Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Kazan, Russia

Recap

1. Graphs
2. Graph ADT
3. Graph Representations

Objectives

1. Build a definition for the ”connected component of a graph”

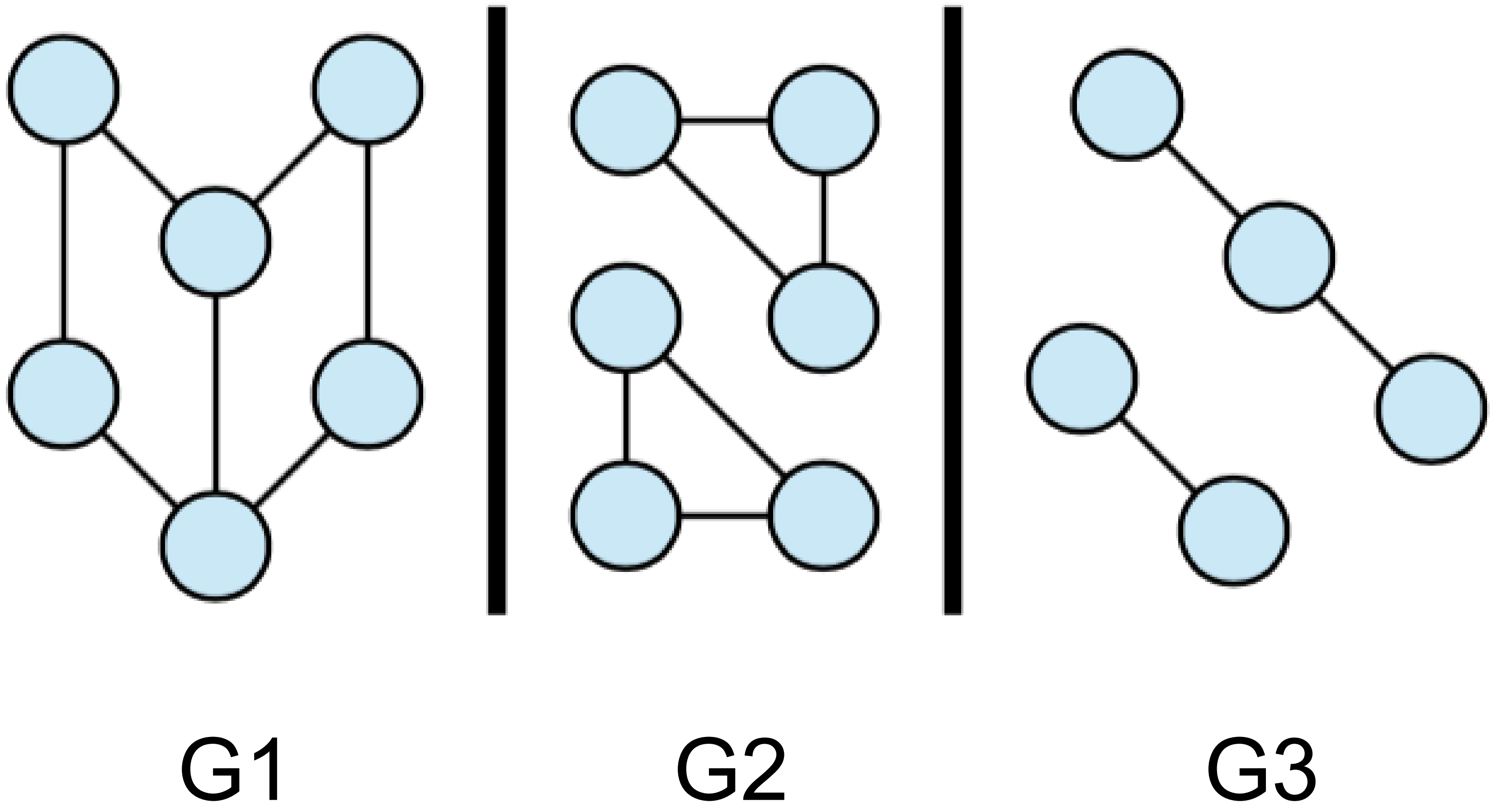
2. Learn graph traversals

- Depth First Search (DFS)

- Breadth First Search (BFS)

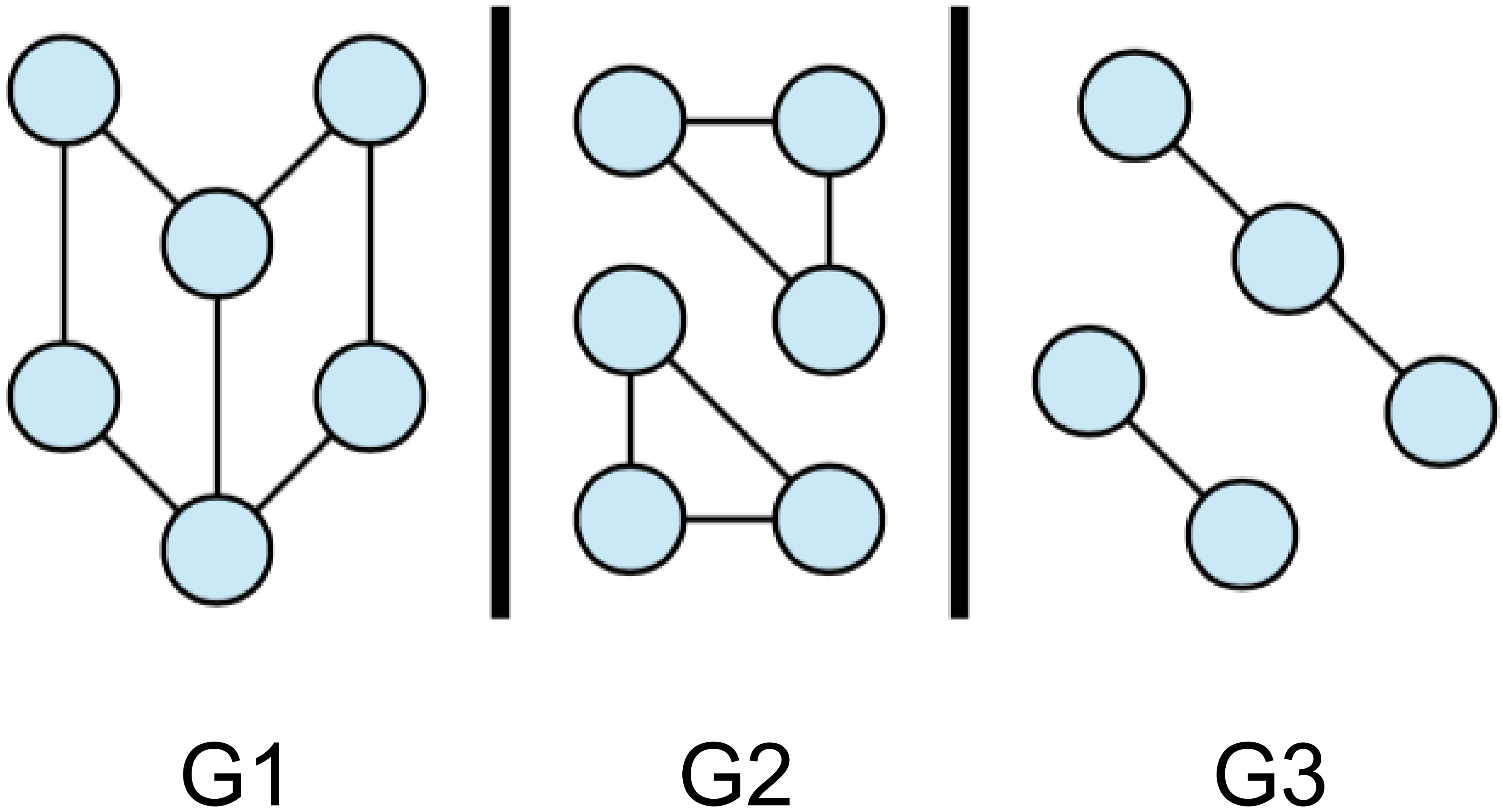
Connected Component

- Let G be an undirected graph.
- Two nodes u and v are called connected if there is a path from u to v in G ($u \longleftrightarrow v$)
- Now consider the following graphs

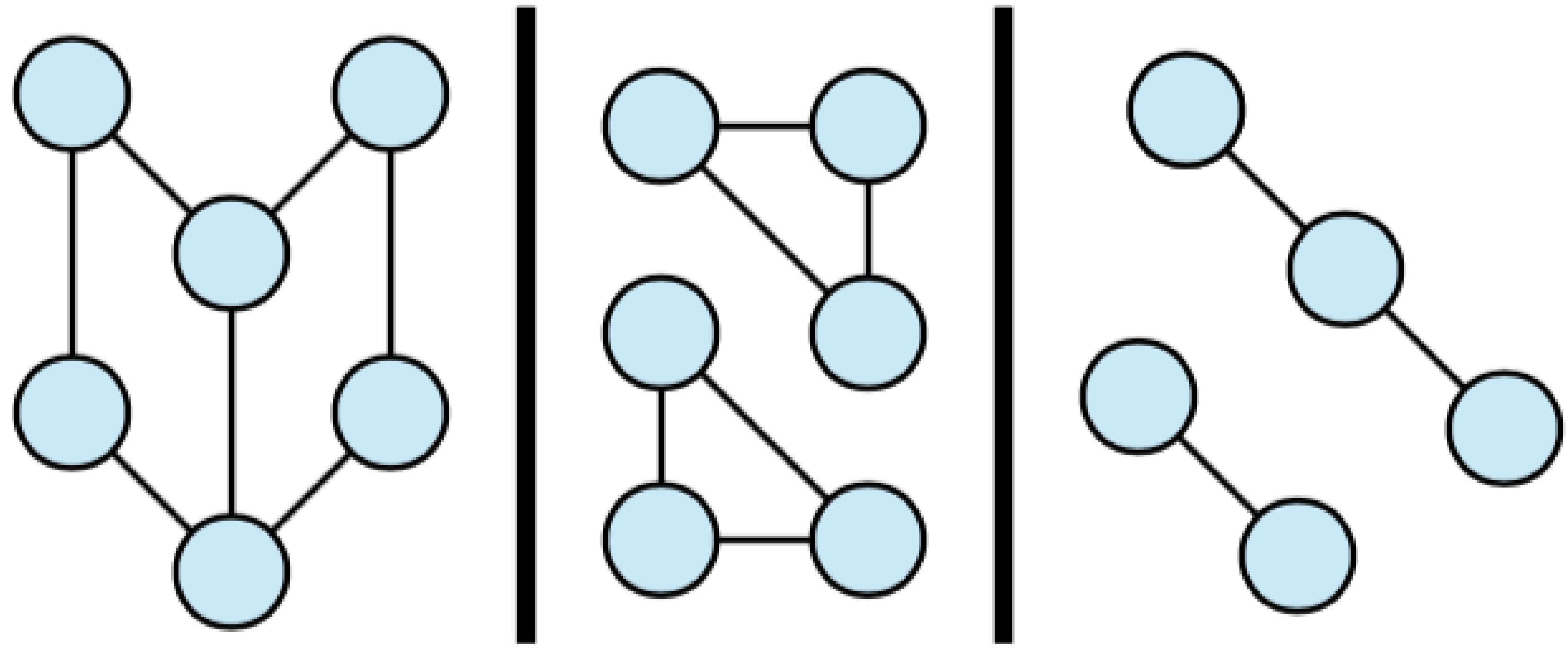


G1 seems like it is one big piece.

G2 and G3 are in multiple pieces.



Knowing that G = (V, E), and what it means for two nodes to be connected, can you formulate a definition for connected c omponent of G?

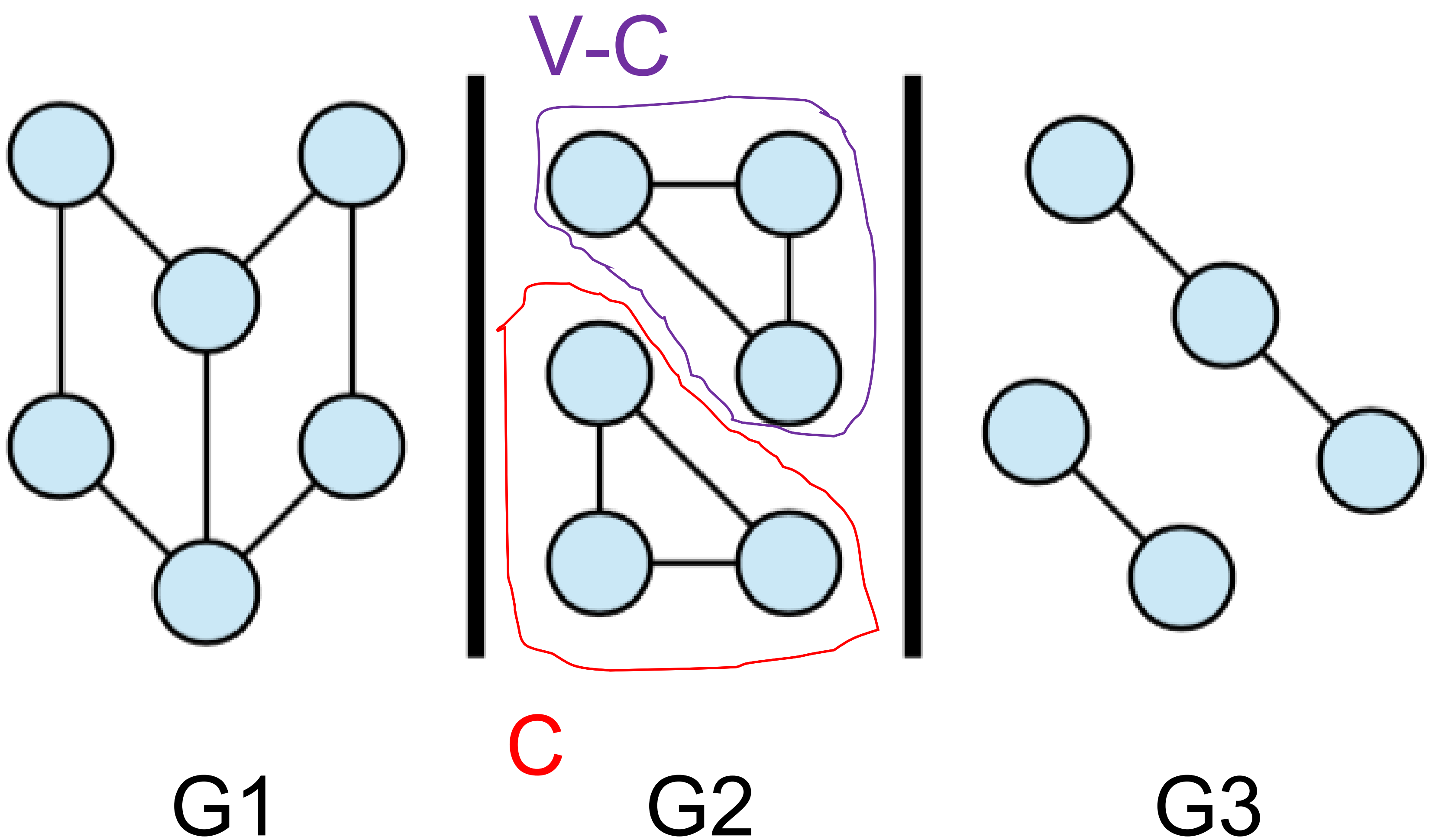


G1 G2 G3

Let $G = (V, E)$ be an undirected graph. A connected component of G is a nonempty set of nodes C (that is, $C \subseteq V$), such that

(1) For any $u, v \in C$, we have $u \leftrightarrow v$.

(2) For any $u \in C$ and $v \in V - C$, we have $u \,!\!\leftrightarrow v$



Let G = (V, E) be an undirected graph. A connected component of G is a nonempty set of nodes C (that is, $C \subseteq V$), such that

(1) For any $u, v \in C$, we have $u \leftrightarrow v$.

(2) For any $u \in C$ and $v \in V - C$, we have $u \not\leftrightarrow v$

Graph Traversals

Traversing a Graph

- Visit every edge and vertex in a systematic way
- Why do this?

"One of the fundamental operations in a graph is finding vertices that can be reached from a specified vertex."

For example, imagine trying to find out how many cities in Russia can be reached by a passenger train from Kazan

Traversing a Graph

- There are two ways to traverse a graph:

  Depth-First Search (DFS)

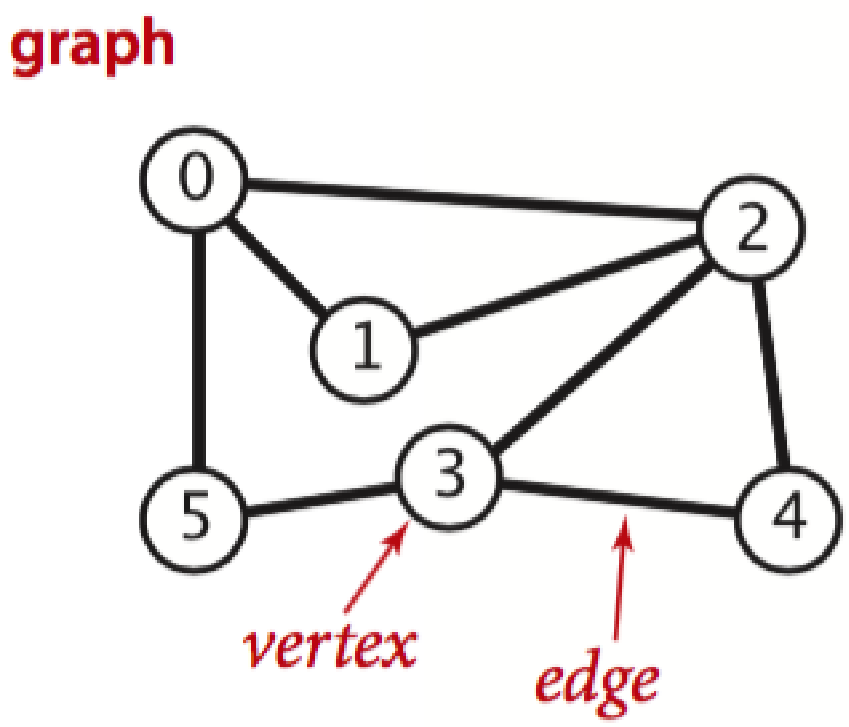
  Breadth-First Search (BFS)

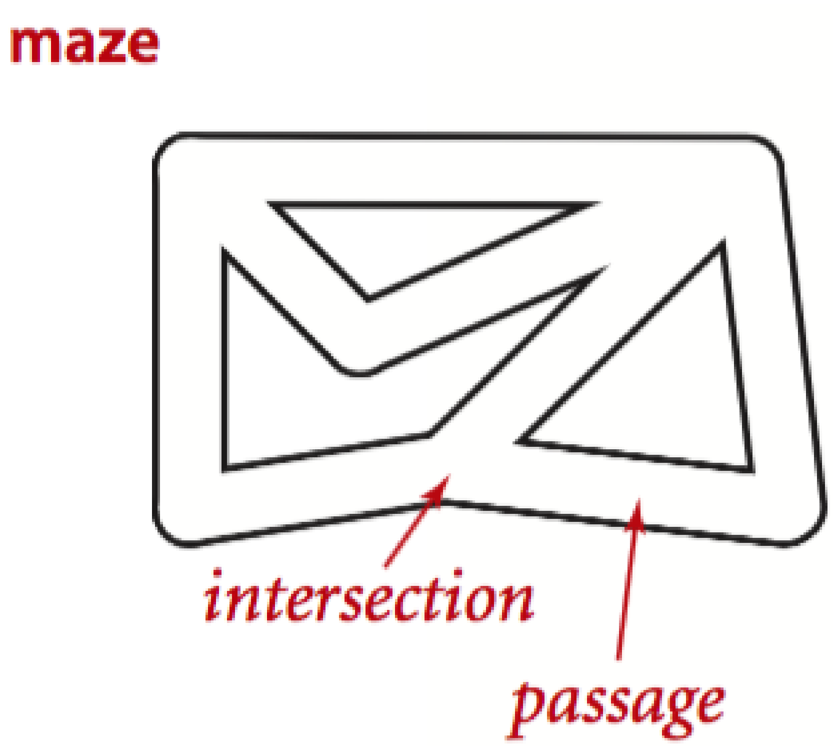
- Both will eventually reach all connected nodes
- The difference is

