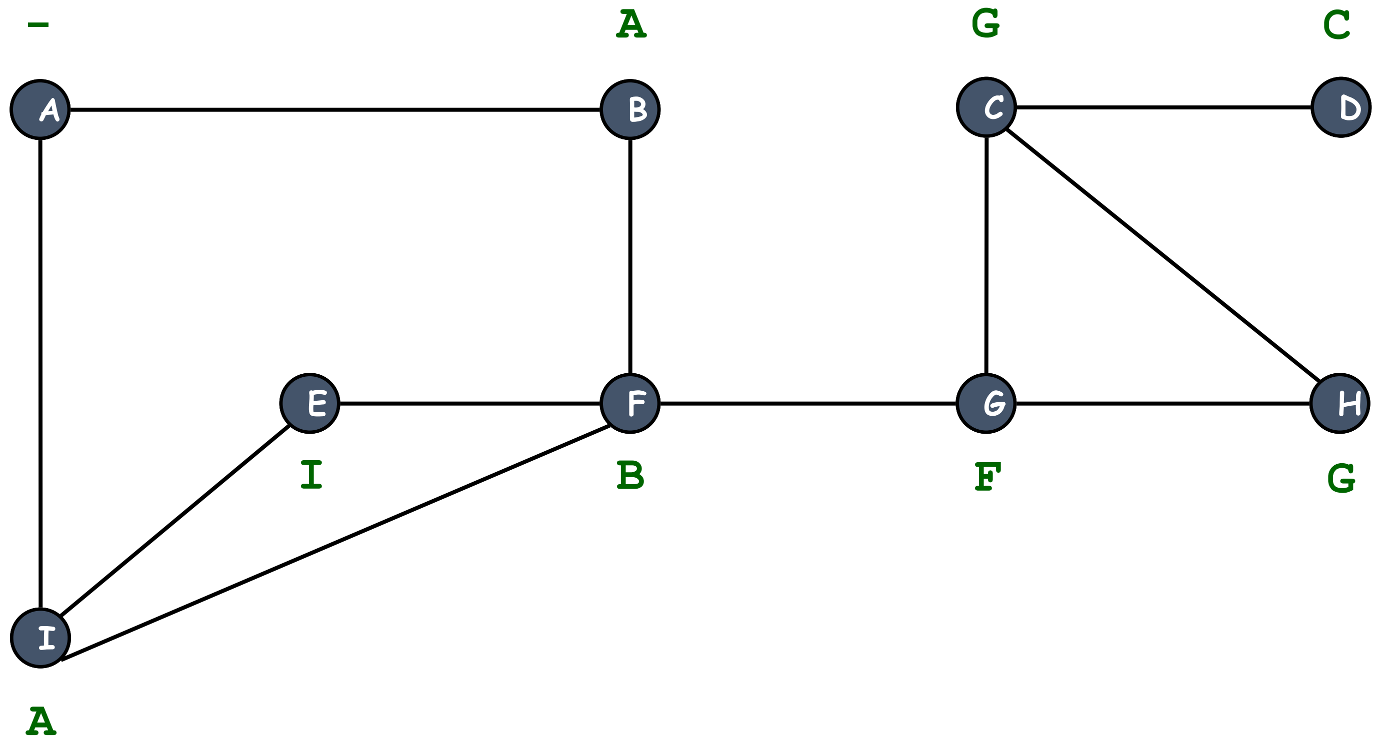


D finished

front

FIFO Queue

Breadth-First Search

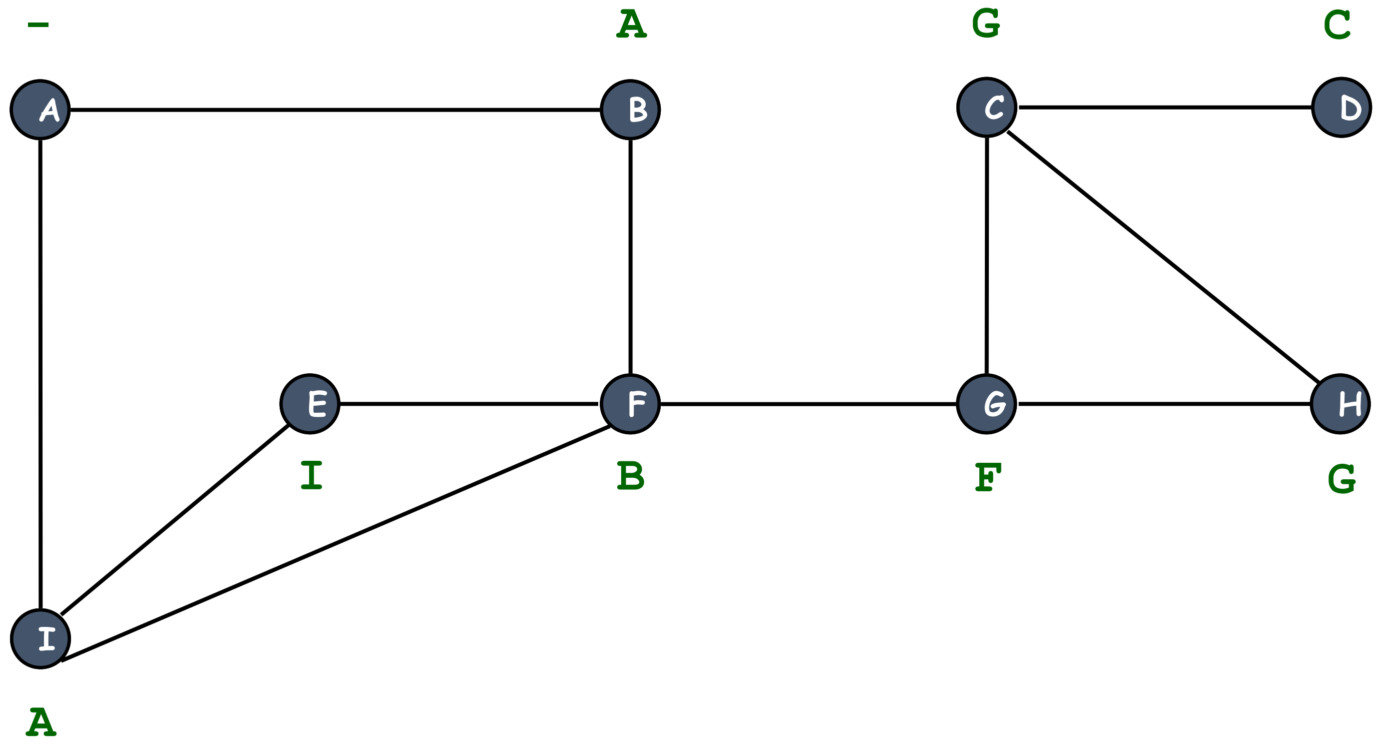


dequeue next vertex

front

FIFO Queue

Breadth-First Search



STOP

front

FIFO Queue

Breadth-First Search (BFS)

```
Create a queue Q
Mark initial node v as visited and enqueue v in Q
While Q is non-empty
   Dequeue u from Q
   For each unvisited neighbor n of u:
      Mark n as visited
      Enqueue n into Q
```