  DFS uses a stack

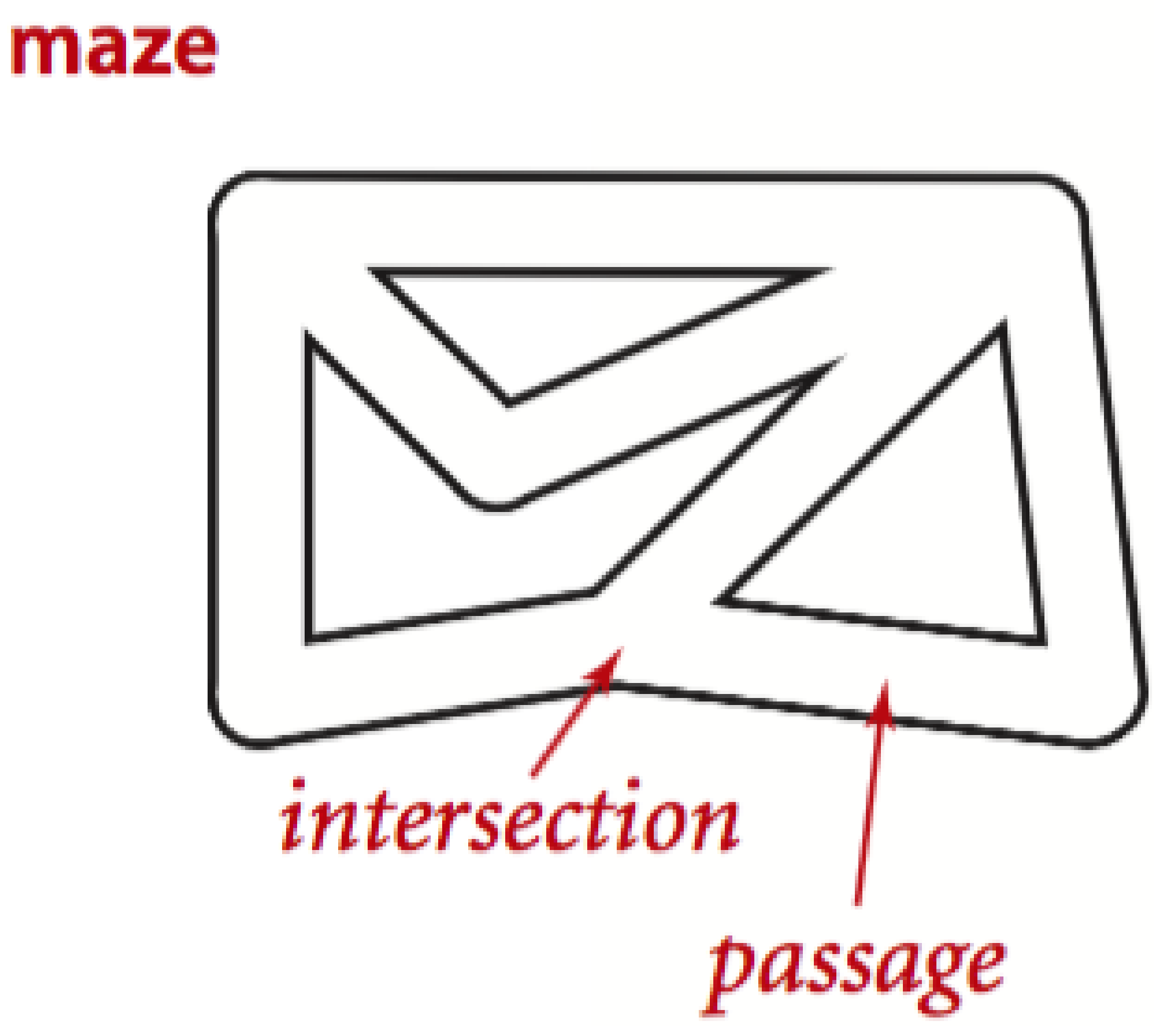
  BFS uses a queue

Searching in a Maze

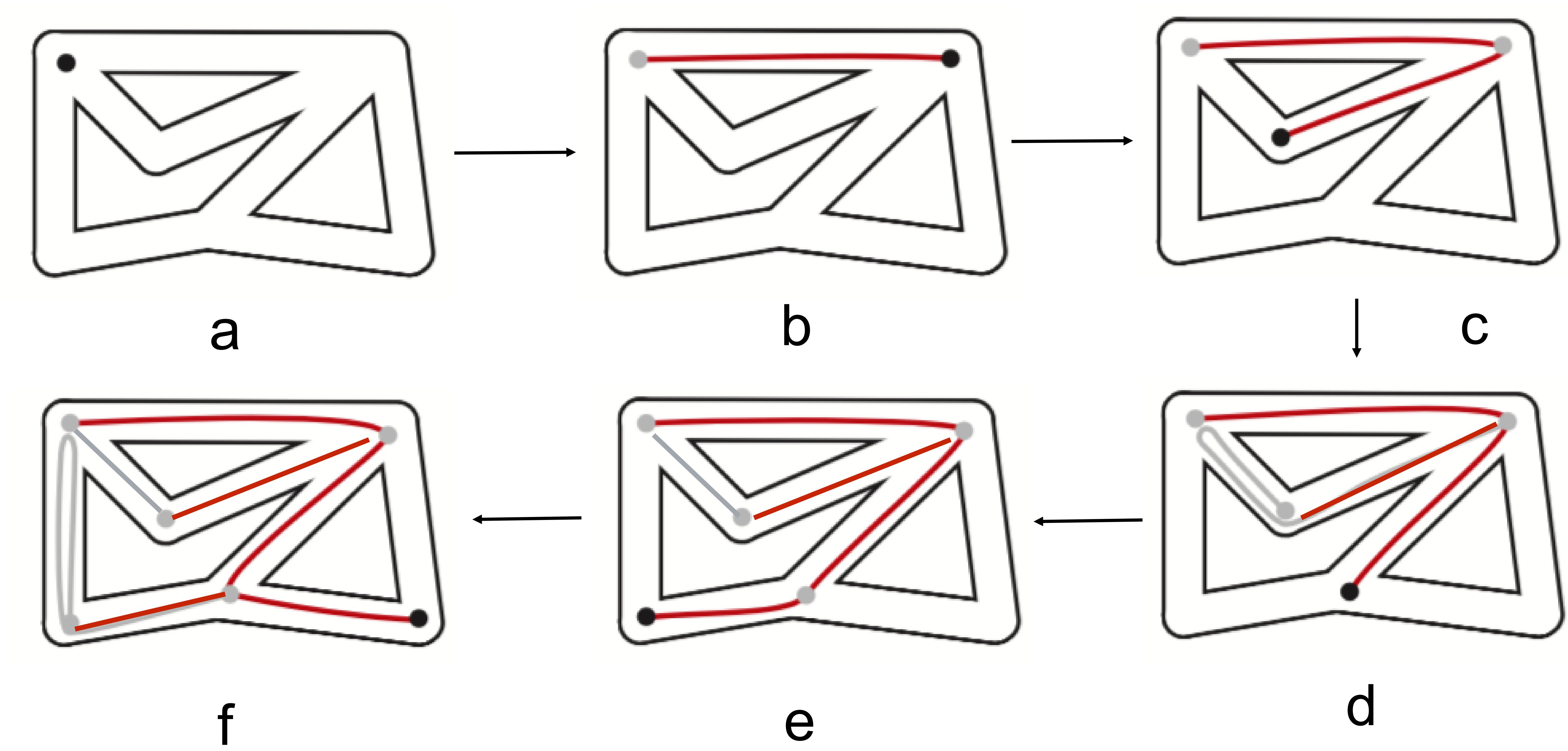




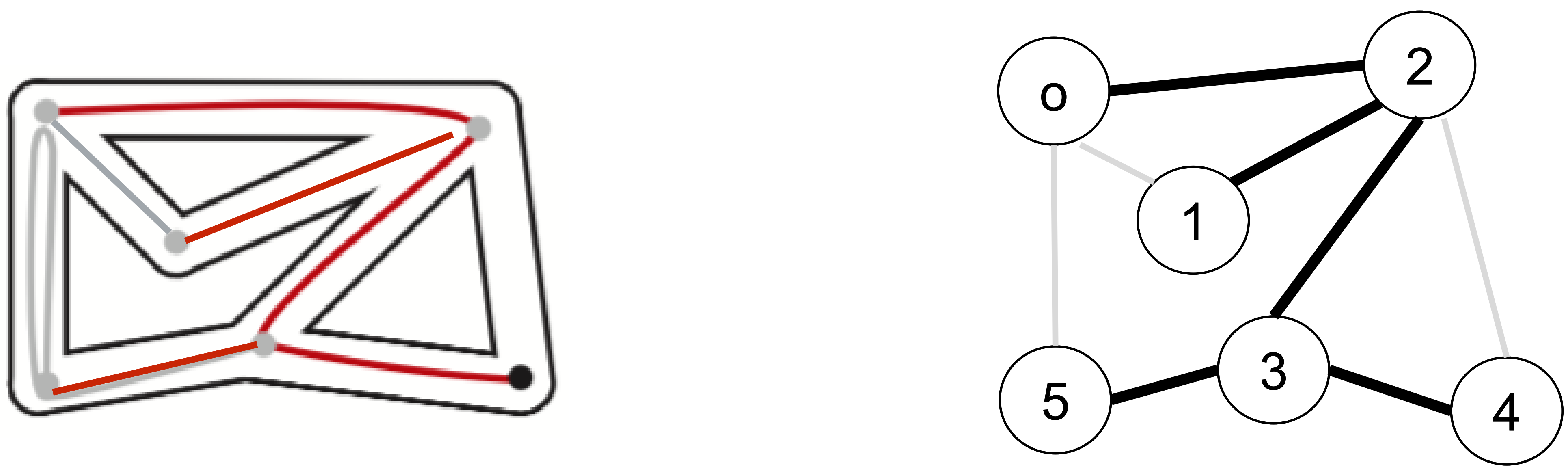
Searching in a Maze



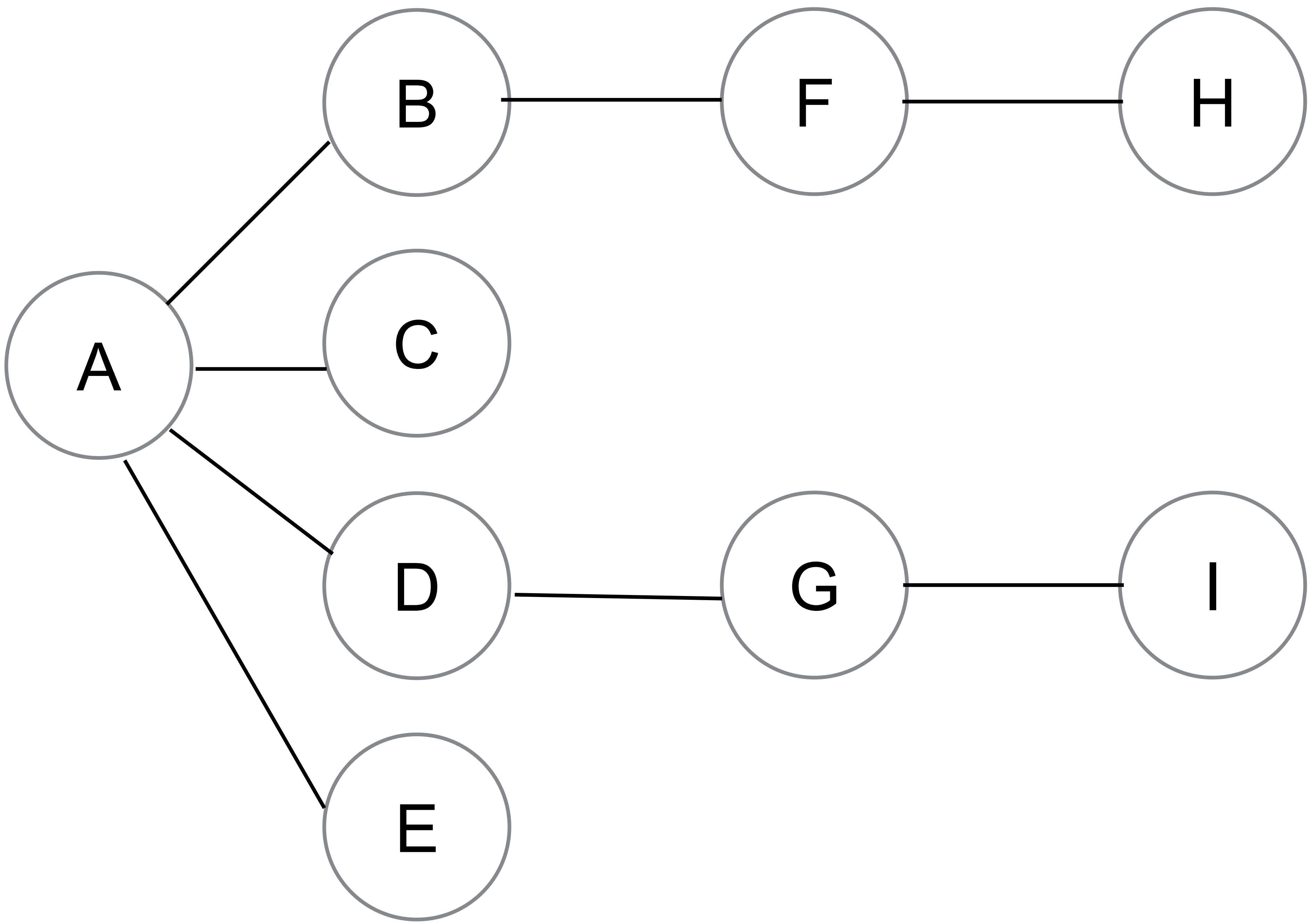
Searching in a Maze



Maze vs. Graph



DFS with a Stack



DFS with a Stack (2)

- Pick a starting point - in this case vertex A, and do three things

  1. visit this vertex
  2. push it on a stack
  3. mark it visited (so you won't visit it again)

DFS with a Stack (3)

- Pick a starting point - in this case vertex A, and do three things

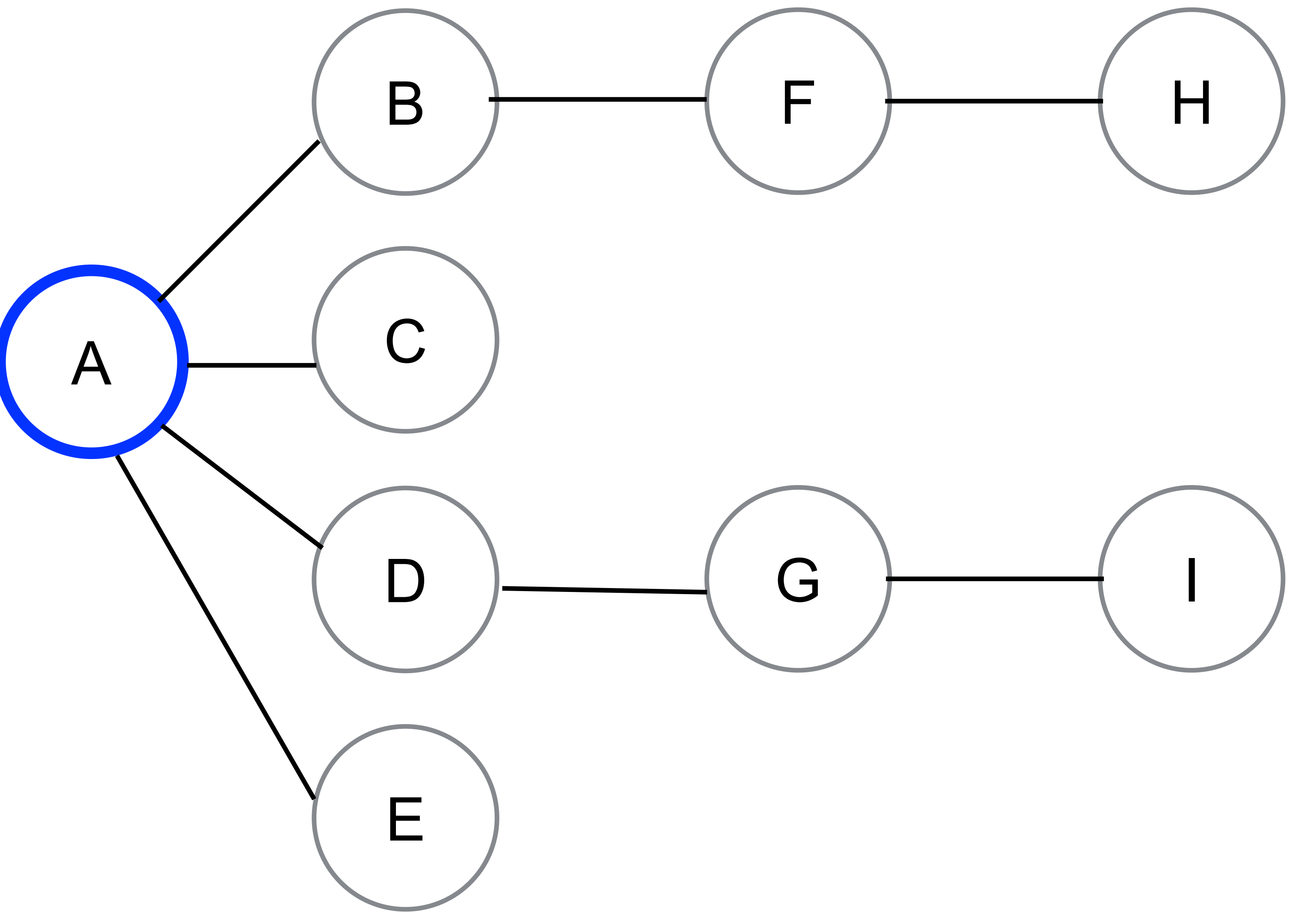
  1. visit this vertex

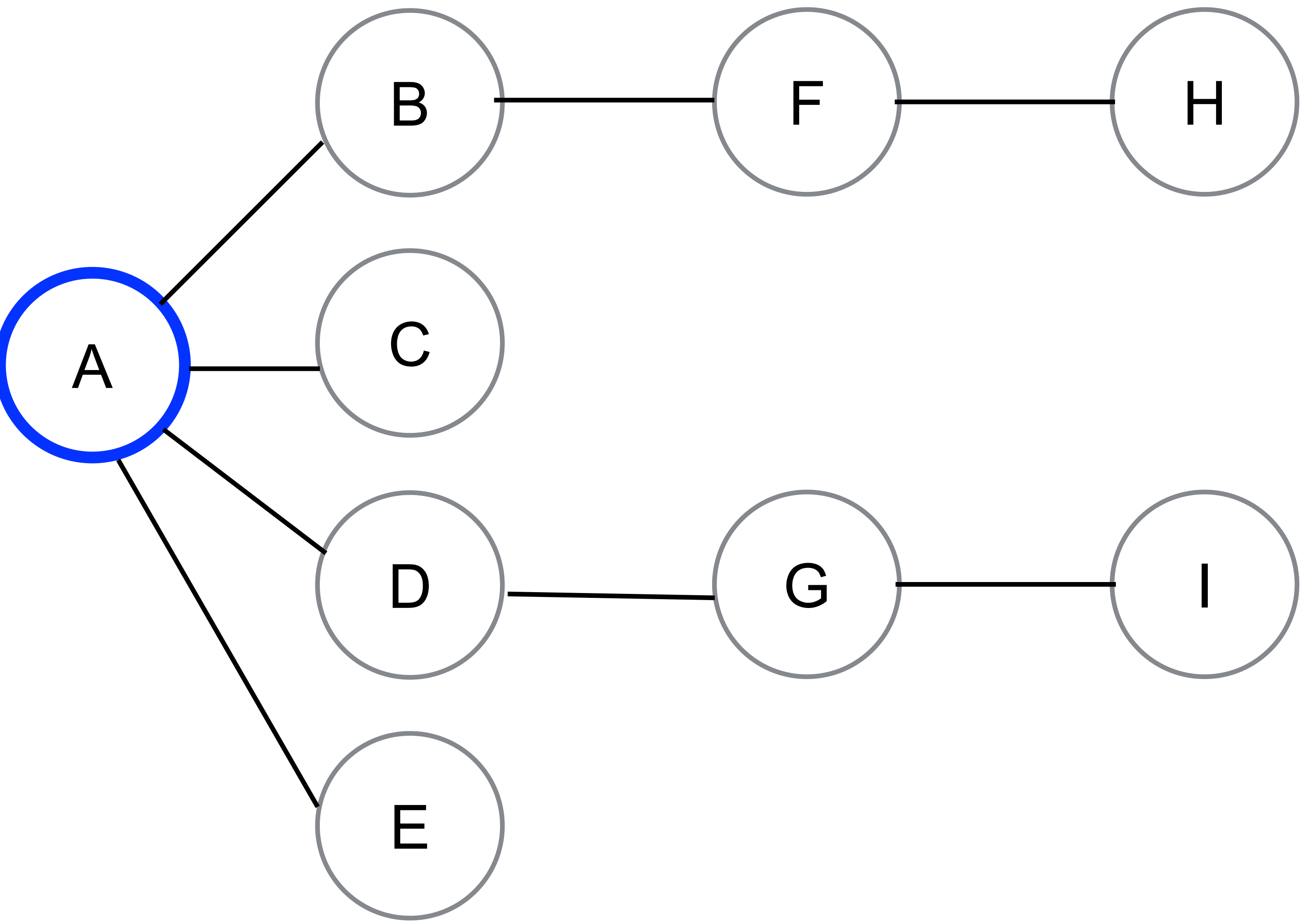
  2. push it on a stack

  3. mark it visited (so you won't visit it again)

Visit is abstract, just like BST

How can you mark a vertex as visited?


A
B
C
D
E
F
G
H
I

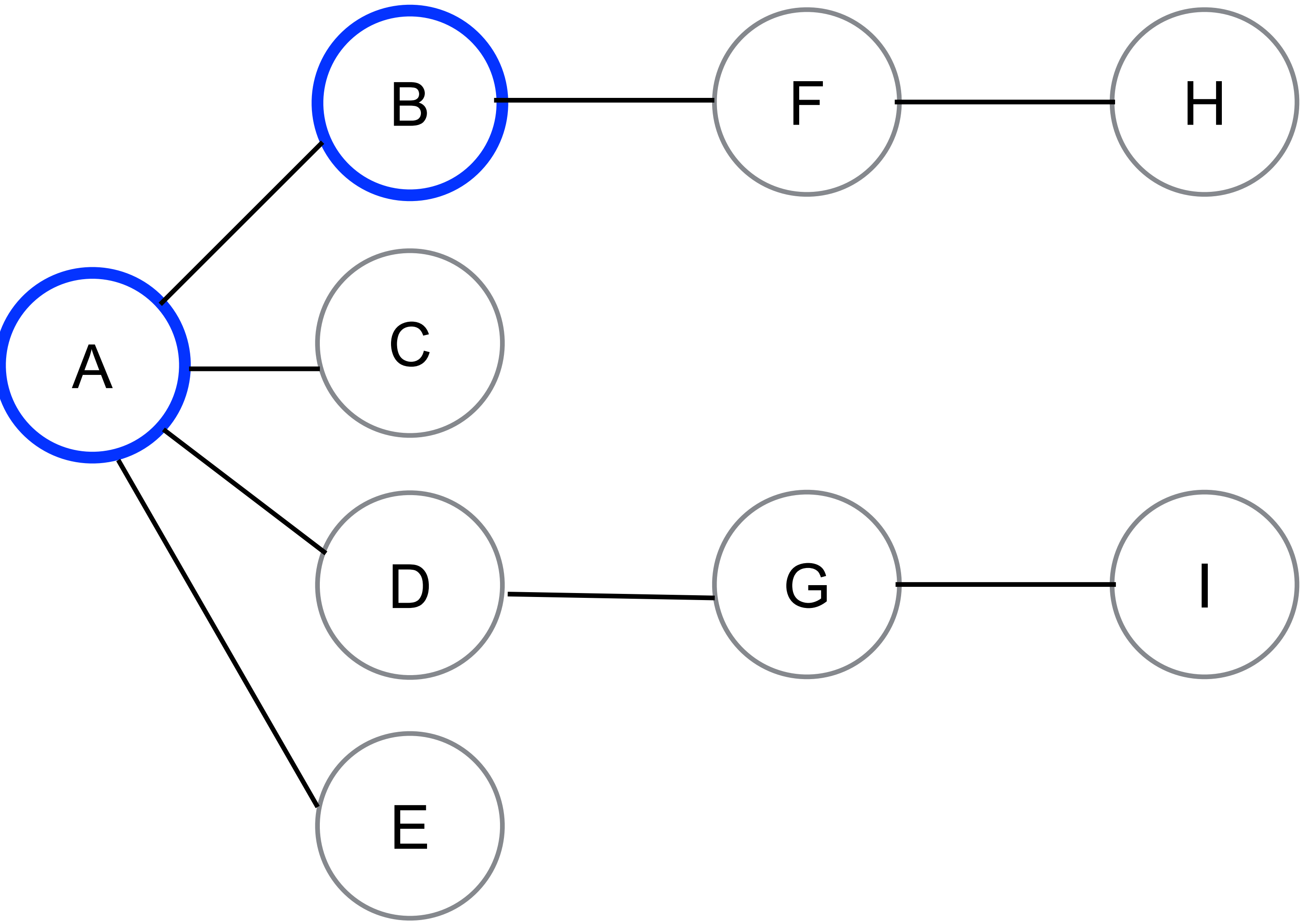


Next, go to a vertex adjacent to A, which hasn’t been yet visited

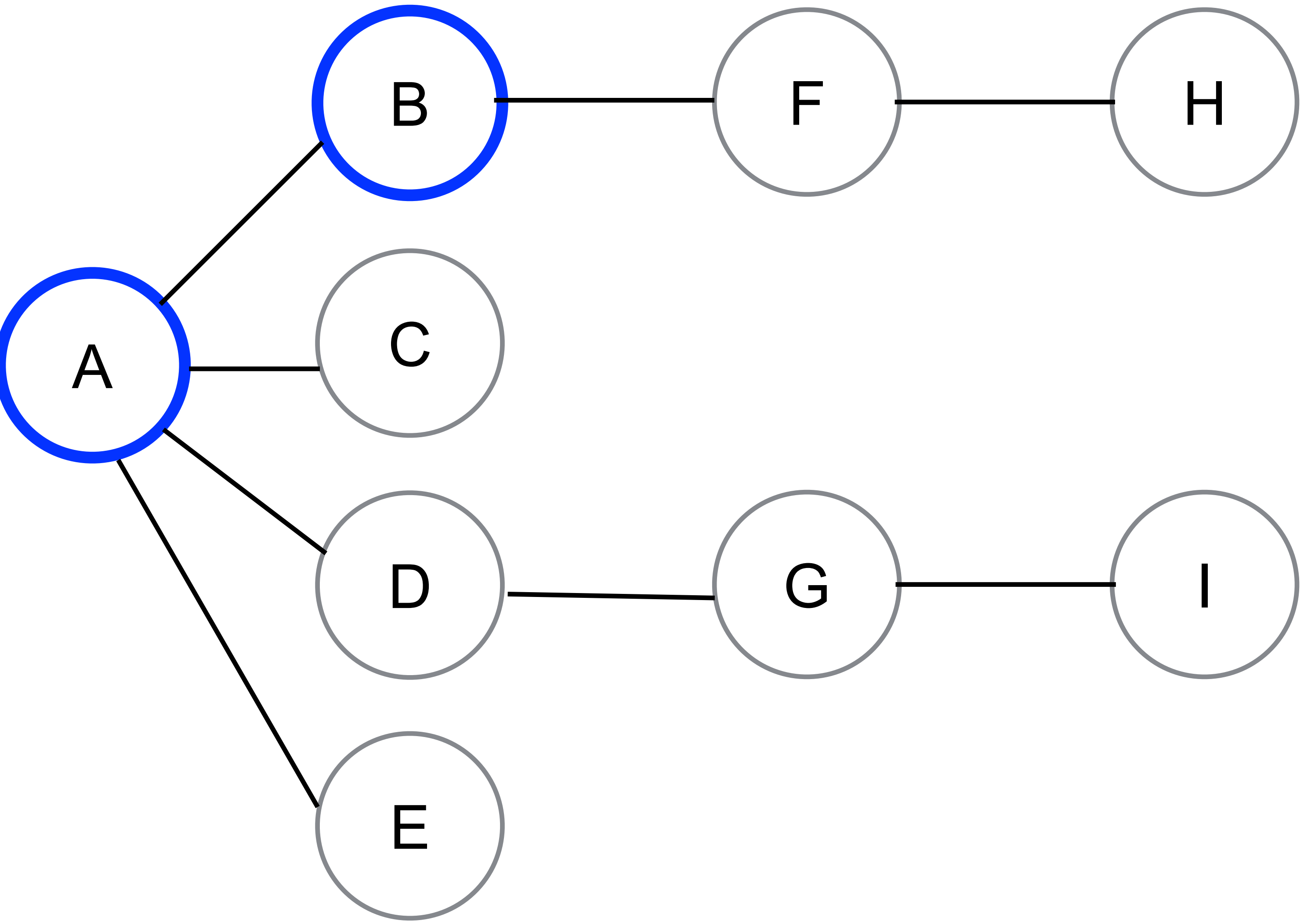
For this example, let’s go to B

Visit B, mark it, and push it on the stack
Let’s call this **Rule 1:**

“If possible, visit an unvisited adjacent vertex, mark it, and push it on the stack”

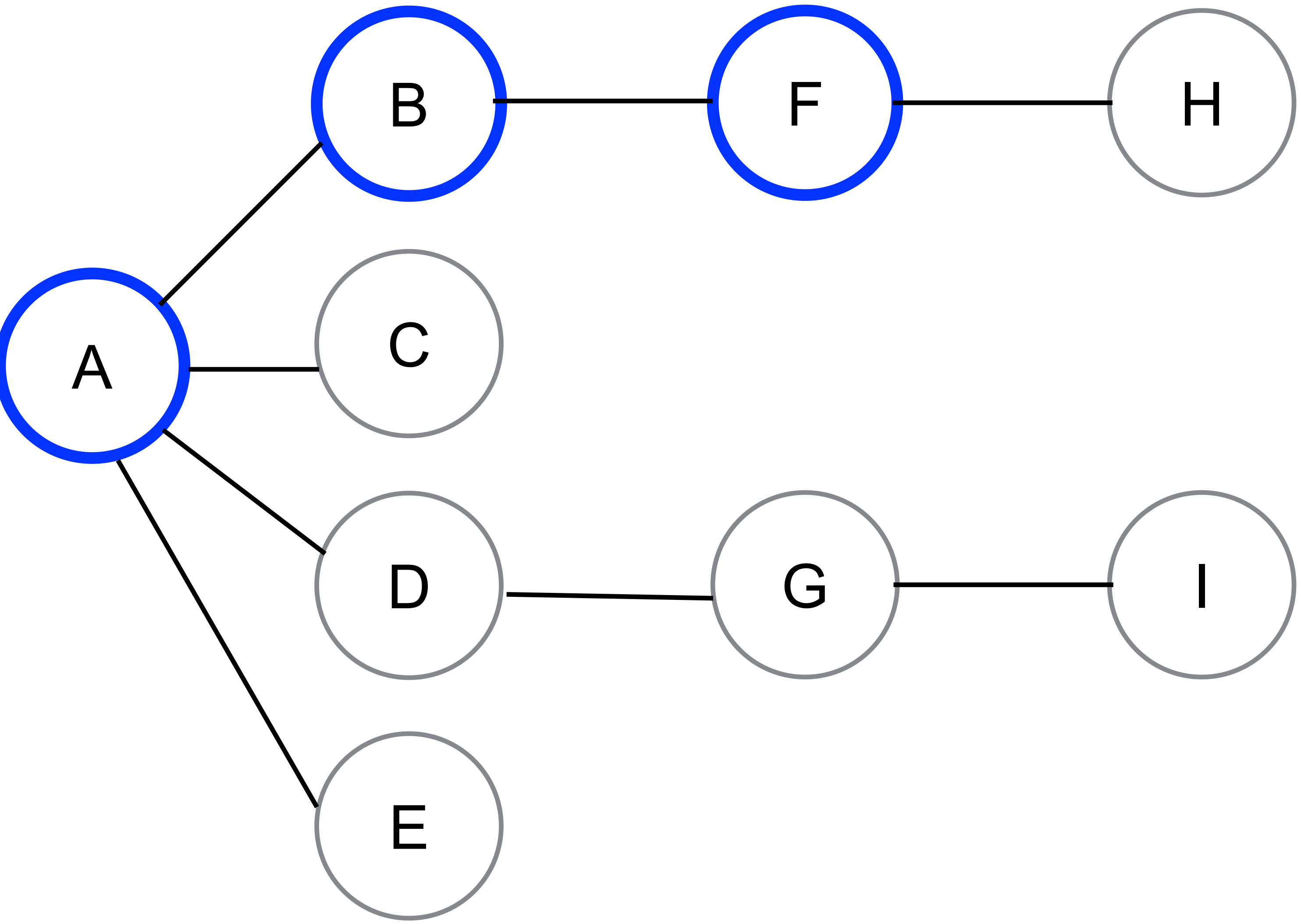


B
F
H
A
C
D
G
I
E

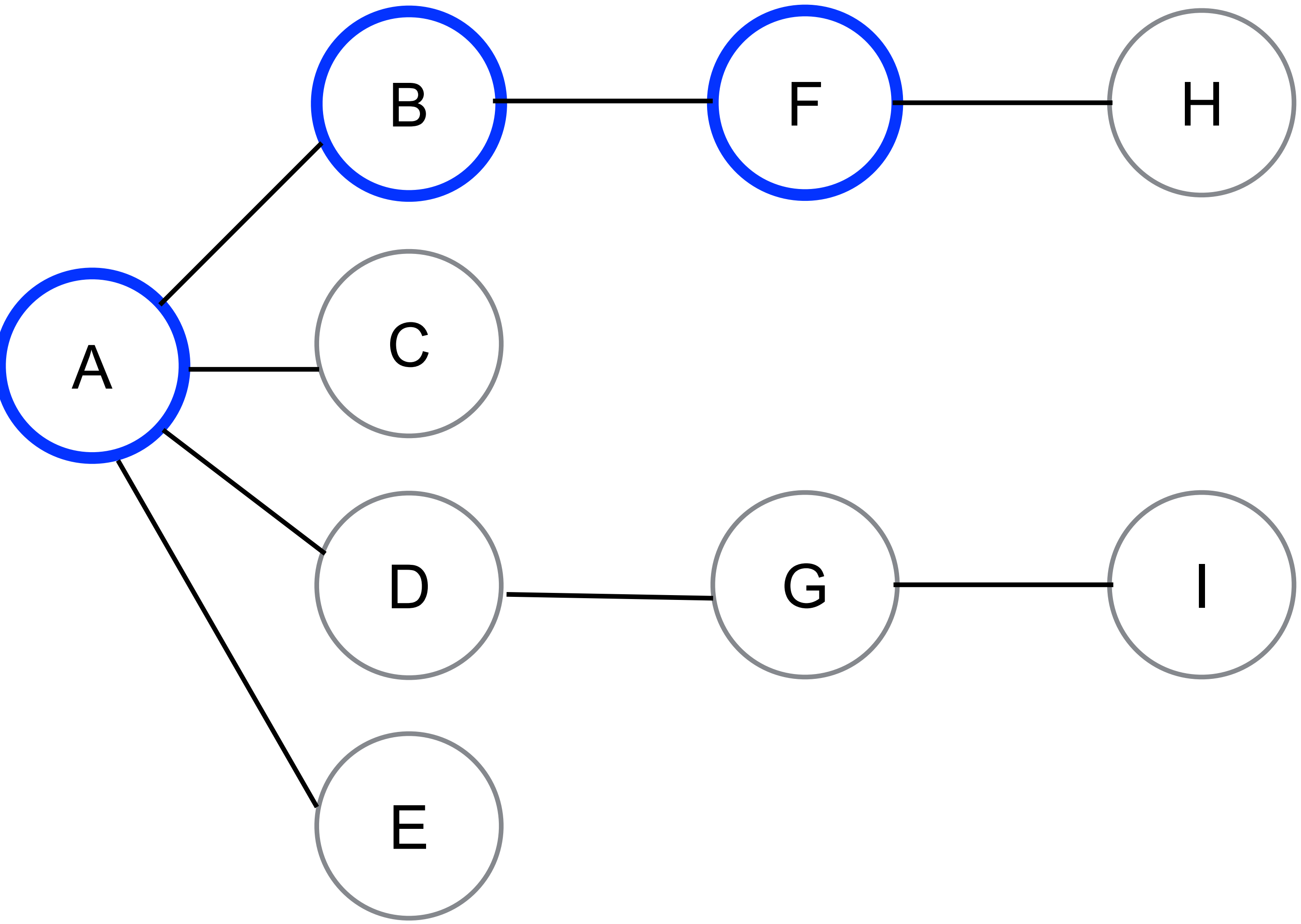


While at B, apply Rule 1 again.

“If possible, visit an unvisited adjacent vertex, mark it, and push it on the stack”

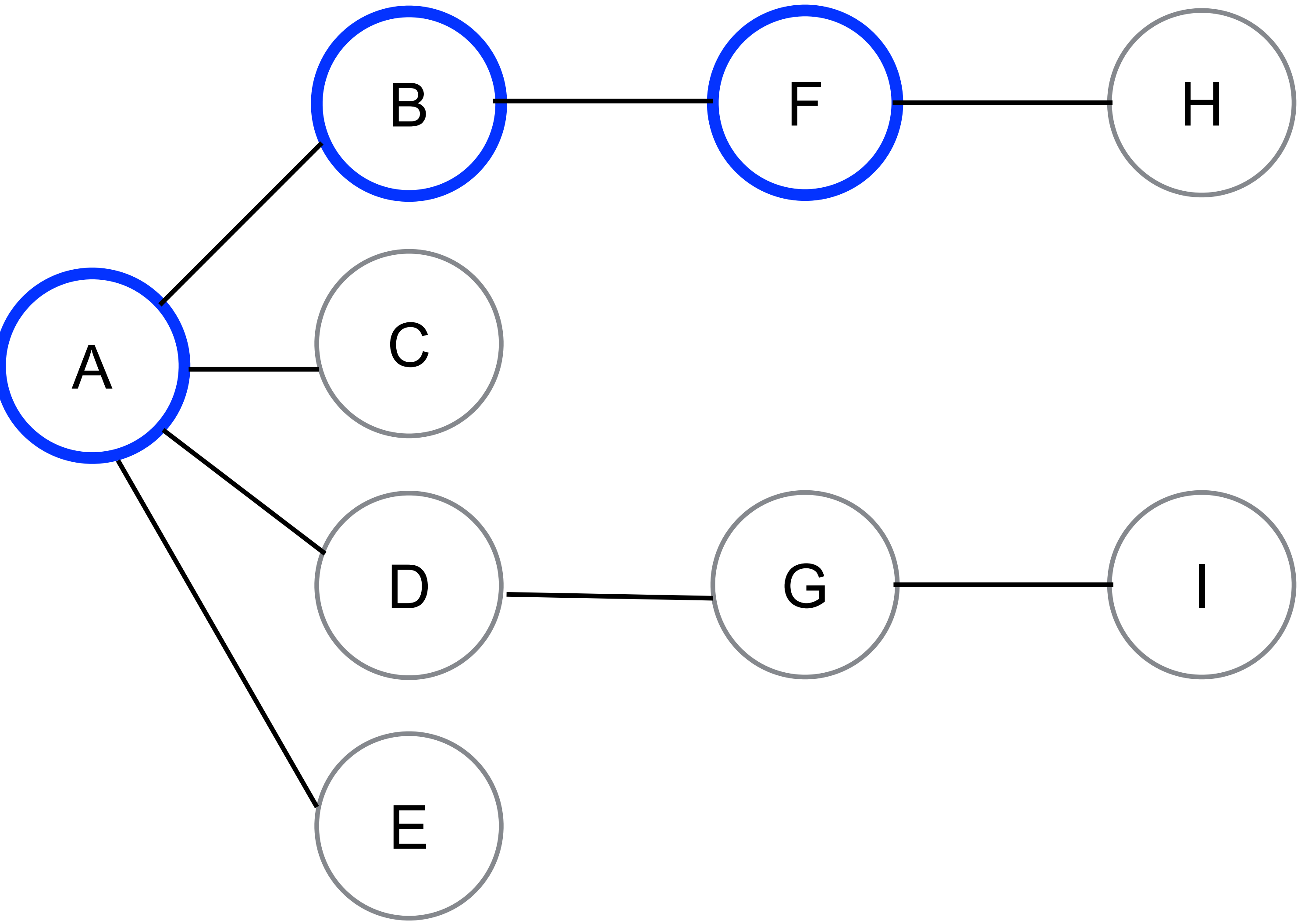

A
B
C
D
E
F
G
H
I

Two Important Questions

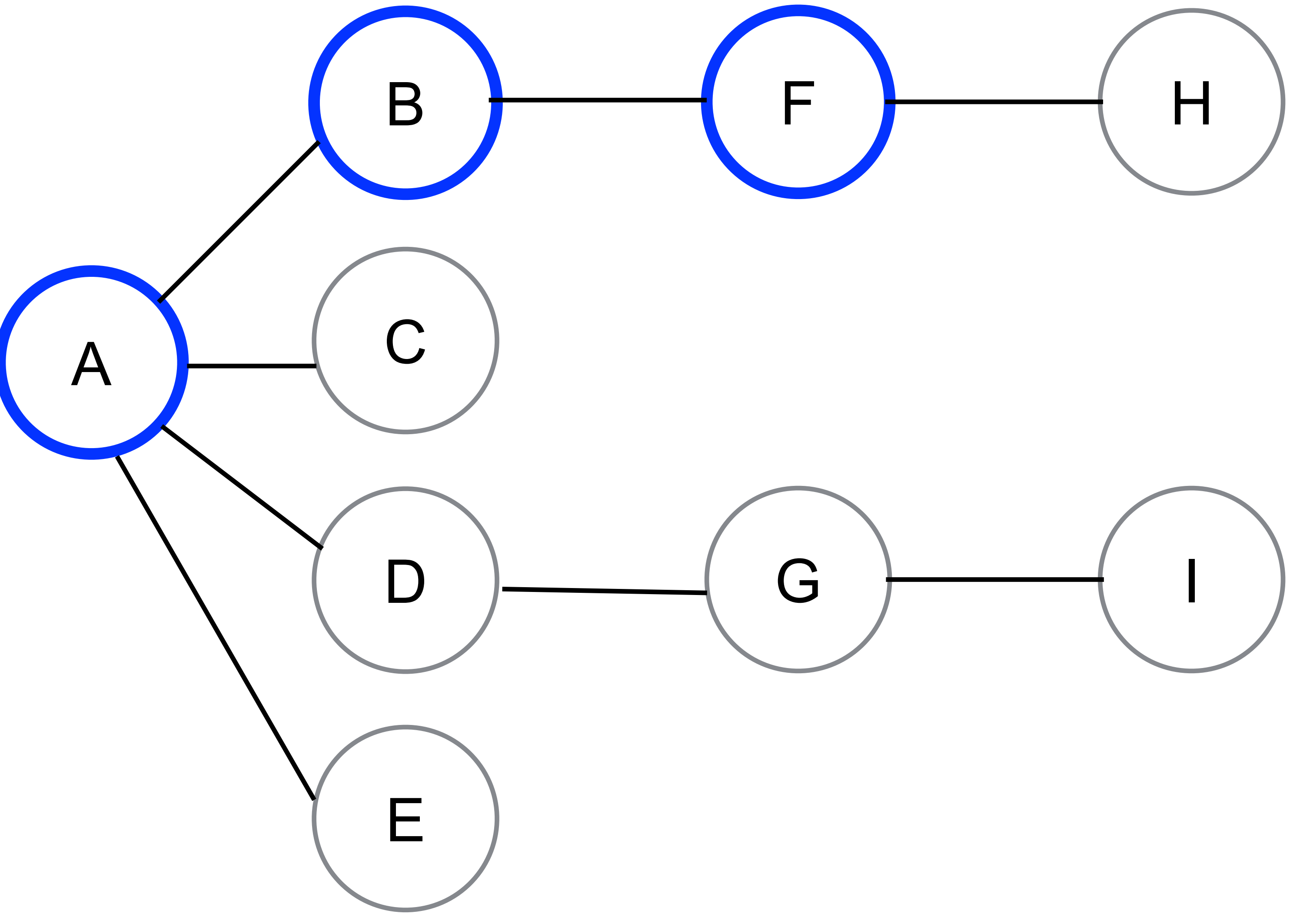


1. What if we had picked edge “BA”?

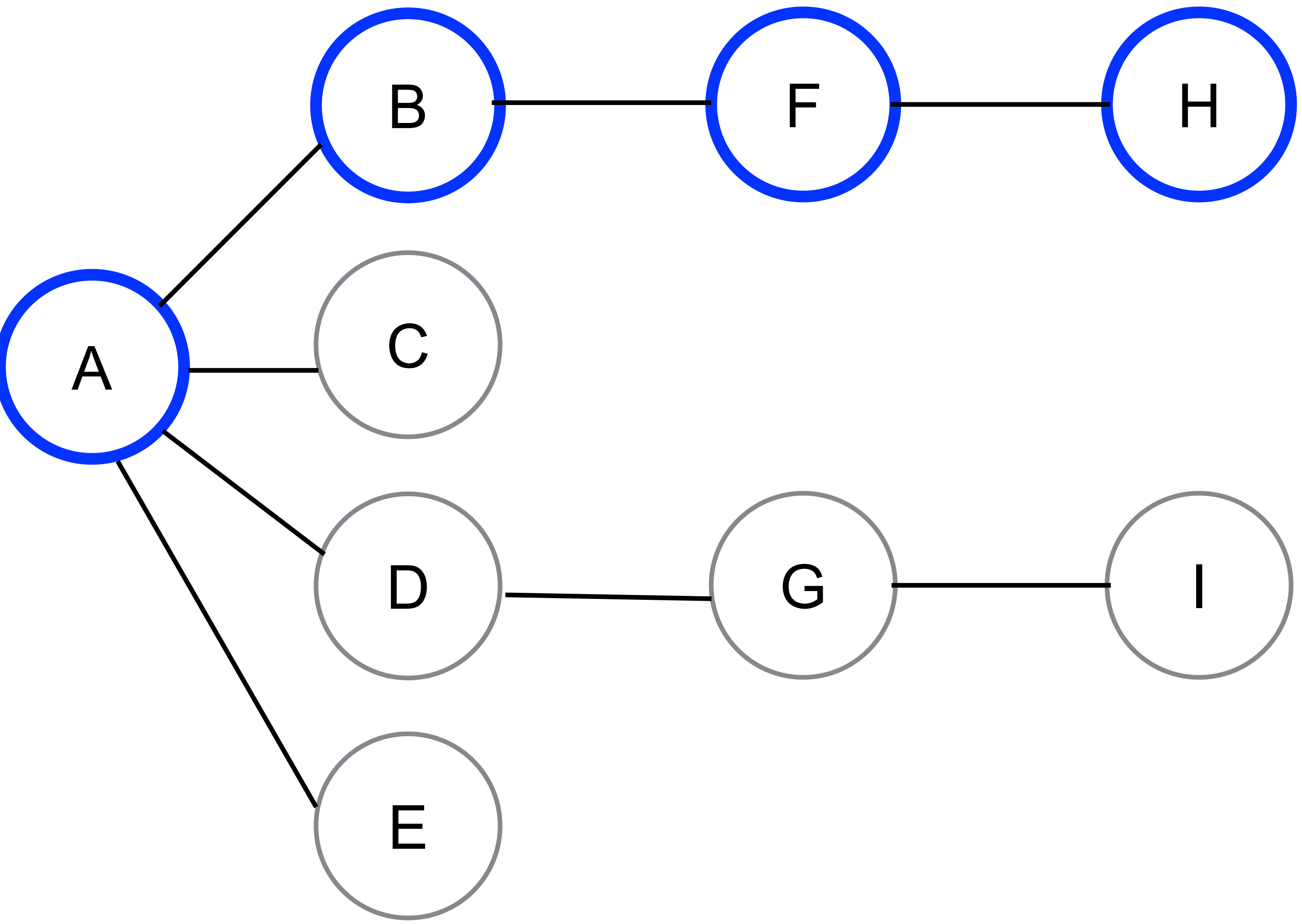
2. Will we move across the edge “BA” at some point d uring the traversal?



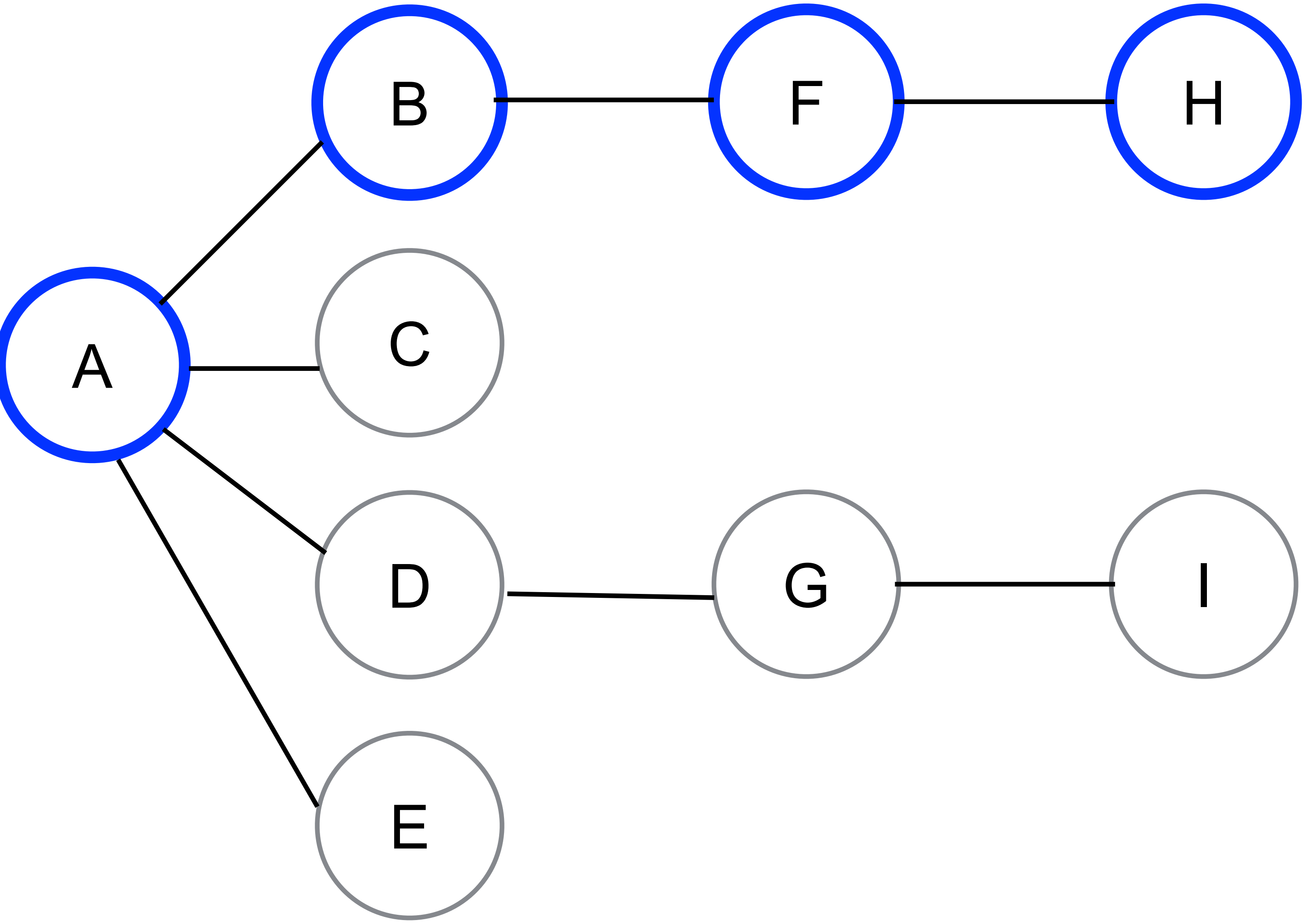
Thus you visit each vertex just once (put it in stack)
but you visit each edge twice!



While at F, apply Rule 1 again.