Cool animation

Time complexity: *O(m+n)* for a graph of m vertices and n edges

BFS generalizes level-order tree traversal

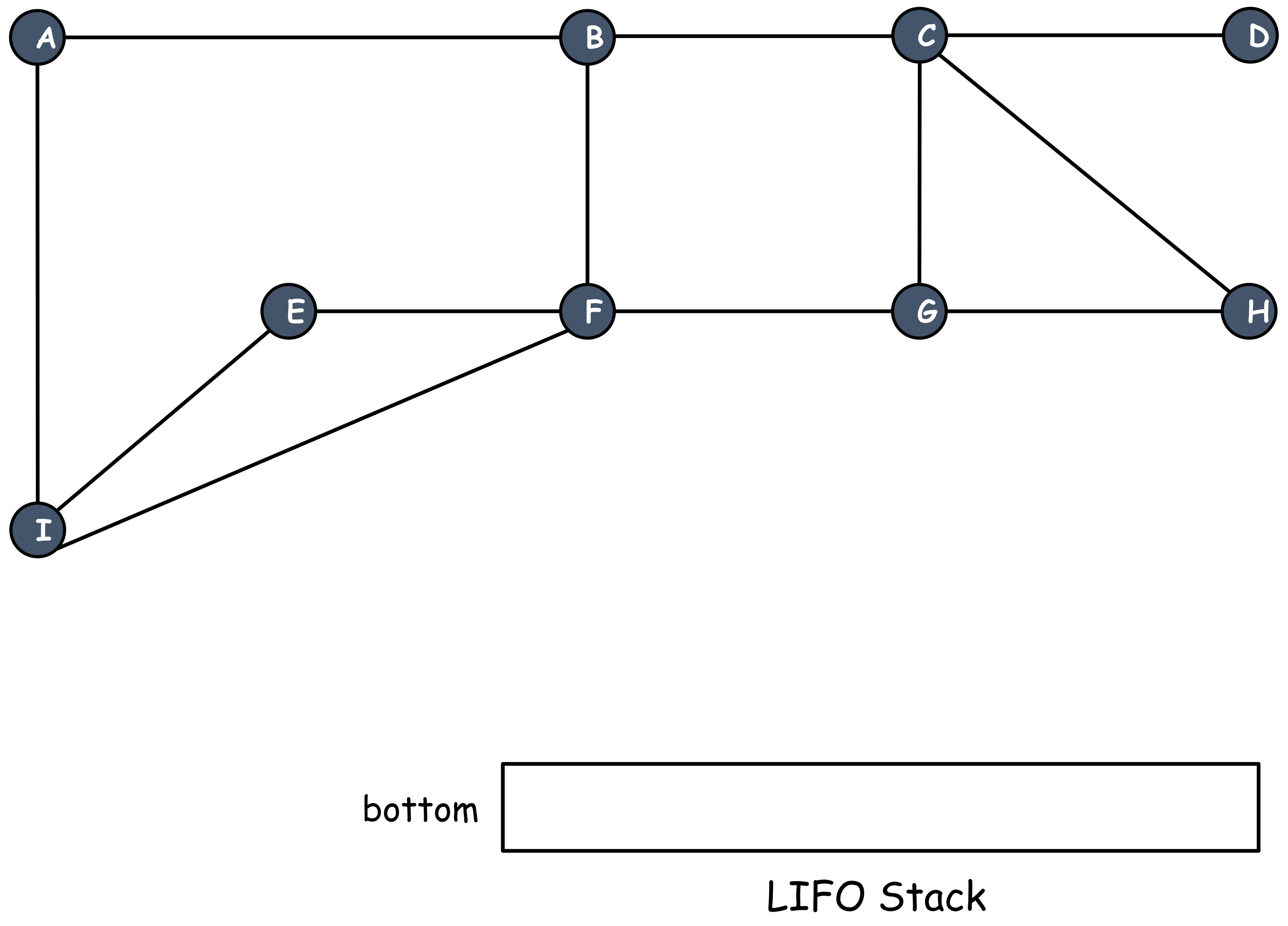
Depth-First Search (DFS)

```
Create a stack Q
Mark initial node v as visited and push v in Q
While Q is non-empty
  Pop u from Q
  For each unvisited neighbor n of u:
    Mark n as visited
    Push n into Q
```

Cool animation

DFS generalizes preorder tree traversal

Exercise: Depth-First Search



BFS in Action: Shortest Path Length

Given a graph G = (V, E) and a node s in V:

- For each node v in V, compute the **length of the shortest path** from s to v.

```
BFS(G,s)
for u ∈ G.V do
    u.color := white
    u.dist := ∞
    u.pred := NULL
s.color := gray
s.dist := 0
Q := new Queue()     // FIFO queue
Q.enqueue(s)
while not Q.isEmpty() do
    u := Q.dequeue()
    for v ∈ u.adj do
        if v.color = white
          then v.color := gray
               v.dist := u.dist + 1
               v.pred := u
               Q.enqueue(v)
```

Initialize all vertices

Initialize BFS with *s*

Handle all of *u*'s children before handling children of children

Could we use DFS here?

- A vertex is white if it is undiscovered
- A vertex is gray if it has been discovered

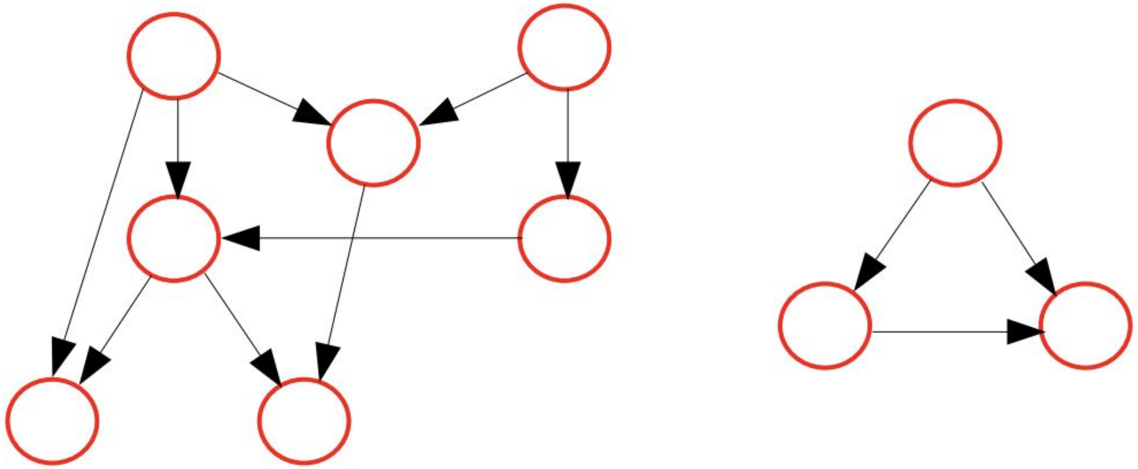
BFS In Action: Shortest Path

```
bool shortest_path (vector<int> adj[], int src, int dest, int v, int pred[], int dist[]) {
    // a queue to maintain queue of vertices whose adjacency list is to be scanned as per normal BFS algorithm
    list<int> queue;
    // boolean array visited[] which stores the information whether ith vertex is reached at least once BFS
    bool visited[v];
    // initially all vertices are unvisited so v[i] for all i is false and as no path is yet constructed
    // dist[i] for all i set to infinity
    for (int i = 0; i < v; i++) {
        visited[i] = false;
        dist[i] = INT_MAX;
        pred[i] = -1;
    }

    // now source is first to be visited and distance from source to itself should be 0
    visited[src] = true; dist[src] = 0; queue.push_back(src);
    // standard BFS algorithm
    while (!queue.empty()) {
        int u = queue.front(); queue.pop_front();
        for (int i = 0; i < adj[u].size(); i++) {
            if (visited[adj[u][i]] == false) {
                visited[adj[u][i]] = true;
                dist[adj[u][i]] = dist[u] + 1;
                pred[adj[u][i]] = u;
                queue.push_back(adj[u][i]);
                // We stop BFS when we find destination.
                if (adj[u][i] == dest)
                    return true;
            }
        }
    }
```

Directed Acyclic Graph

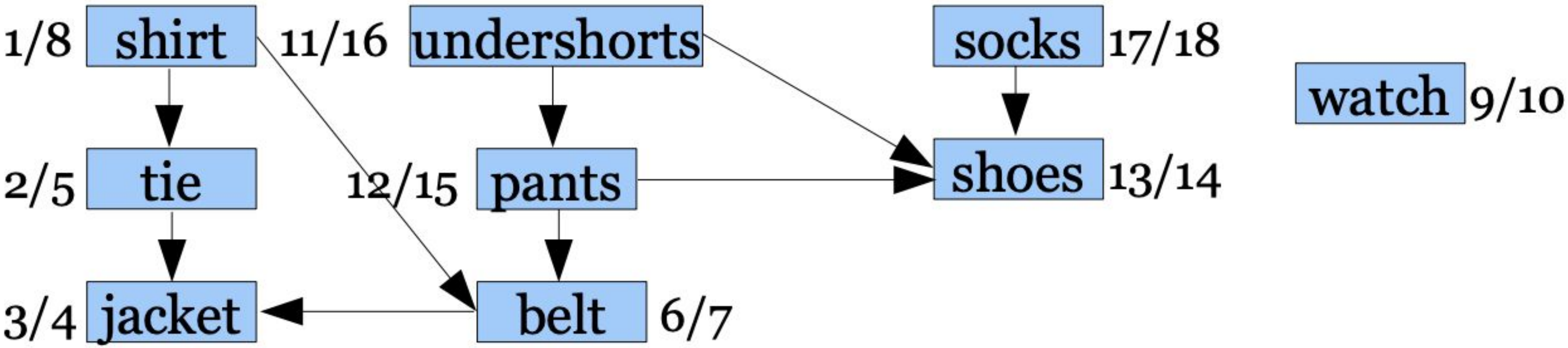
- A DAG is a directed graph without cycles



- DAGs are used to indicate precedence among events (event *x* must happen before *y*)
- An example would be a parallel code execution

DAG Operation: Topological Sort

A **topological sort** is an *ordering* of DAG vertices such that, if there is a *path* from v1 to v2, v1 appears *before* v2 in the ordering



One topological ordering: *(many are possible!)*
shirt, tie, undershorts, pants, belt, jacket, socks, shoes, watch

Brute force: repeatedly remove nodes with 0 incoming edges
Time complexity?

Topological Sort Algorithm: Kahn's Alg

Idea: BFS-style "take apart" the graph in order of its edges

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge // can be a queue or a stack

while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

Time complexity?

Topological Sort: DFS-Based

- **Idea:** DFS-style dive into the graph, visit most dependent nodes **first** and least dependent — **last** (use a stack to keep track)
- Initialize all nodes to be unvisited and stack **S** to be empty

```
For all nodes n:
   If n is unvisited:
      topoSort(n)
Pop and print S

topoSort(node n):
   Mark n as visited
   For each outgoing edge n->v:
      If v is not visited: topoSort(v)
   Push n into S // pushes on top of their reachable "children"
```

Time complexity?

Topological Sort: DFS-Based Example