B
F
H
A
C
D
G
I
E

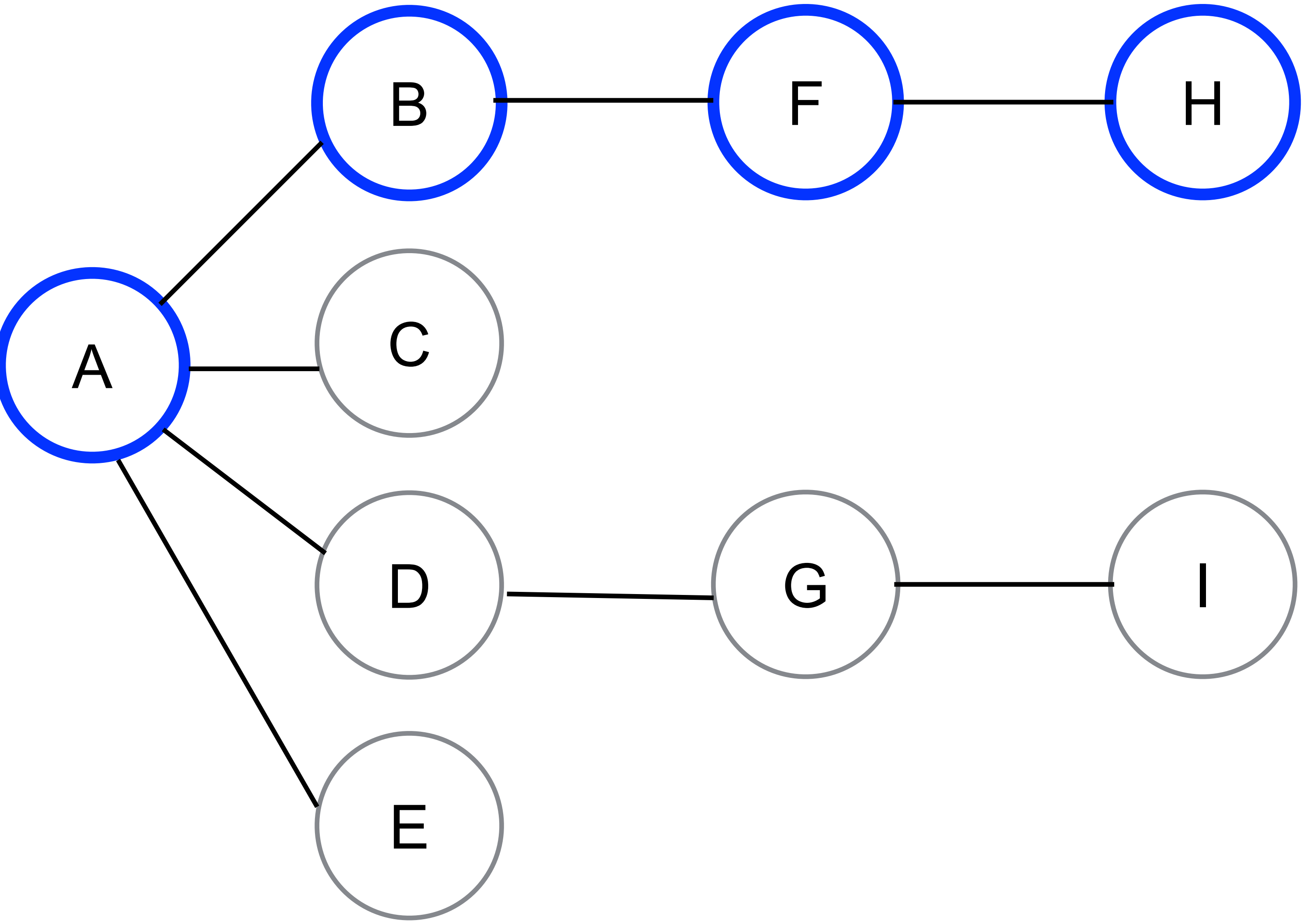


At this point (at H), there are no unvisited adjacent vertices (HF leads back to F)

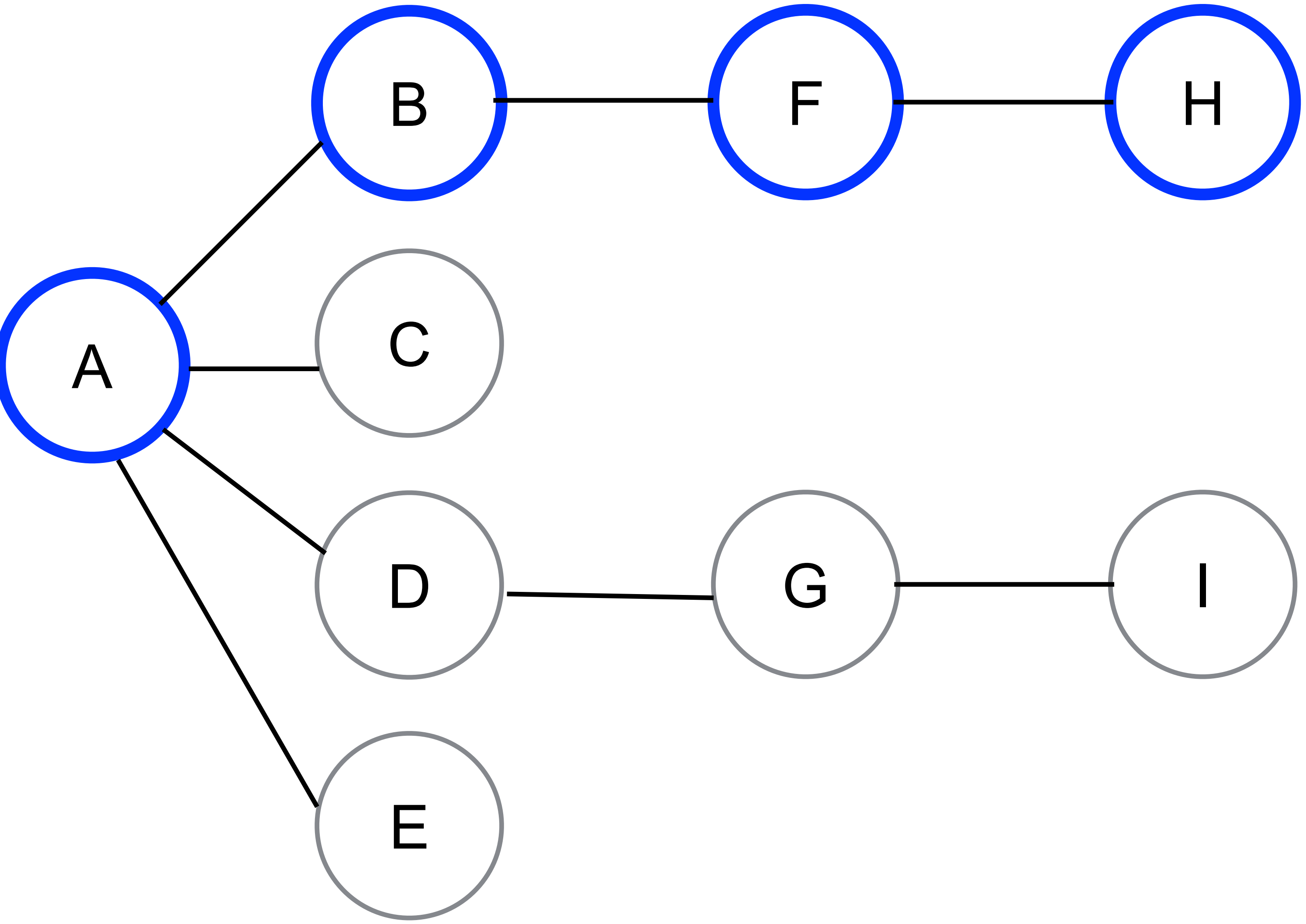
So we need to do something else

Rule 2:

"If you cannot follow Rule 1, then, if possible, pop a vertex off the stack"

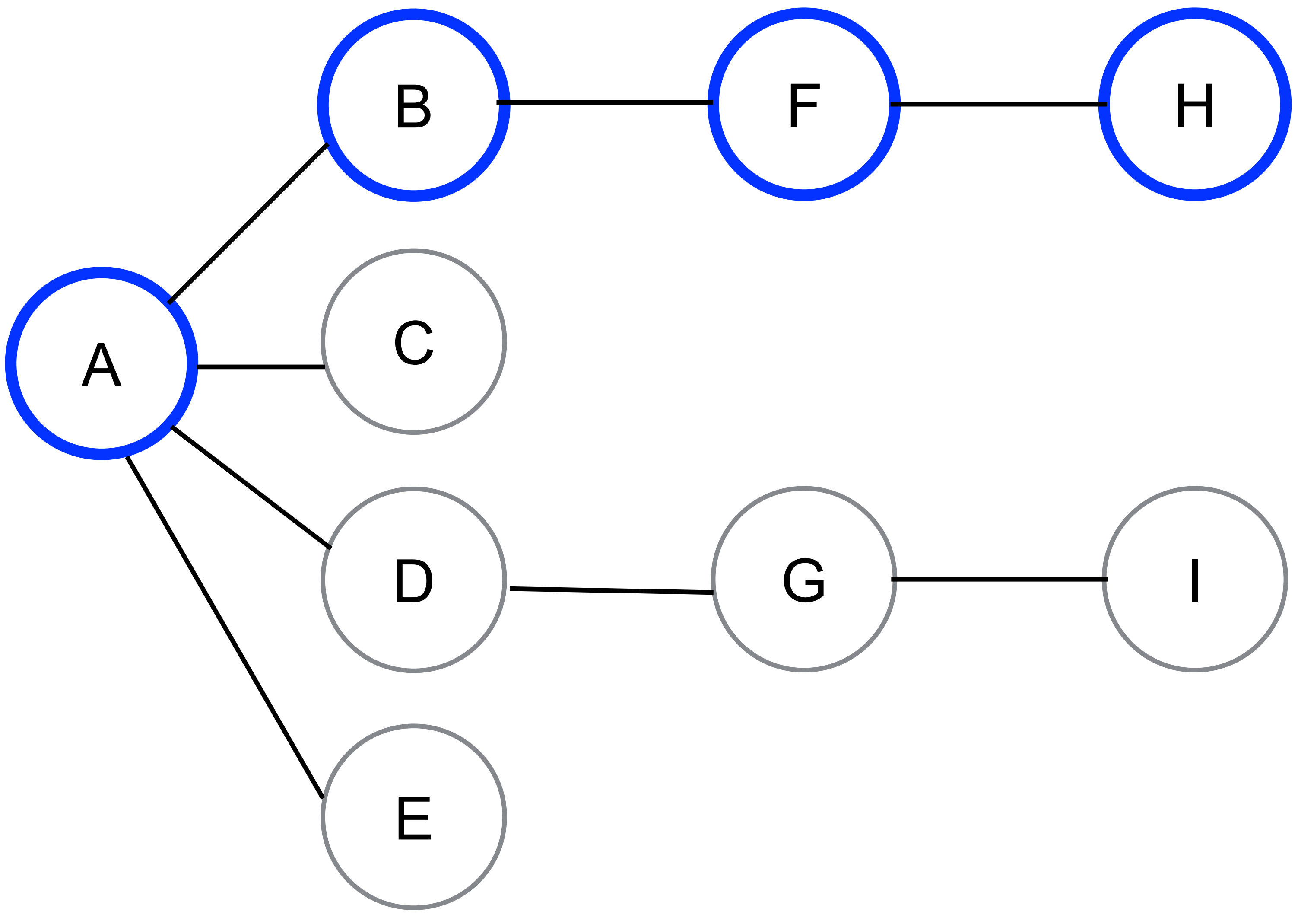


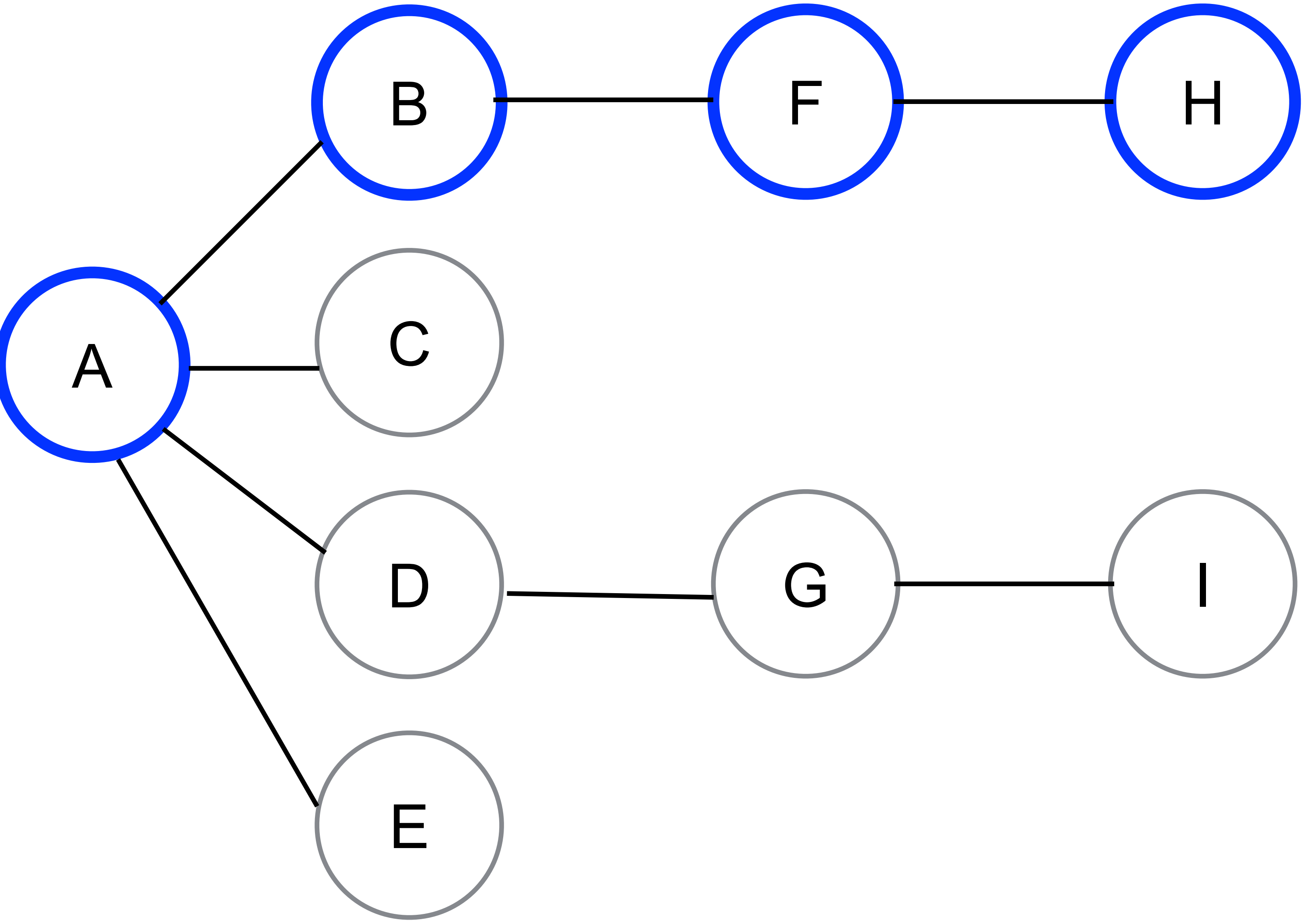
A
B
C
D
E
F
G
H
I



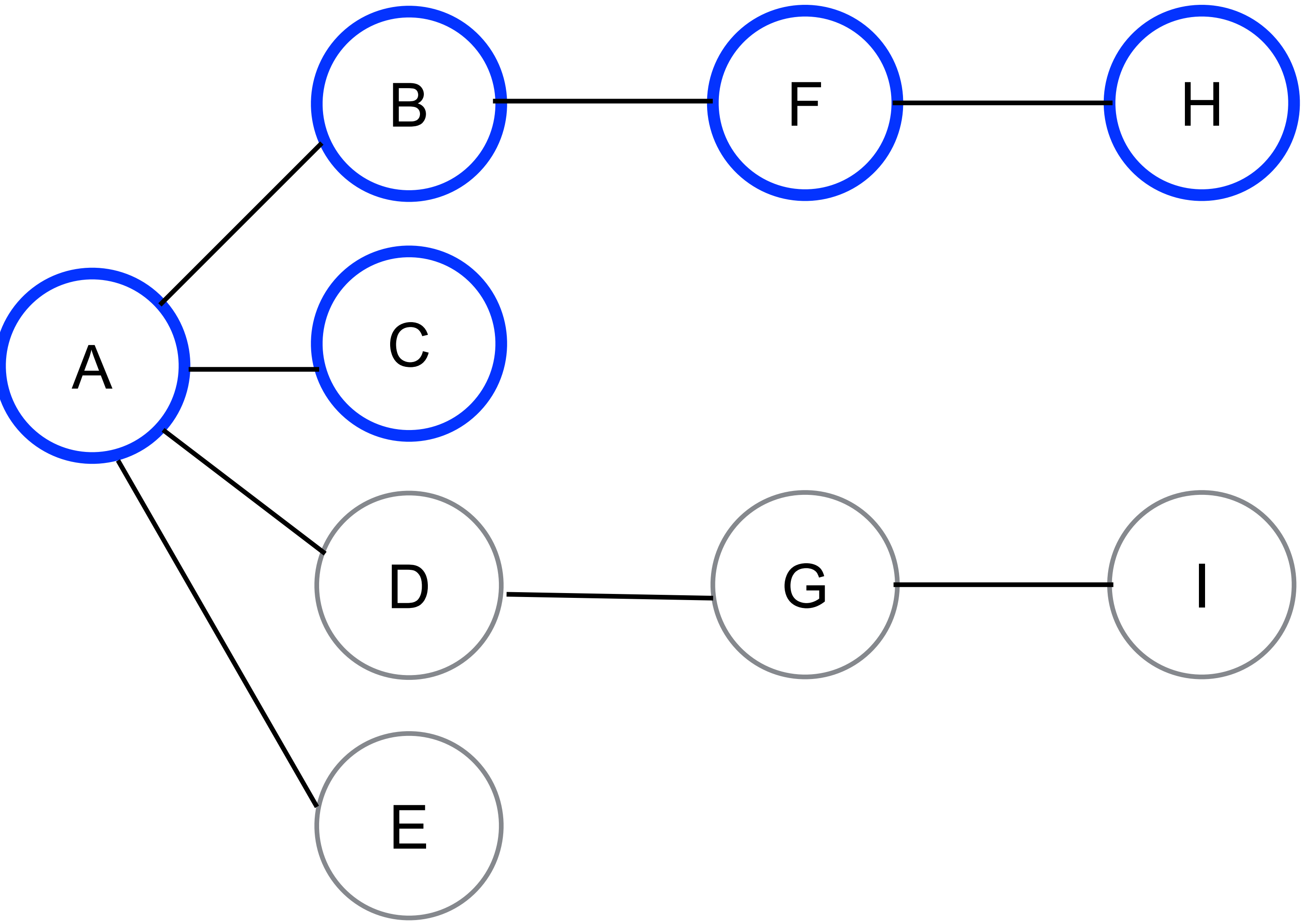
We are back at F

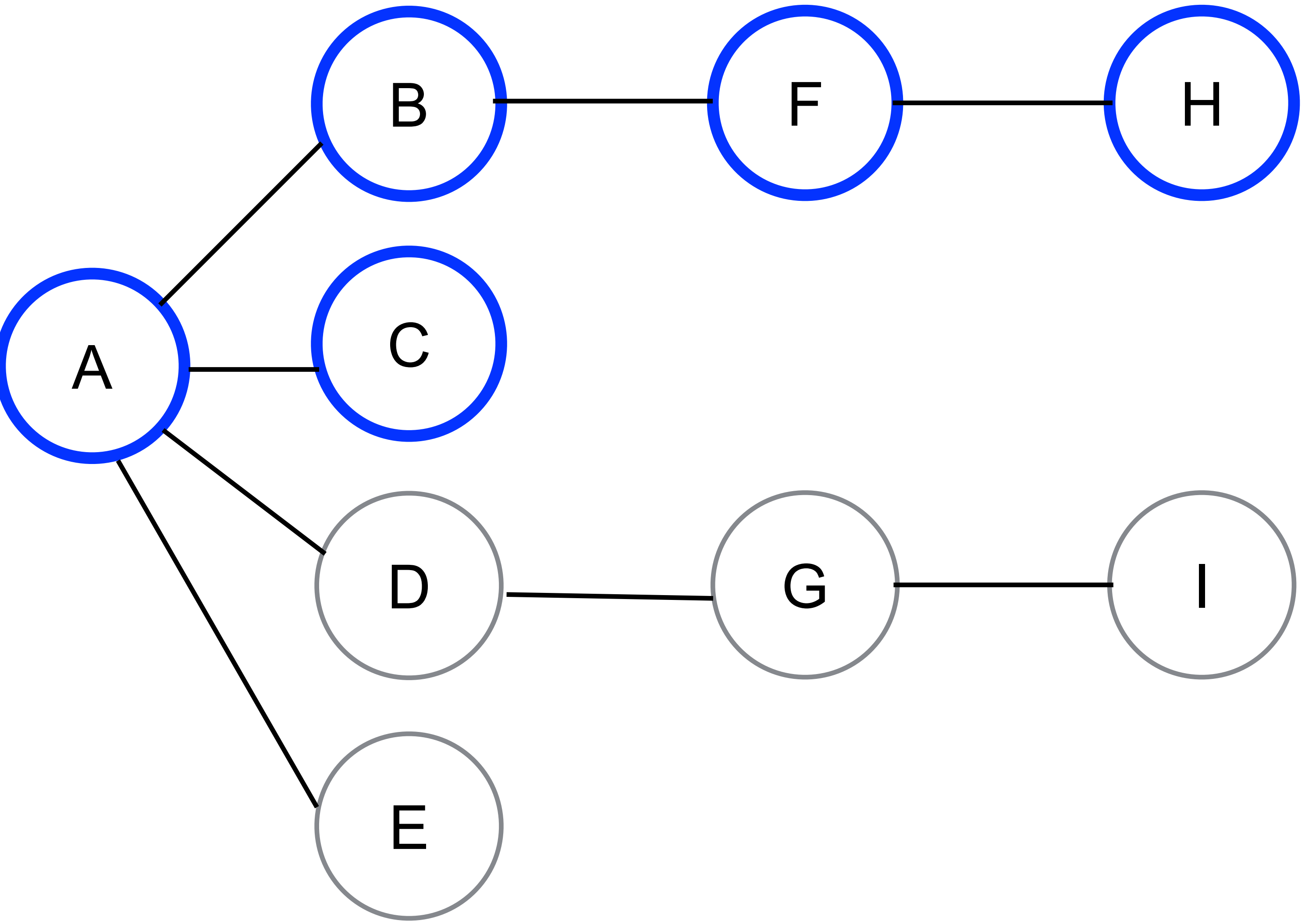
No more unvisited adjacent vertices, so pop it off, too

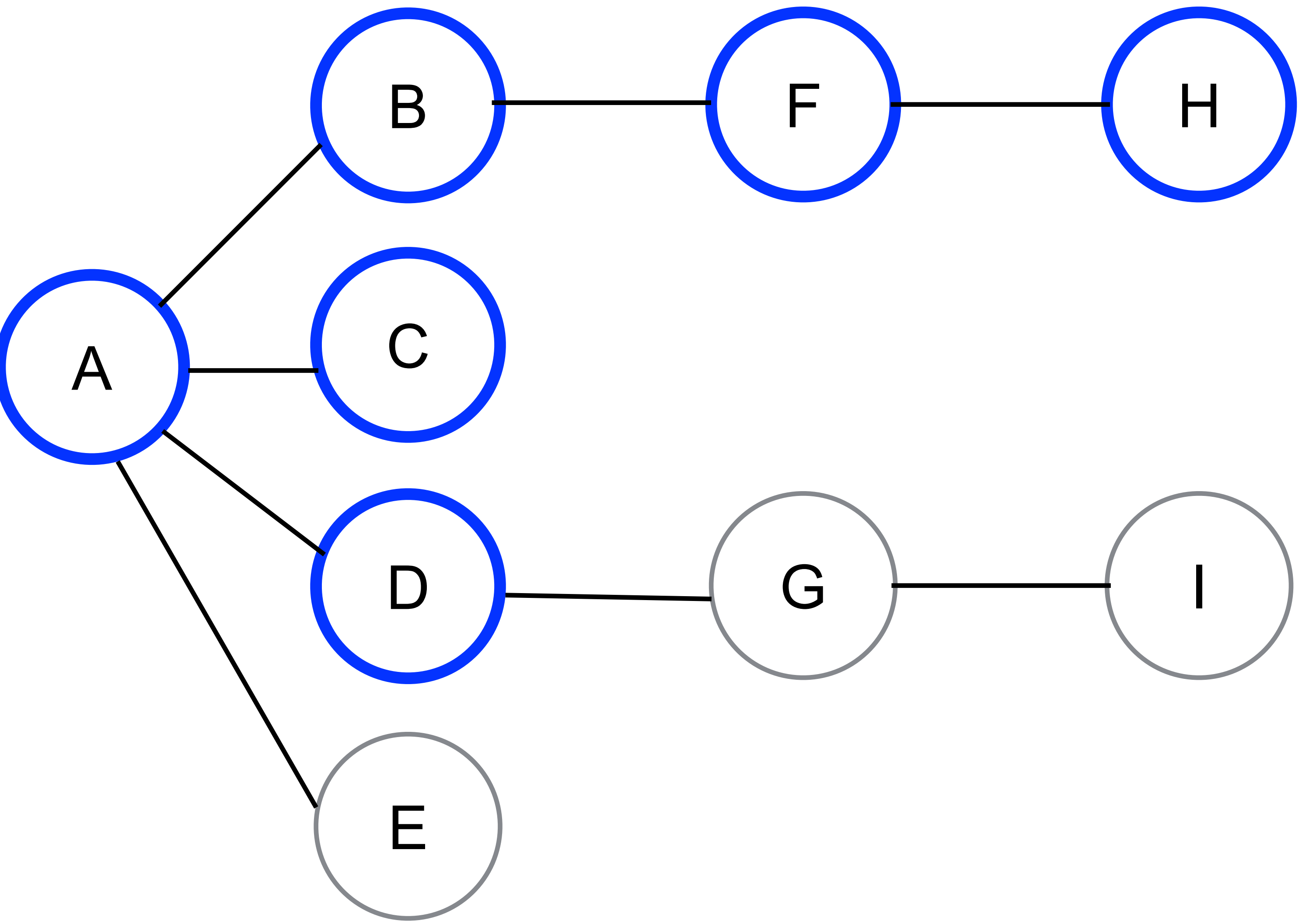

A
B
C
D
E
F
G
H
I

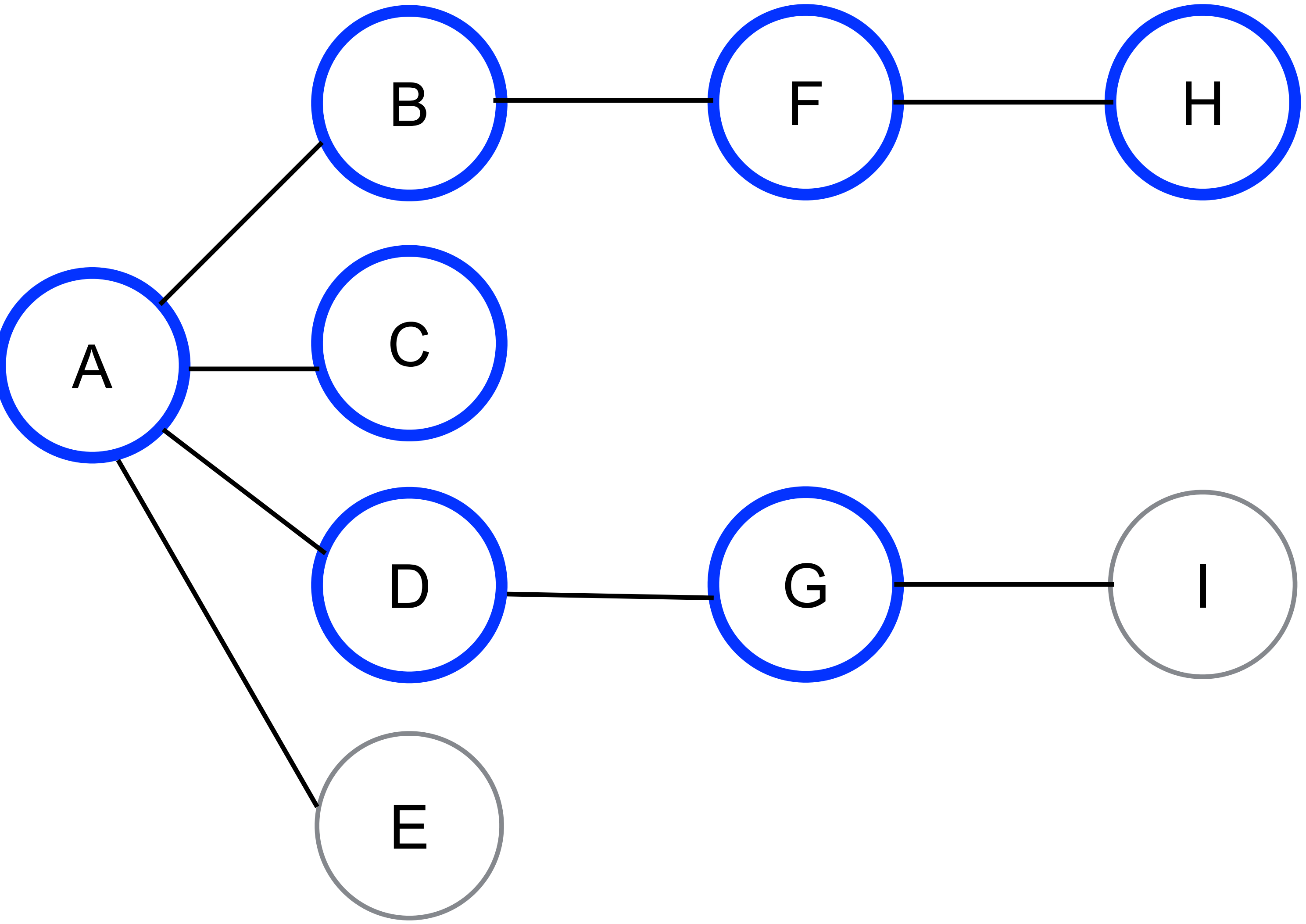


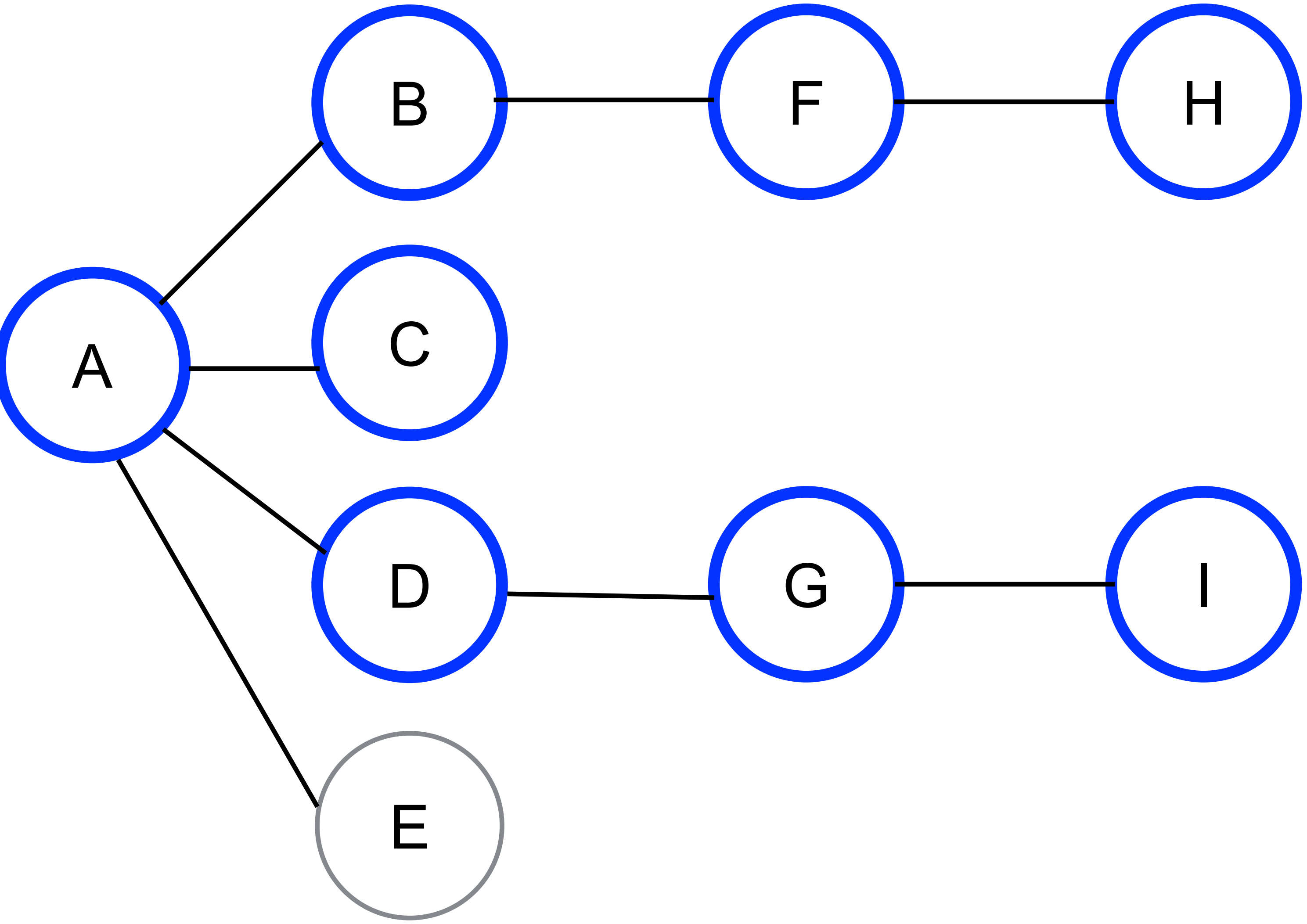
- We are back at A
- Pick the next adjacent vertex and repeat


A
B
C
D
E
F
G
H
I

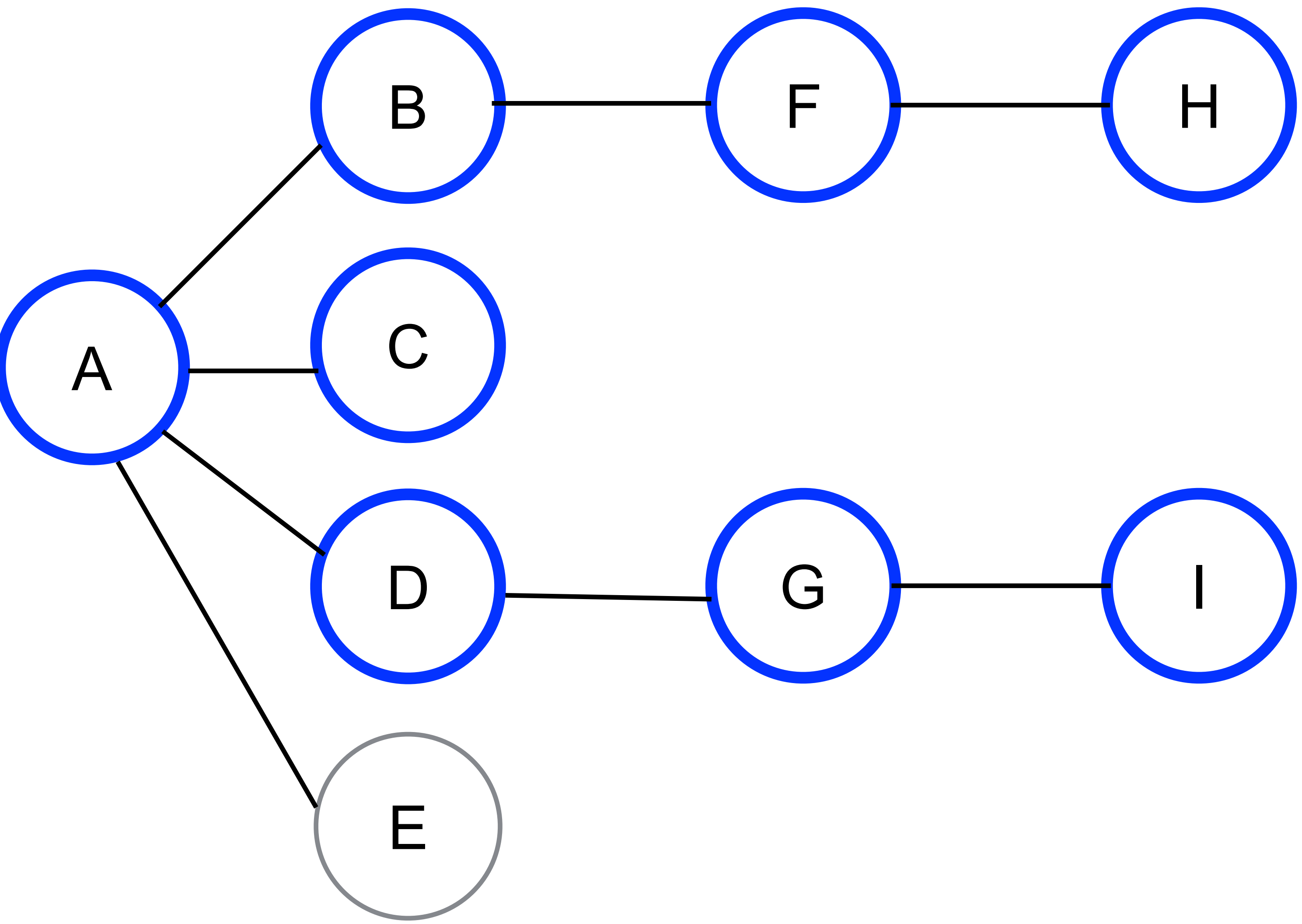

A
B
C
D
E
F
G
H
I


B
F
H
A
C
D
G
I
E

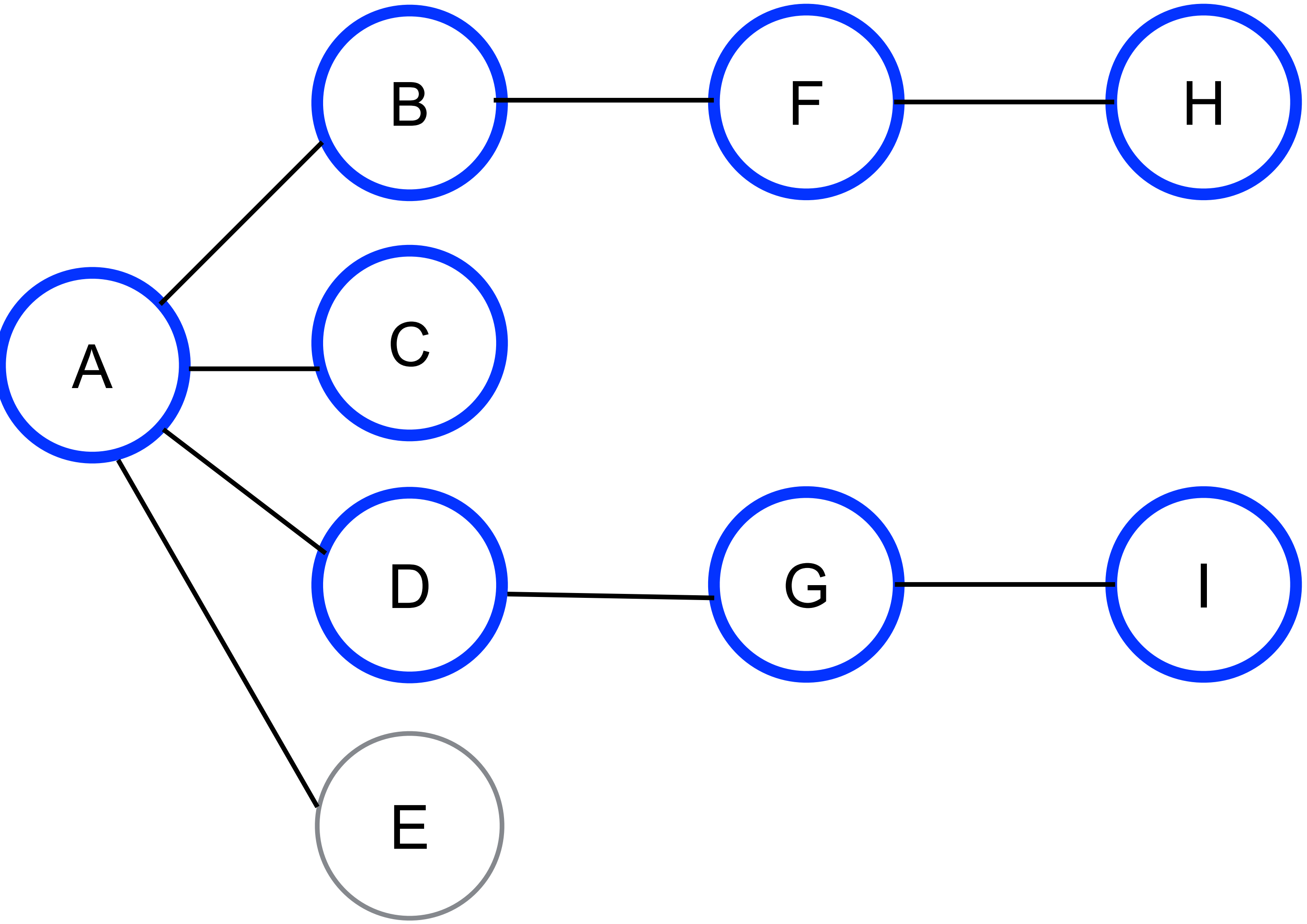

A
B
C
D
E
F
G
H
I



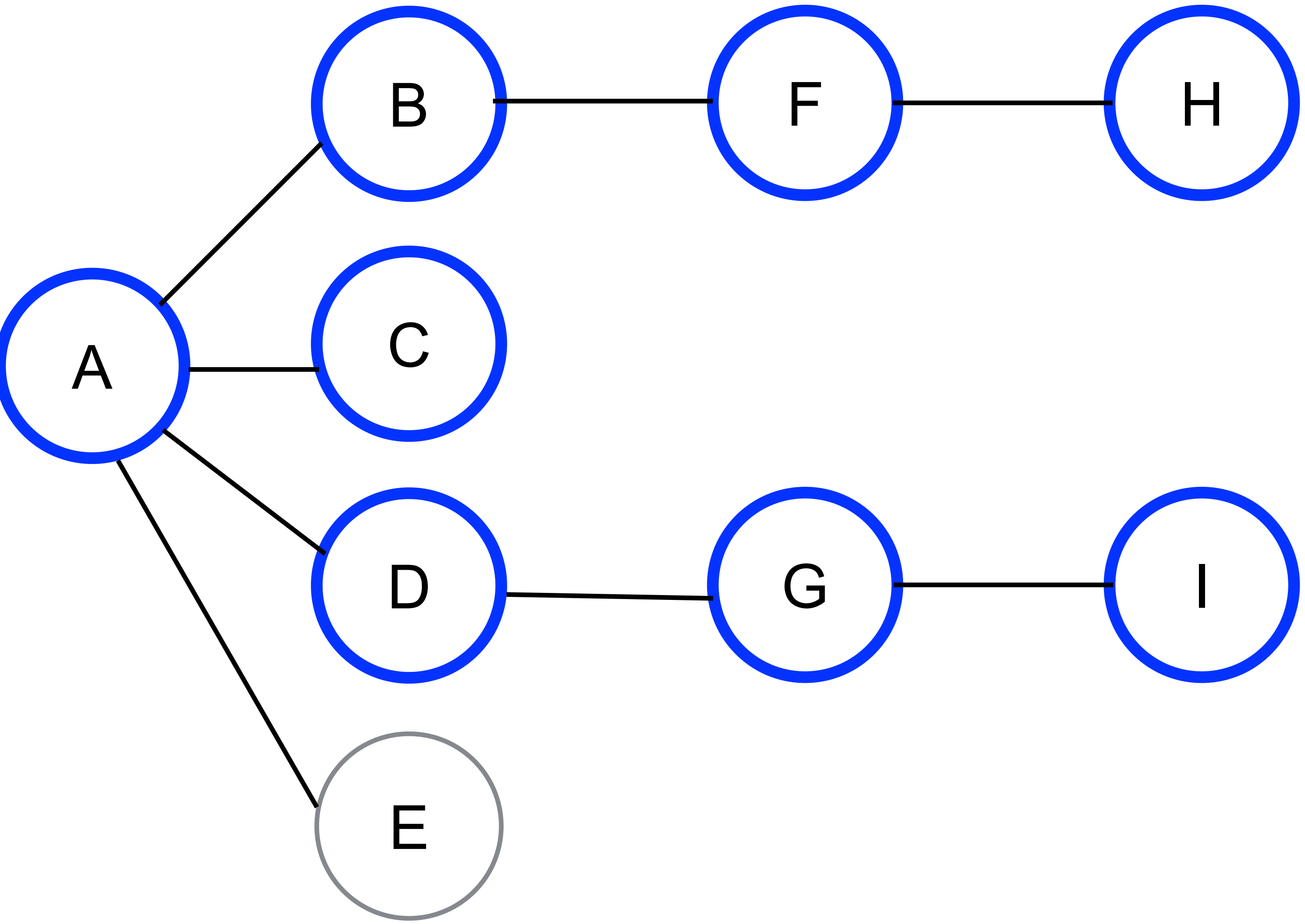
A
B
C
D
E
F
G
H
I

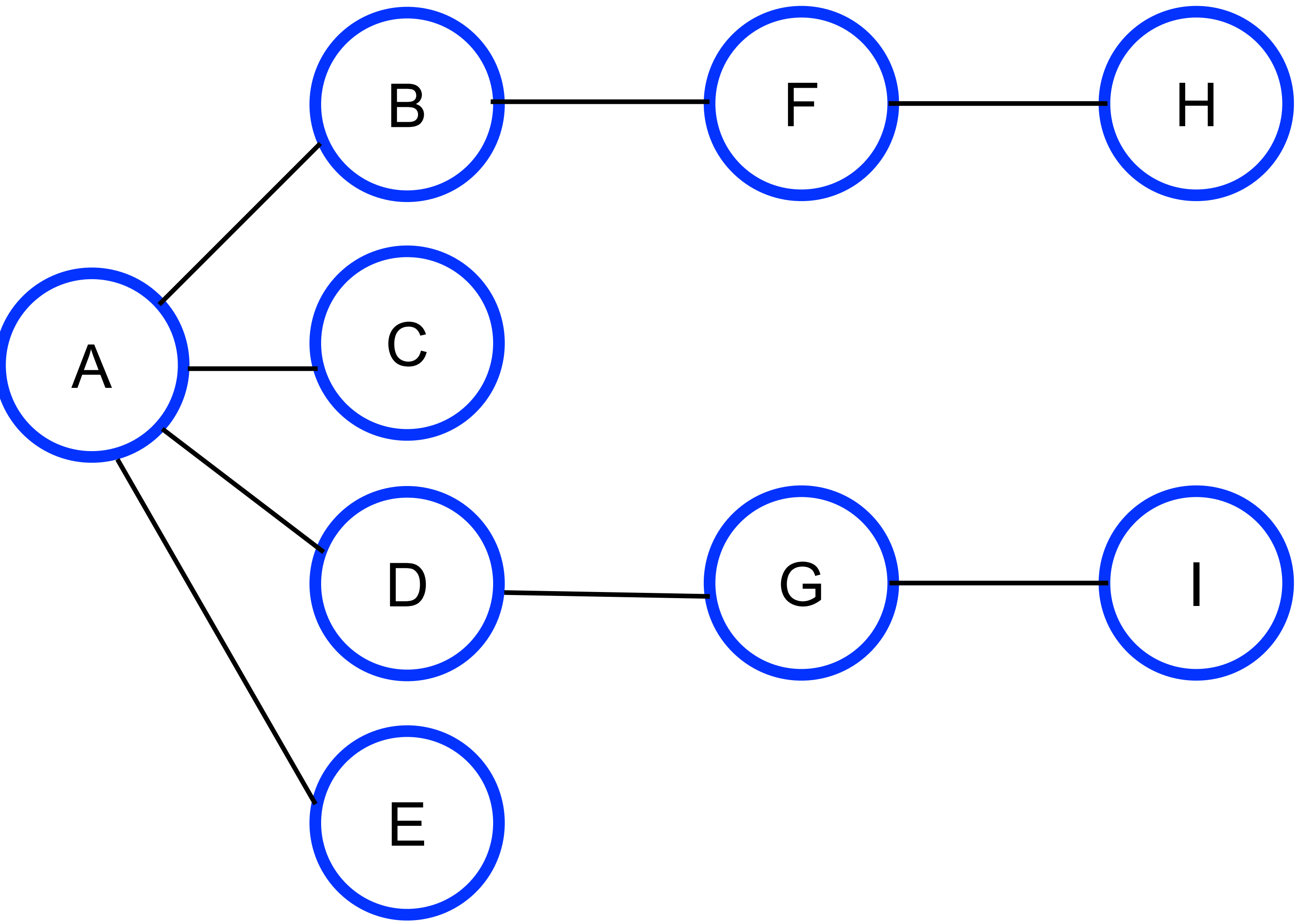


A
B
C
D
E
F
G
H
I

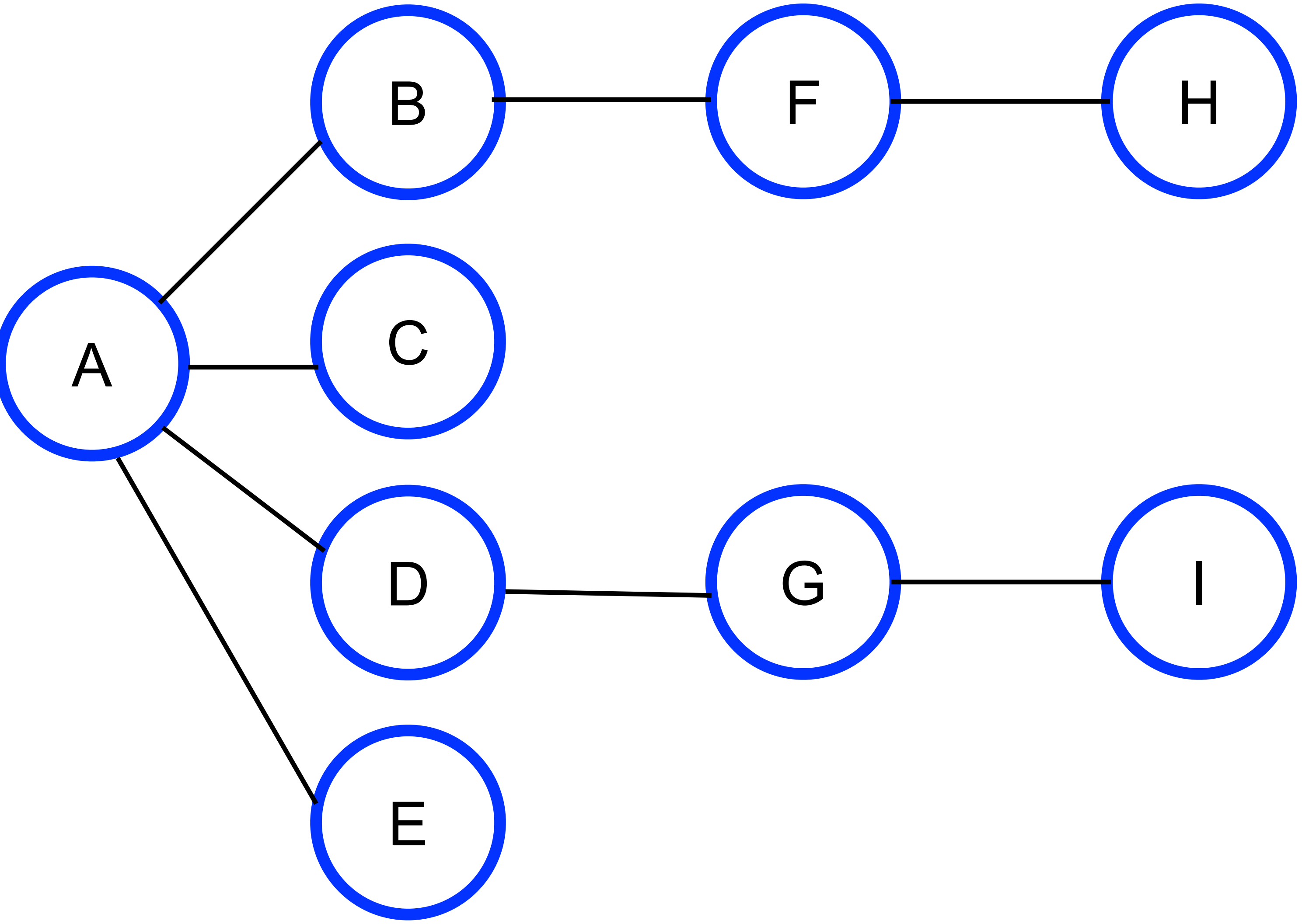


A
B
C
D
E
F
G
H
I

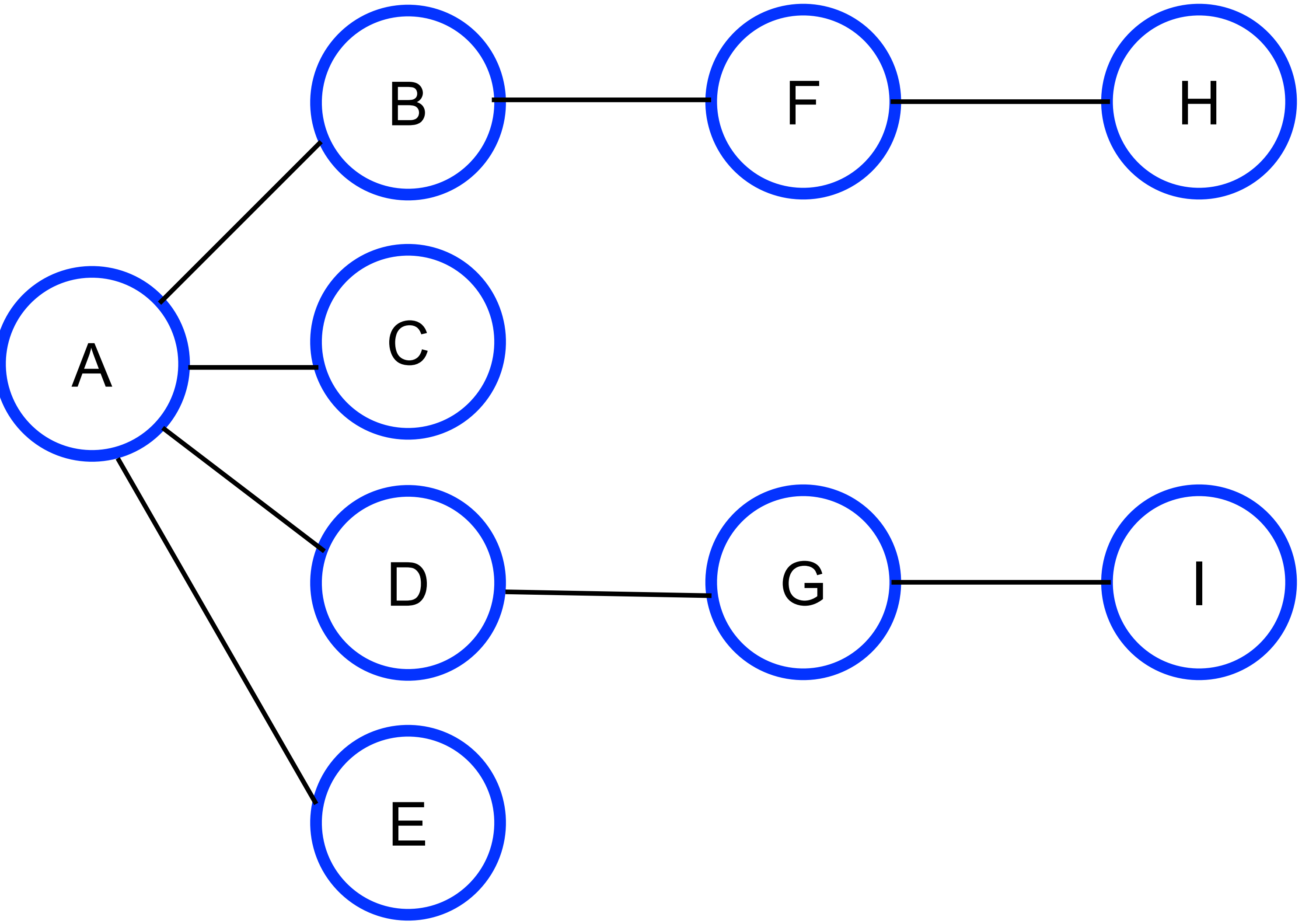

A
B
C
D
E
F
G
H
I



A
B
C
D
E
F
G
H
I

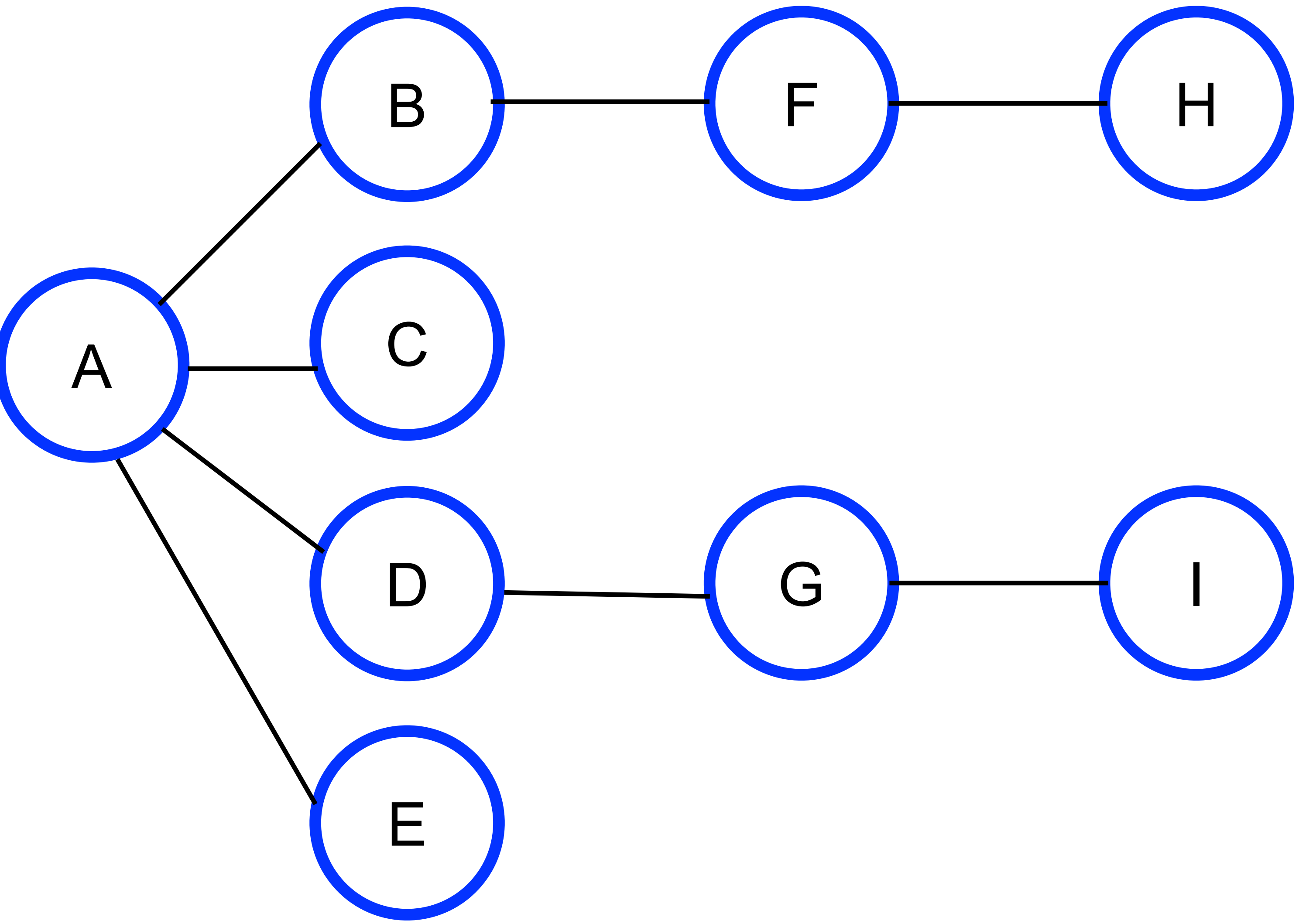


- At this point, A has no more adjacent unvisited vertices left
- We pop it off the stack



- This brings us to **Rule 3:**

“If you cannot follow Rule 1 or Rule 2, you are done”



Order: ABFHCDGIE

Time: O(|V| + |E|)

DFS

- Notice that,
  - DFS tries to get as far away from the starting point as quickly as possible
  - And returns only when it reaches a dead end
  - Thus the name, **Depth First Search**

DFS Implementation

```
// dfs.java
// demonstrates depth-first search
// to run this program: C>java DFSApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class StackX
   {
   private final int SIZE = 20;
   private int[] st;
   private int top;
   public StackX()            // constructor
      {
      st = new int[SIZE];    // make array
      top = -1;
      }
   public void push(int j)   // put item on stack
      { st[++top] = j; }
```

DFS Implementation (2)

```
   public int pop()           // take item off stack
      { return st[top--]; }
   public int peek()          // peek at top of stack
      { return st[top]; }
   public boolean isEmpty()   // true if nothing on stack
      { return (top == -1); }
   }  // end class StackX

////////////////////////////////////////////////////////////////

class Vertex
   {
   public char label;         // label (e.g. 'A')
   public boolean wasVisited;
```

DFS Implementation (3)

```
// ------------------
   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      }

// ------------------
   }  // end class Vertex

////////////////////////////////////////////////////////////////

class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private StackX theStack;
```

DFS Implementation (4)

```
// ------------------
   public Graph()                    // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      theStack = new StackX();
      }  // end constructor

// ------------------
```

DFS Implementation (5)

```
    public void addVertex(char lab)
       {
       vertexList[nVerts++] = new Vertex(lab);

        }

 // -----------------
     public void addEdge(int start, int end)
        {
        adjMat[start][end] = 1;
        adjMat[end][start] = 1;
        }

 // -----------------
     public void displayVertex(int v)
        {
        System.out.print(vertexList[v].label);
        }
```

DFS Implementation (6)

```
// ------------------
   public void dfs()  // depth-first search
      {                                  // begin at vertex 0
      vertexList[0].wasVisited = true;   // mark it
      displayVertex(0);                  // display it
      theStack.push(0);                  // push it

      while( !theStack.isEmpty() )       // until stack empty,
         {
         // get an unvisited vertex adjacent to stack top
         int v = getAdjUnvisitedVertex( theStack.peek() );
         if(v == -1)                     // if no such vertex,
            theStack.pop();
```

DFS Implementation (7)

```
    else                                // if it exists,
       {
       vertexList[v].wasVisited = true;  // mark it
       displayVertex(v);                 // display it
       theStack.push(v);                 // push it
       }
    }  // end while

    // stack is empty, so we're done
    for(int j=0; j<nVerts; j++)          // reset flags
       vertexList[j].wasVisited = false;
 }  // end dfs
```

DFS Implementation (8)

```
    // ------------------
       // returns an unvisited vertex adj to v
       public int getAdjUnvisitedVertex(int v)
          {
          for(int j=0; j<nVerts; j++)
             if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
                return j;
          return -1;
          }  // end getAdjUnvisitedVert()

    // ------------------


  }  // end class Graph

/////////////////////////////////////////////////////////////////
```

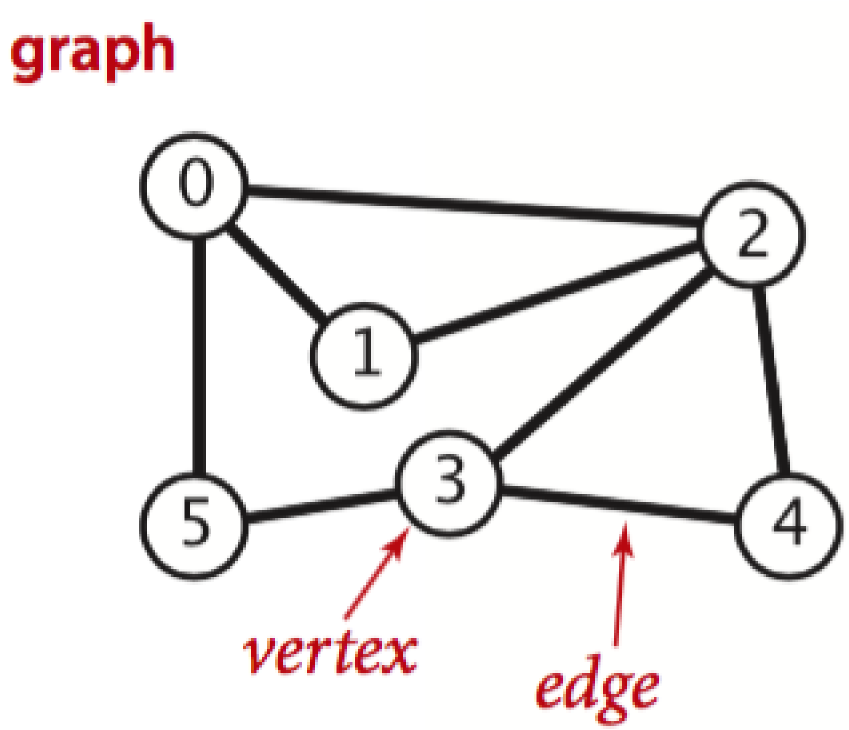
DFS Implementation (9)

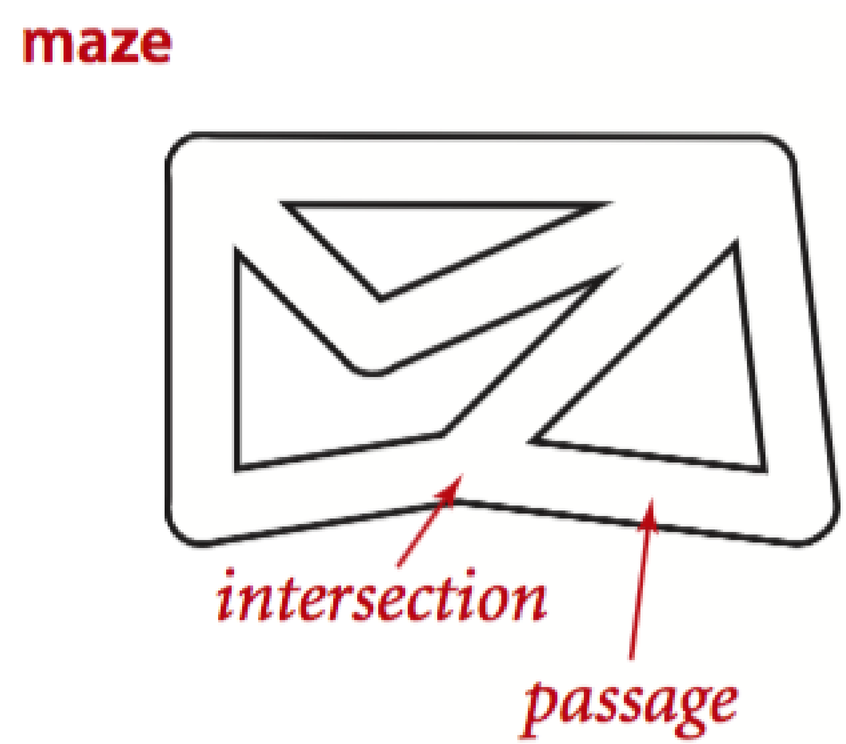
```
class DFSApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0  (start for dfs)
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4

      theGraph.addEdge(0, 1);     // AB
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(3, 4);     // DE

      System.out.print("Visits: ");
      theGraph.dfs();             // depth-first search
      System.out.println();
      }  // end main()
   }  // end class DFSApp
```

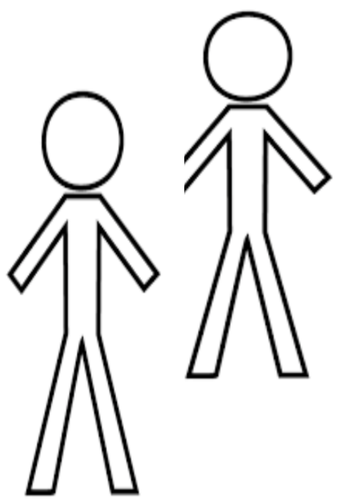
Breadth First Search



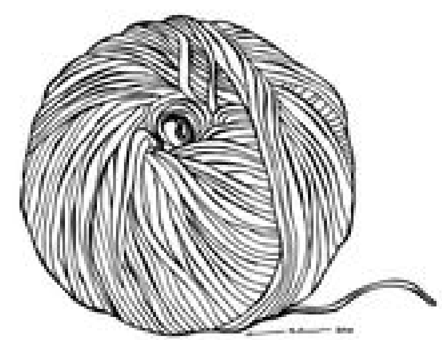


Breadth First Search

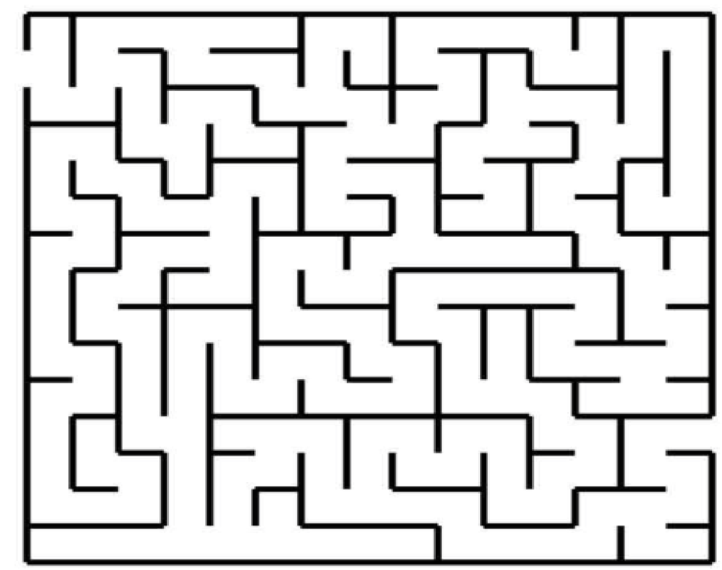
- Searching in a maze



Group of Searchers

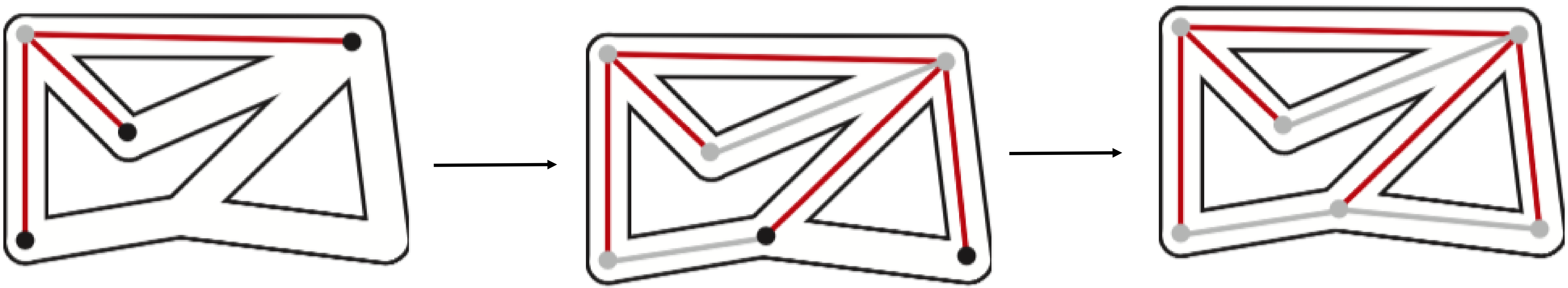


String



Maze

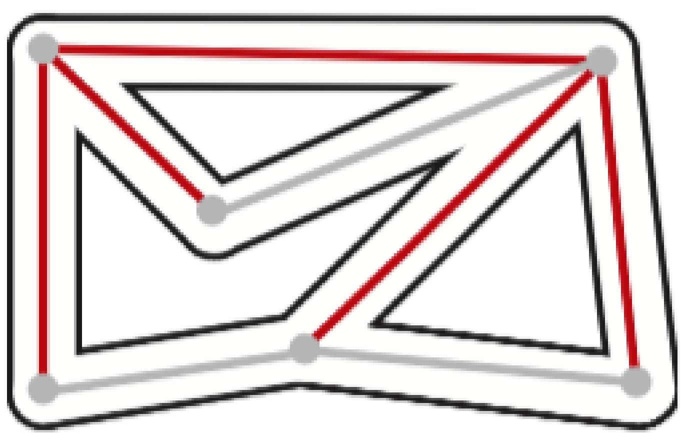
Breadth First Search

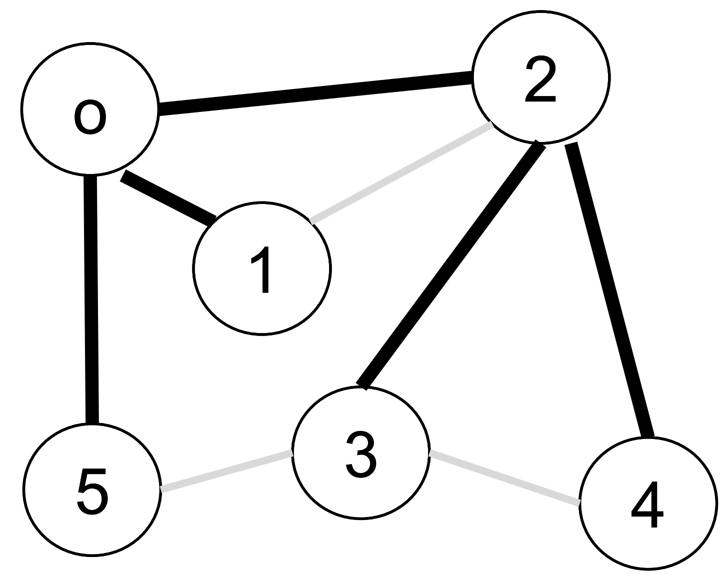


a b c

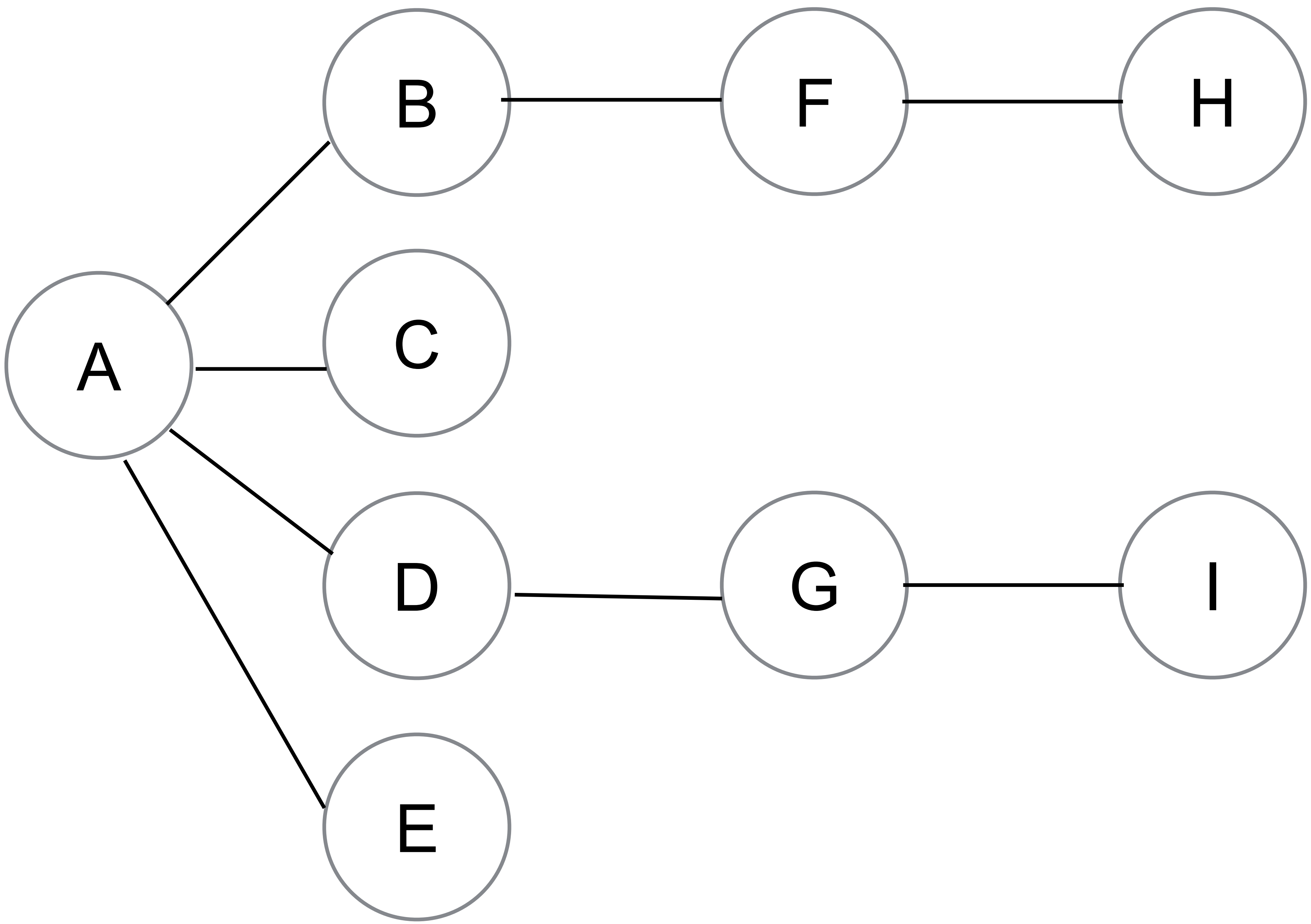
BFS Maze Exploration

Maze vs. Graph



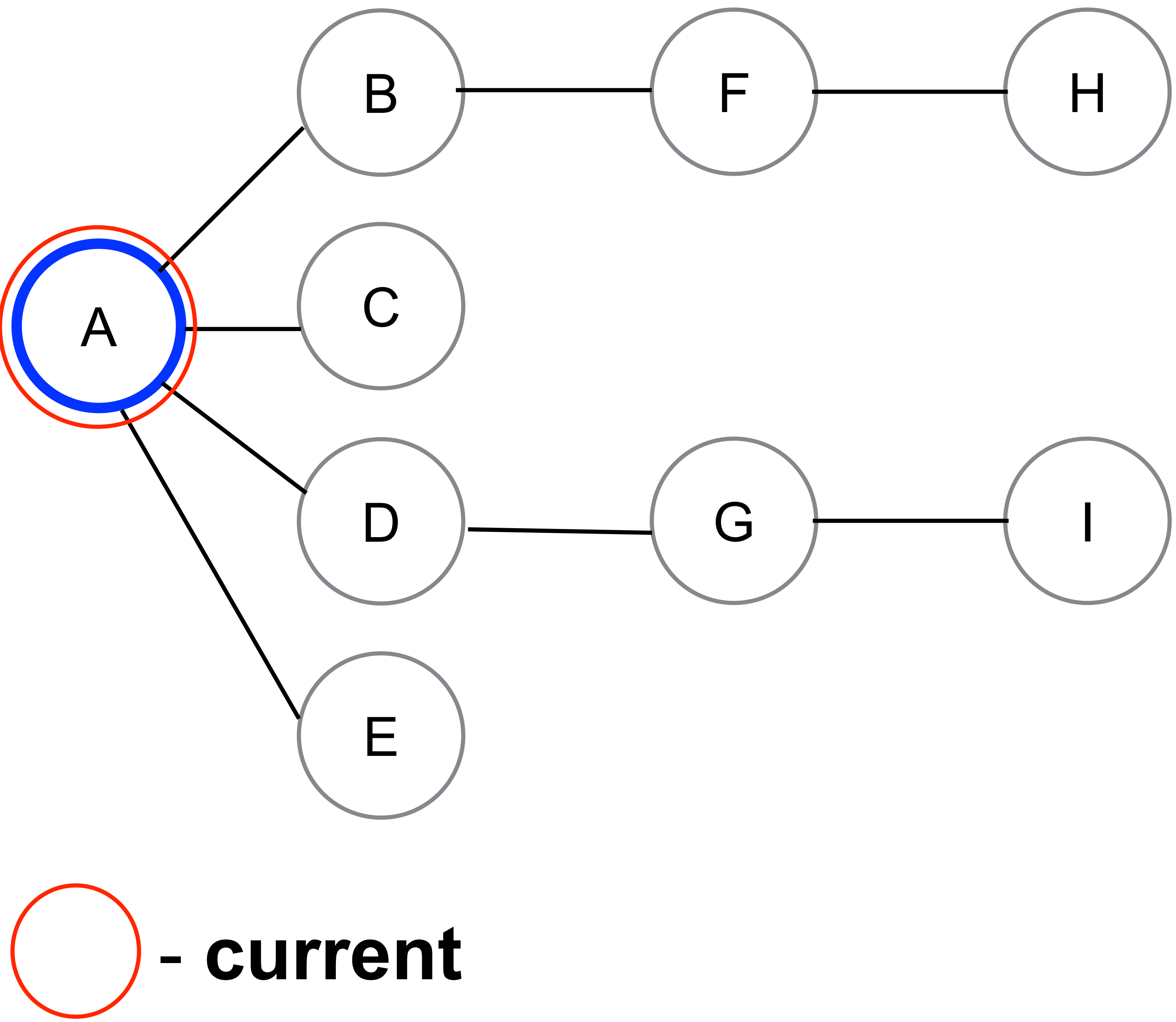


BFS with a Queue

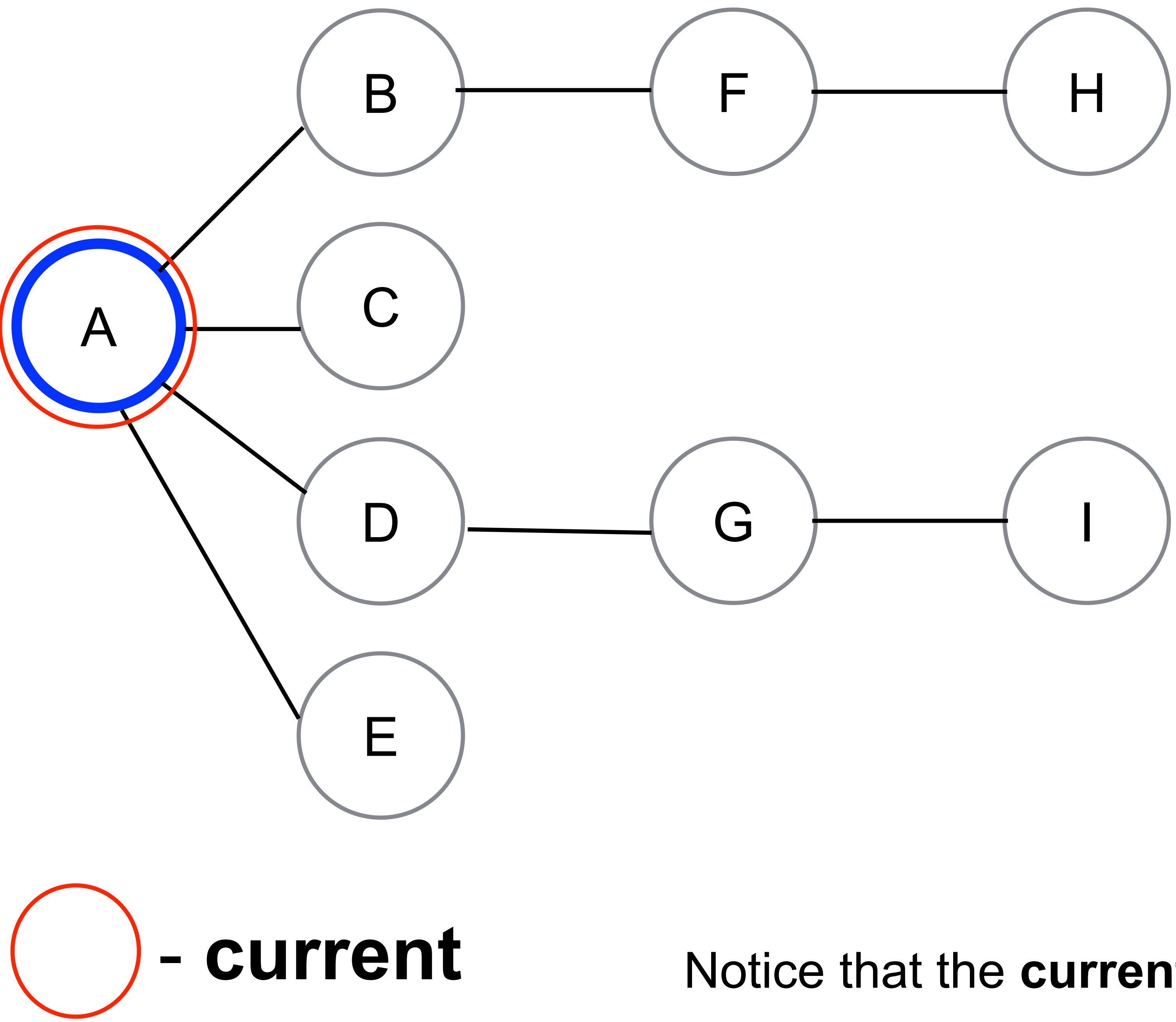


BFS with a Queue (2)

- Start with a vertex, visit it, and call it **current**
- Let's start with vertex A



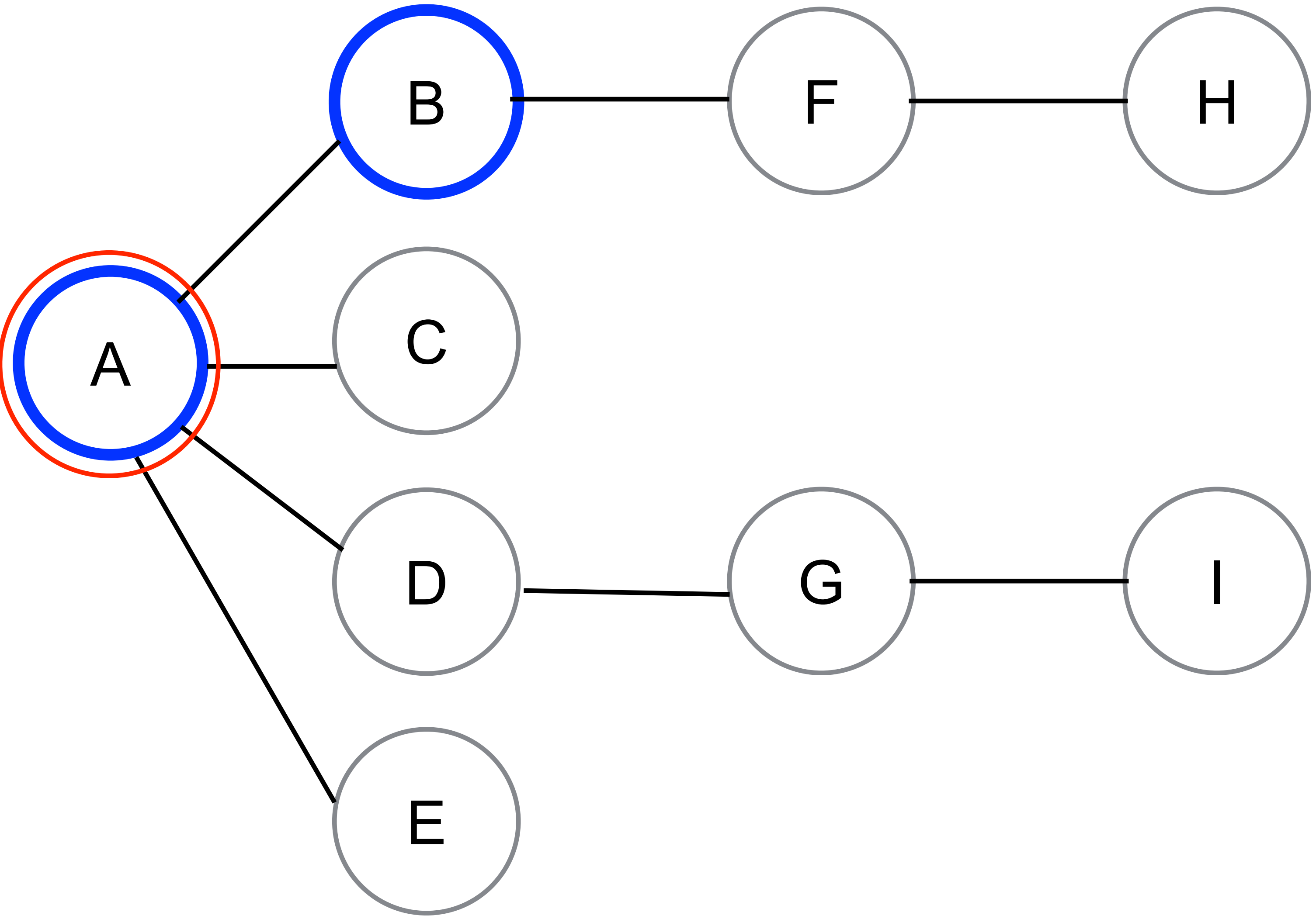
A
B
C
D
E
F
G
H
I
- current



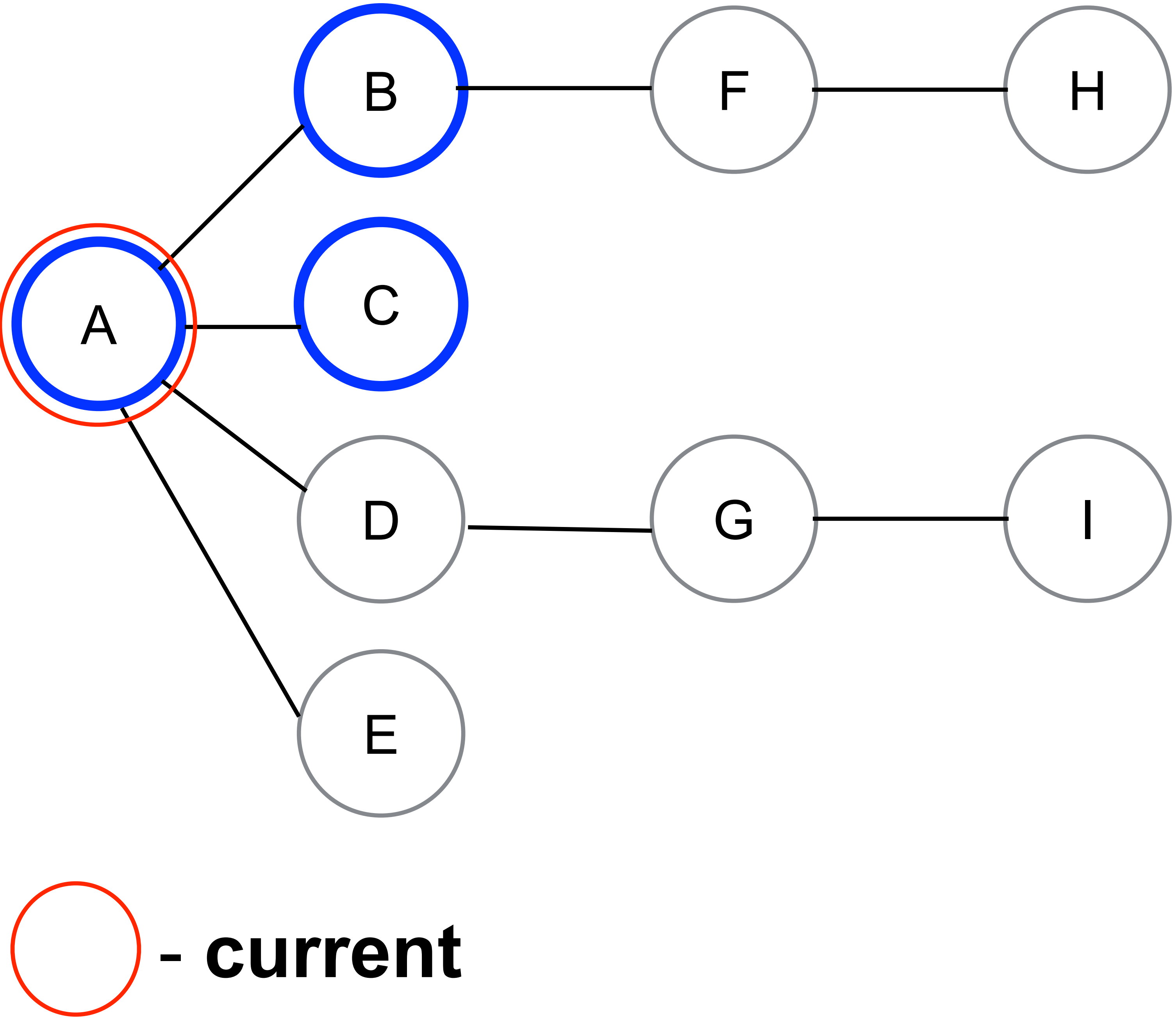
Notice that the **current** is not inserted into the queue

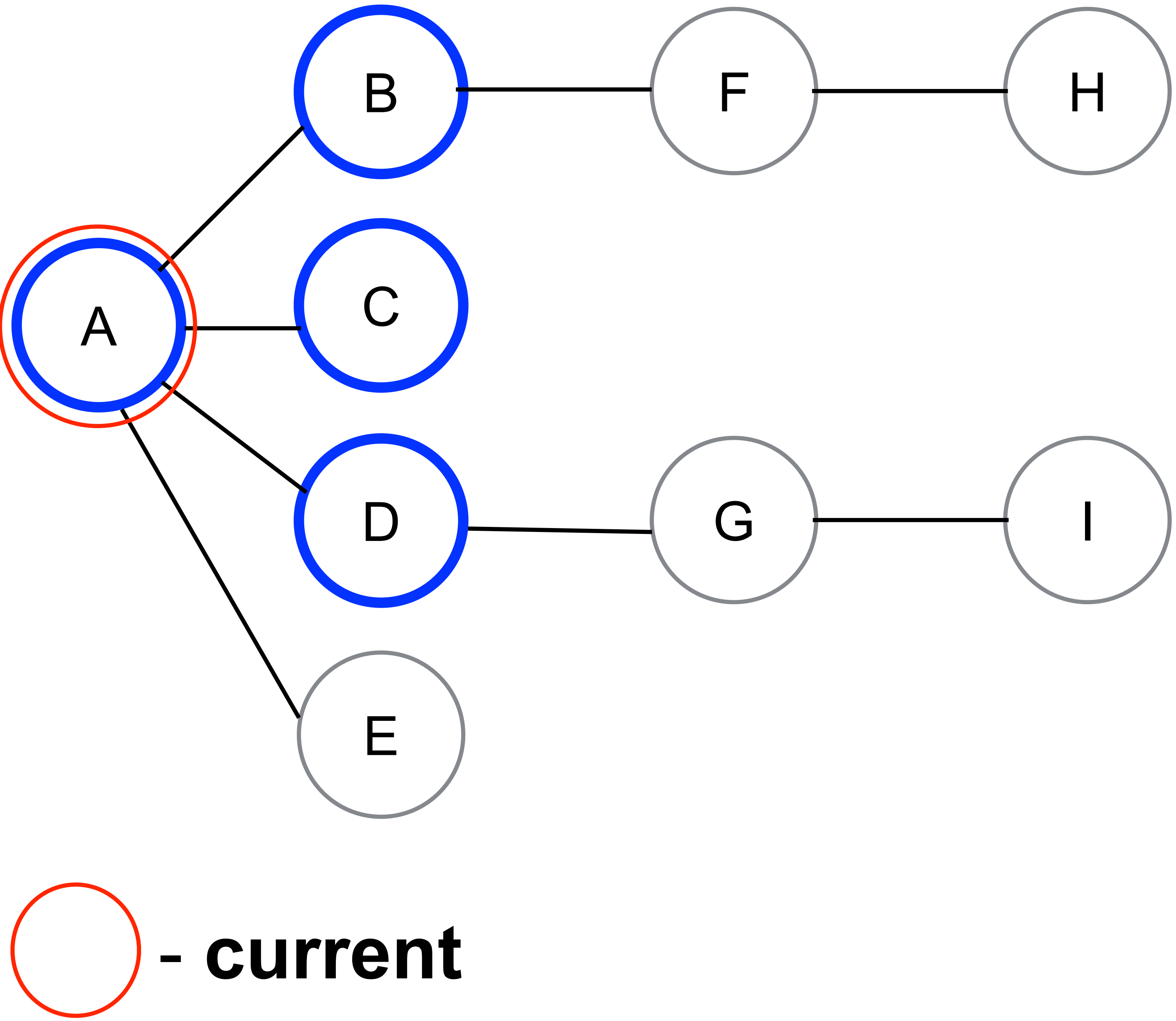
Now follow this rule

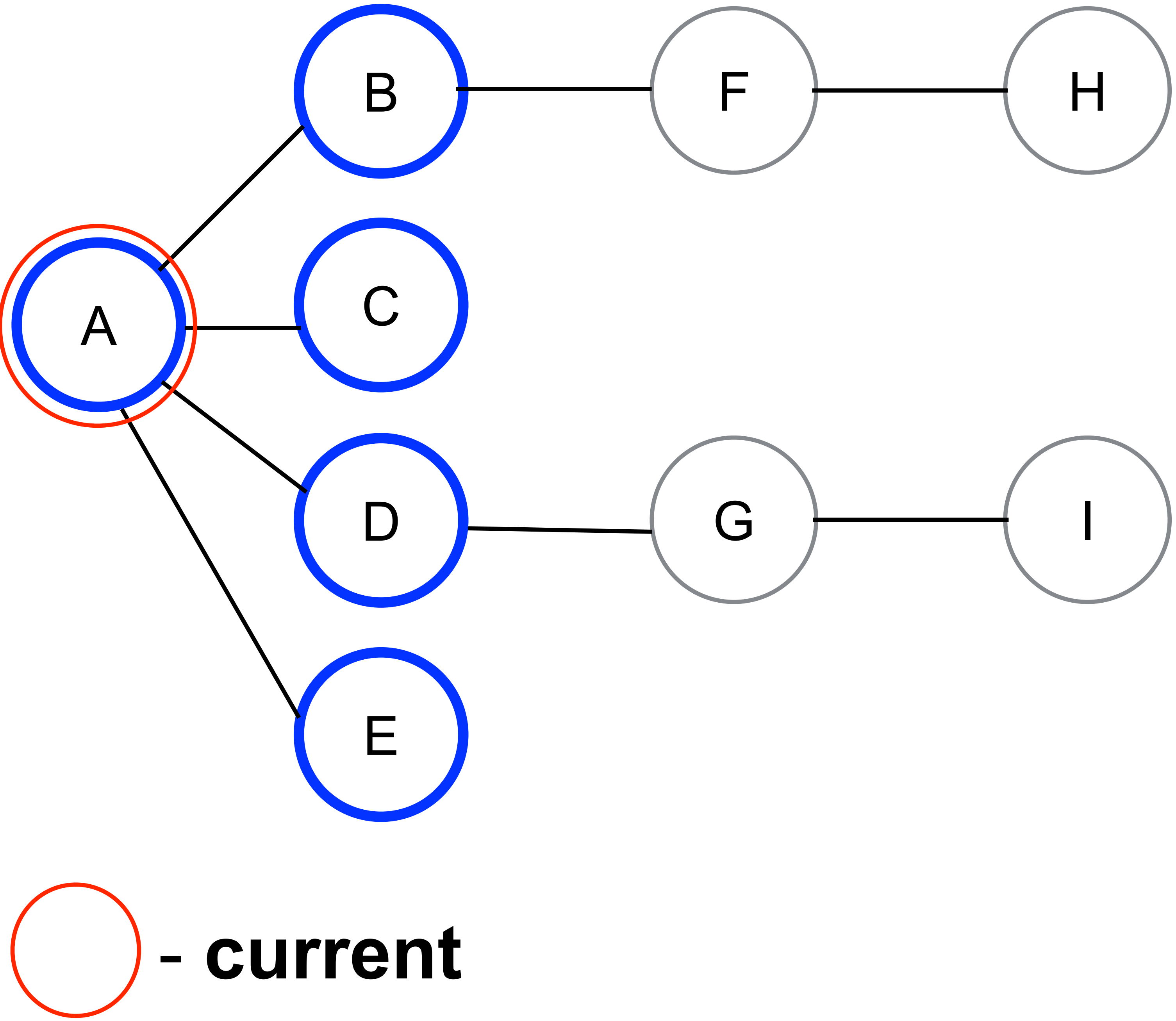
Rule 1: Visit the next unvisited vertex (if there is one) that is adjacent to the **current** vertex, mark it, and insert it into the queue

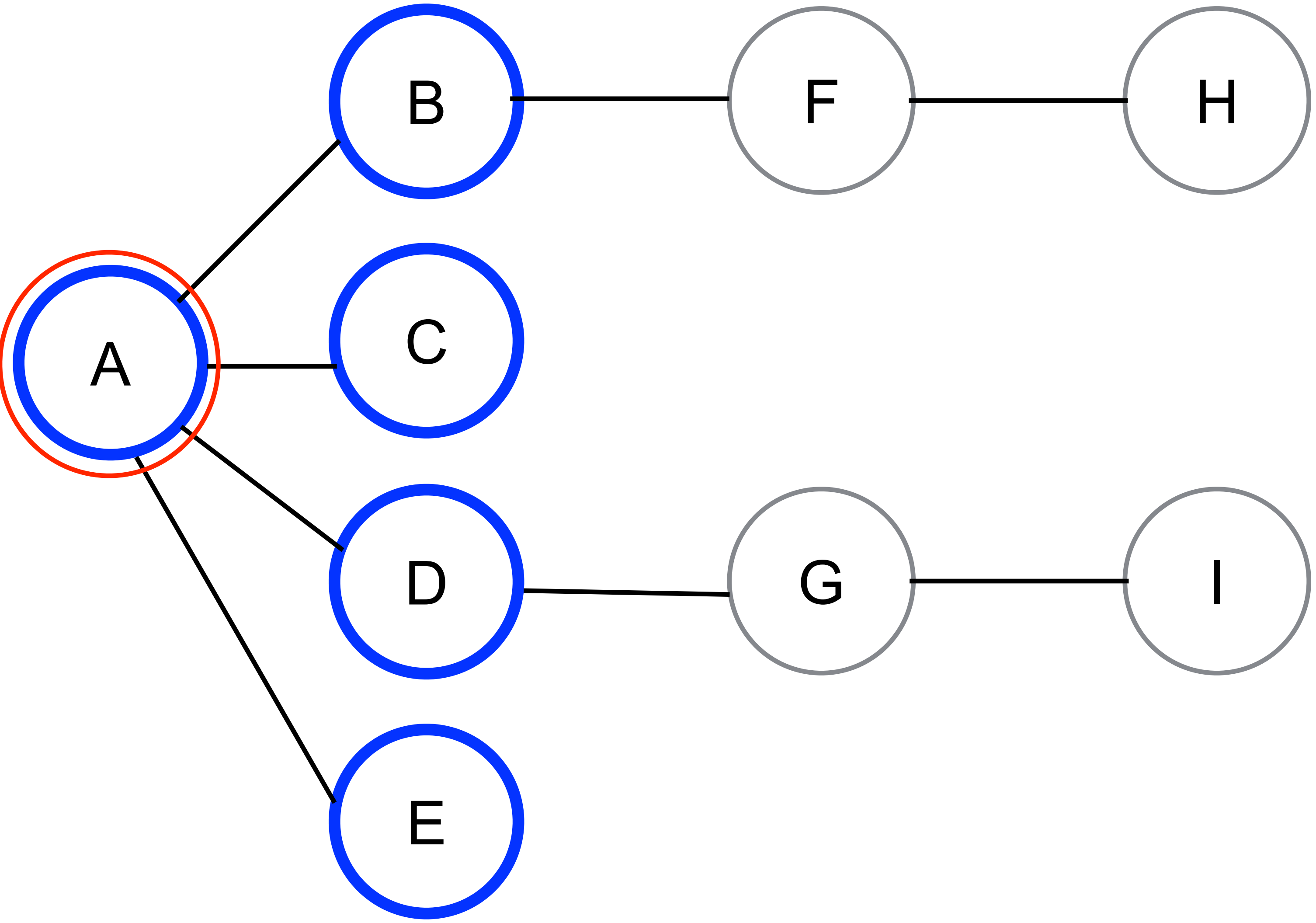


- current


A
B
C
D
E
F
G
H
I
- current

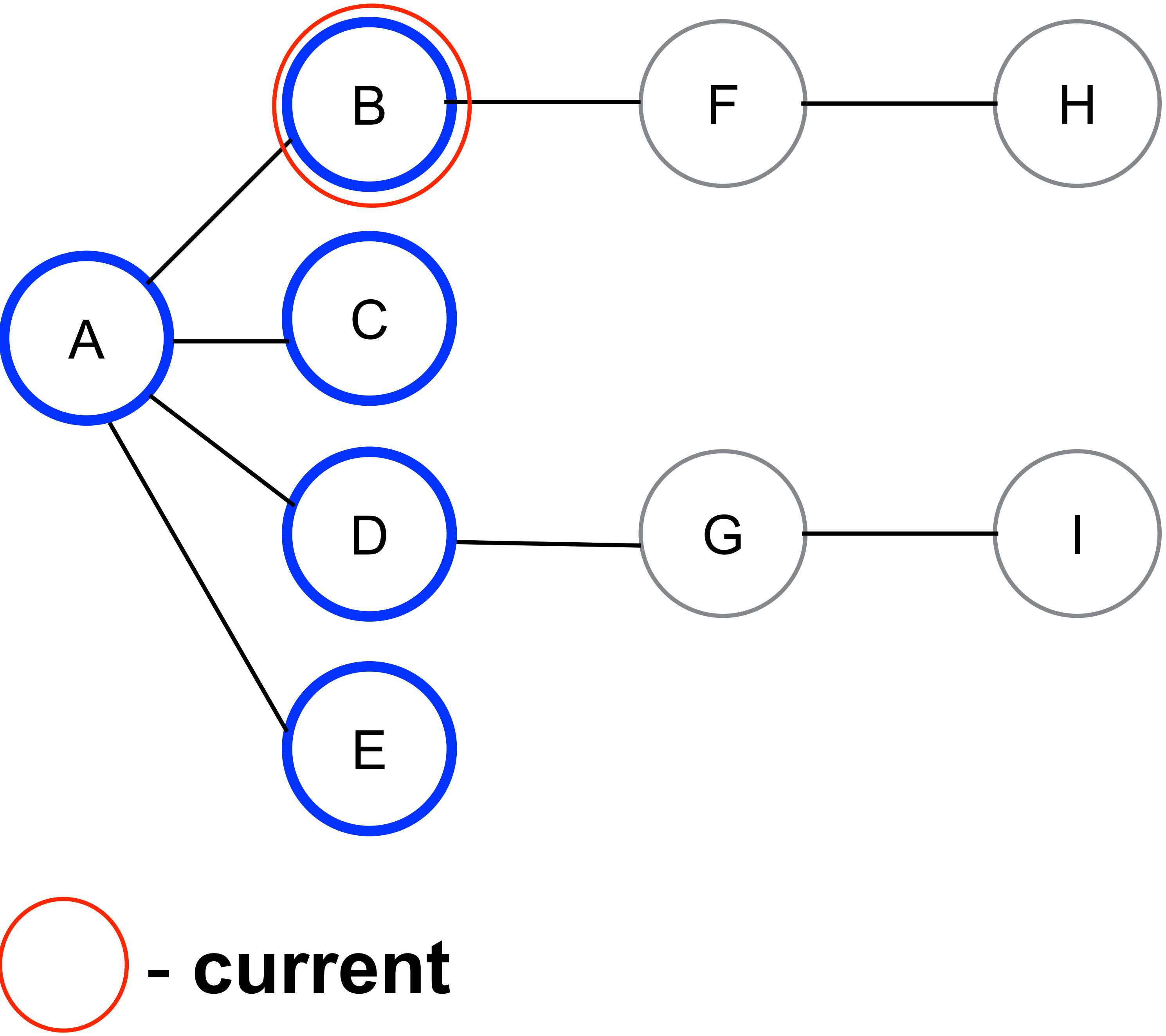

A
B
C
D
E
F
G
H
I
- current

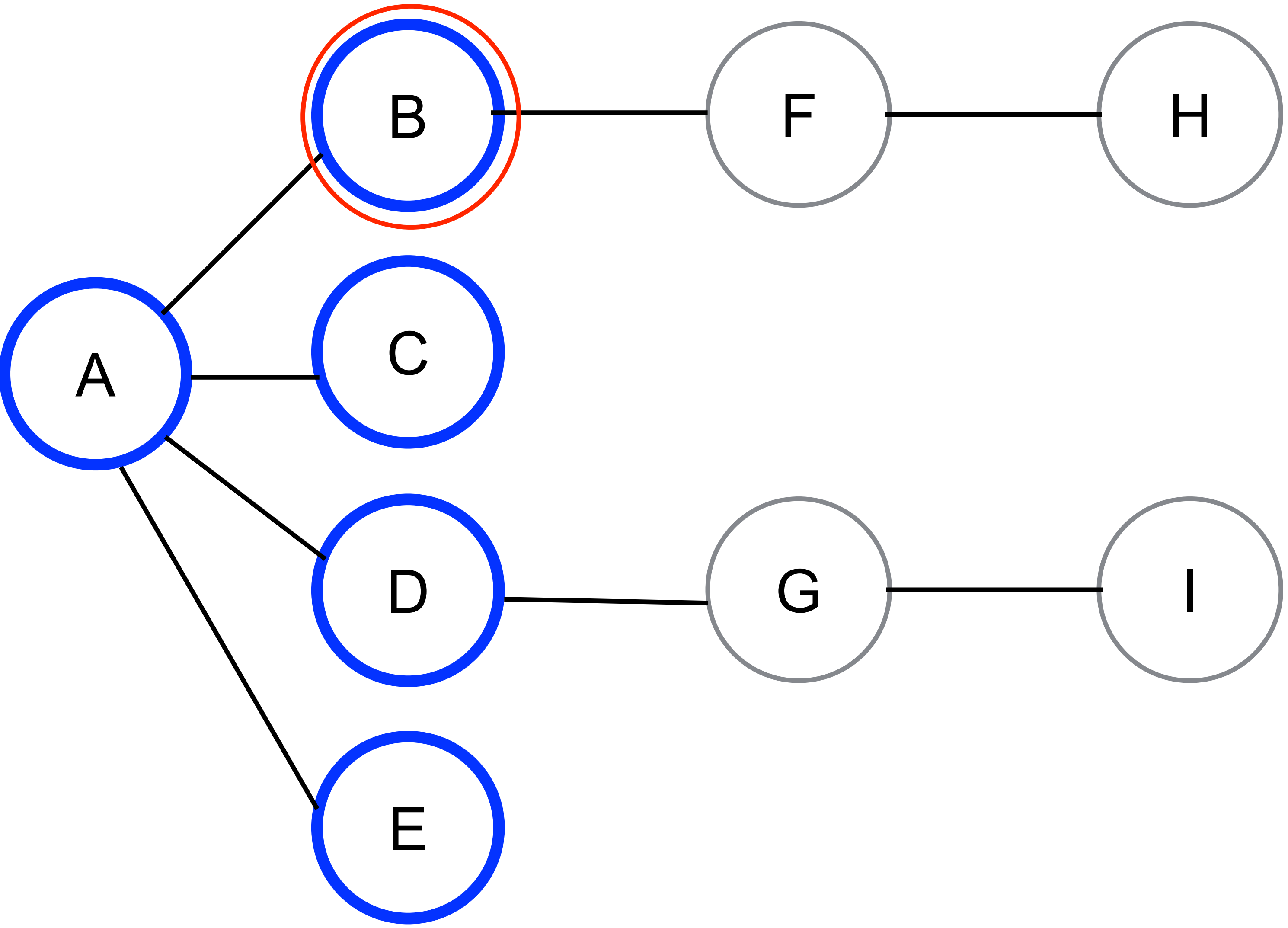

A
B
C
D
E
F
G
H
I
- current



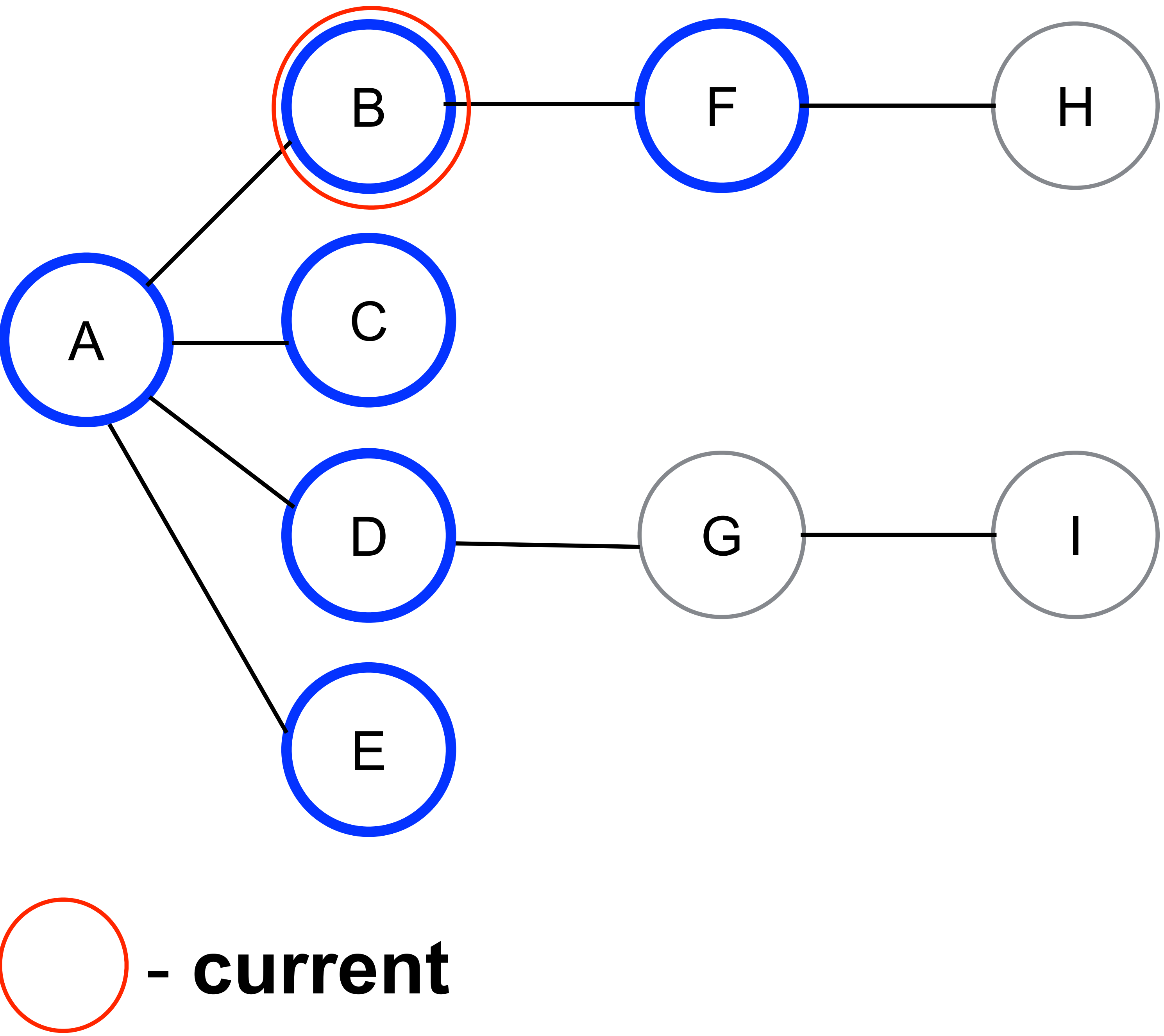
At this point A (**the current**) has no more unvisited adjacent vertex
So, follow **Rule 2**:

If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it **current** vertex

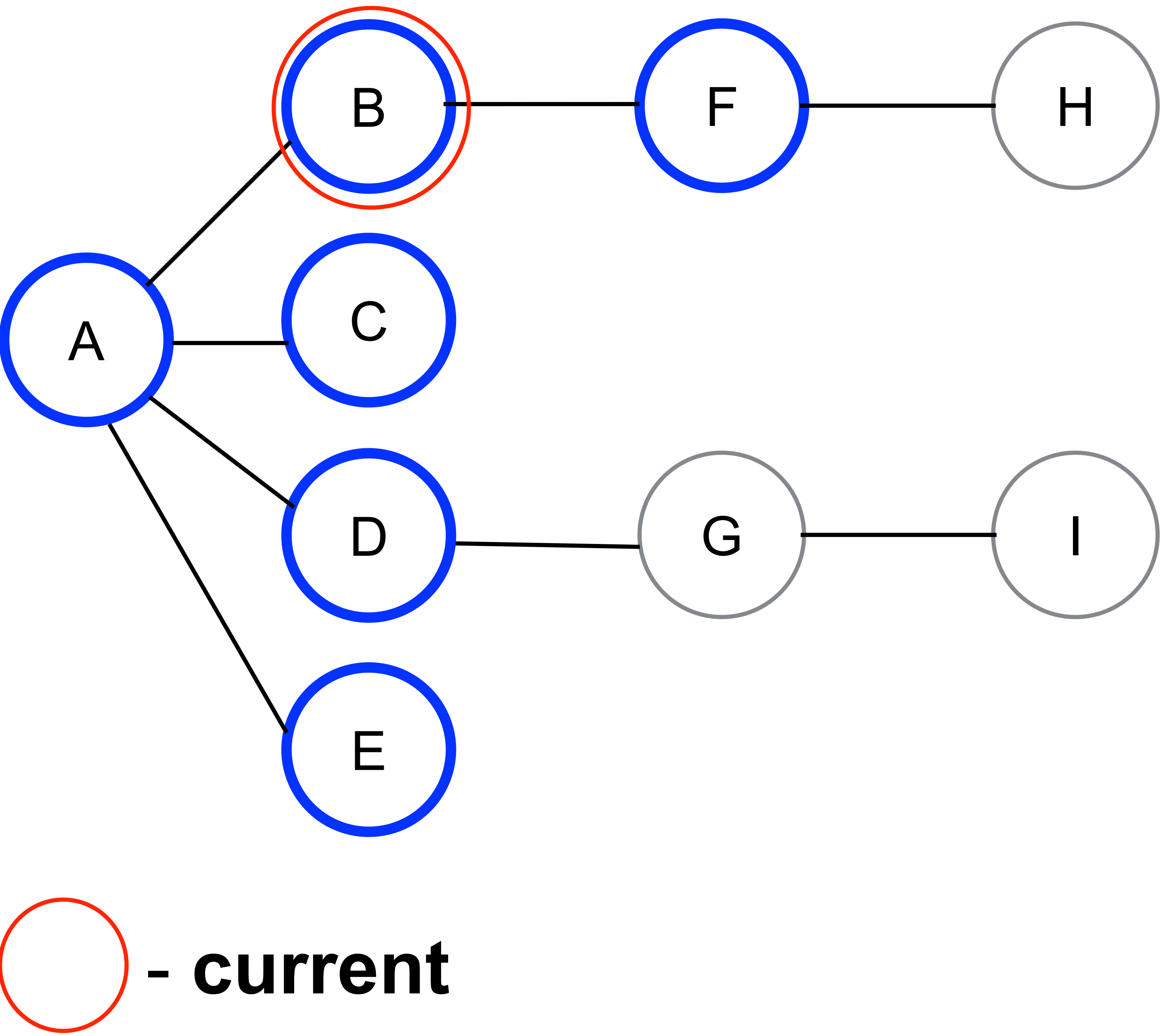

A
B
C
D
E
F
G
H
I
- current



Repeat Rule 1 for the new **current**

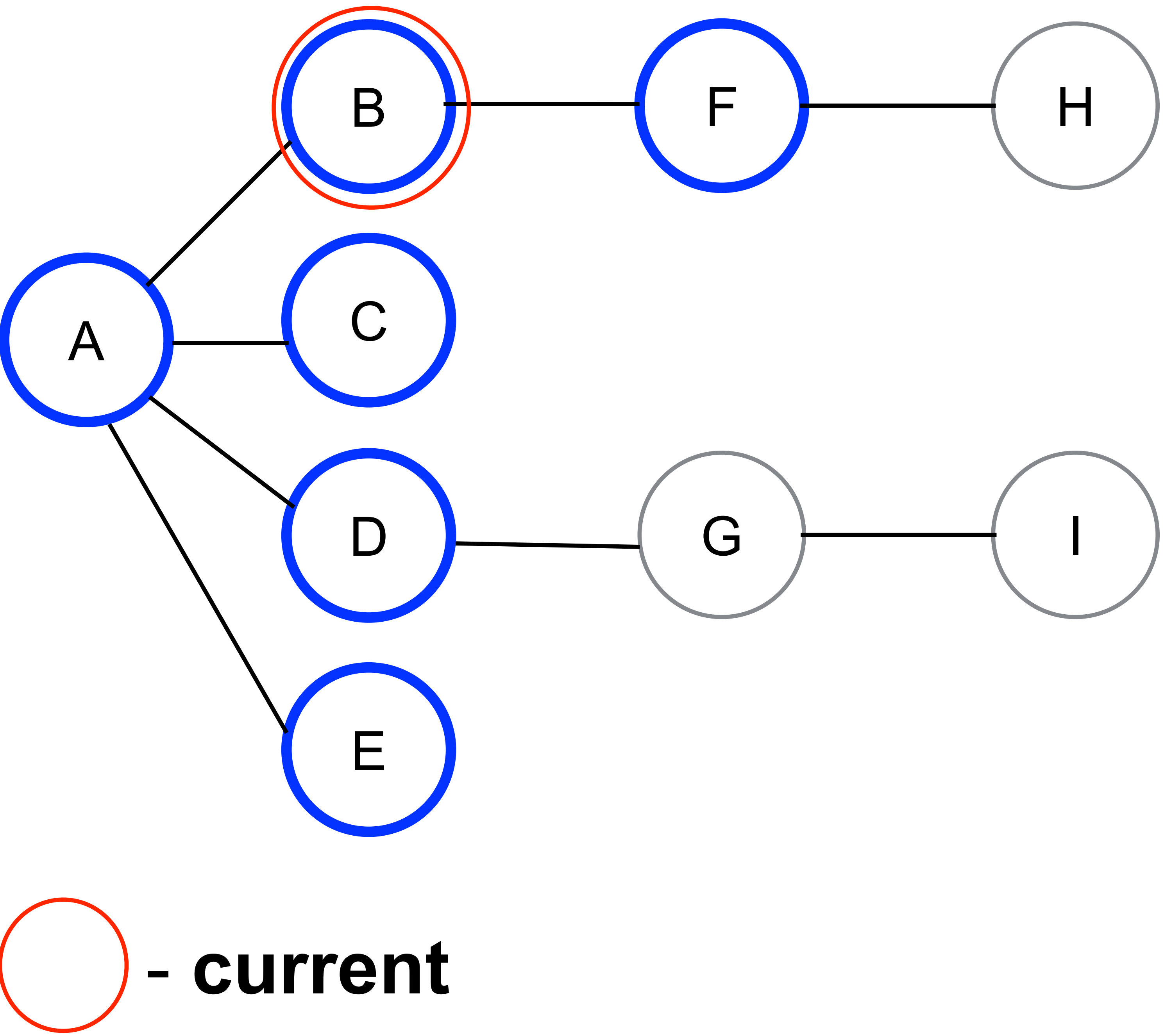


Will we follow BA?



Will we follow BA?

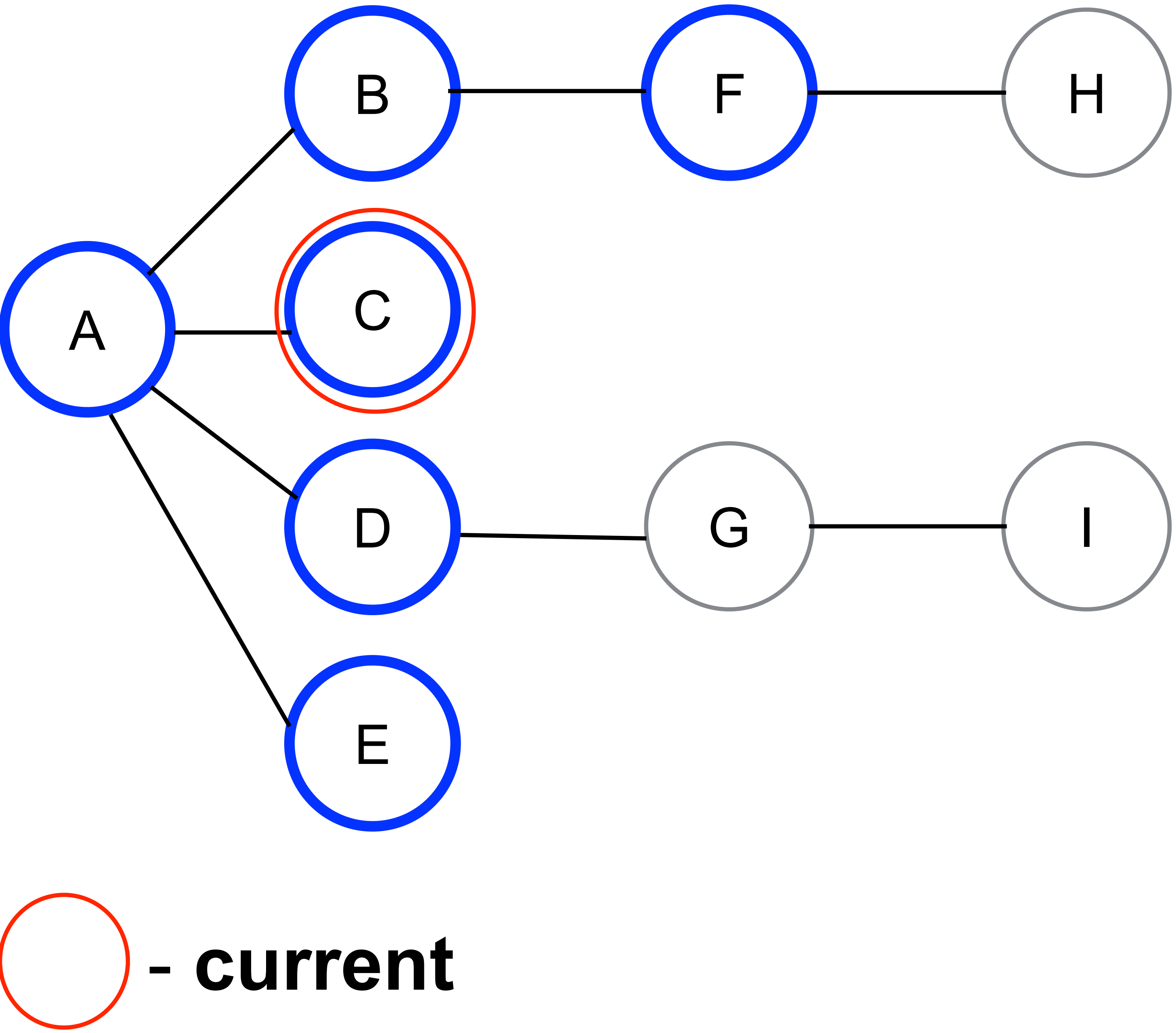
Yes! But it will take us back to A, which is already visited!

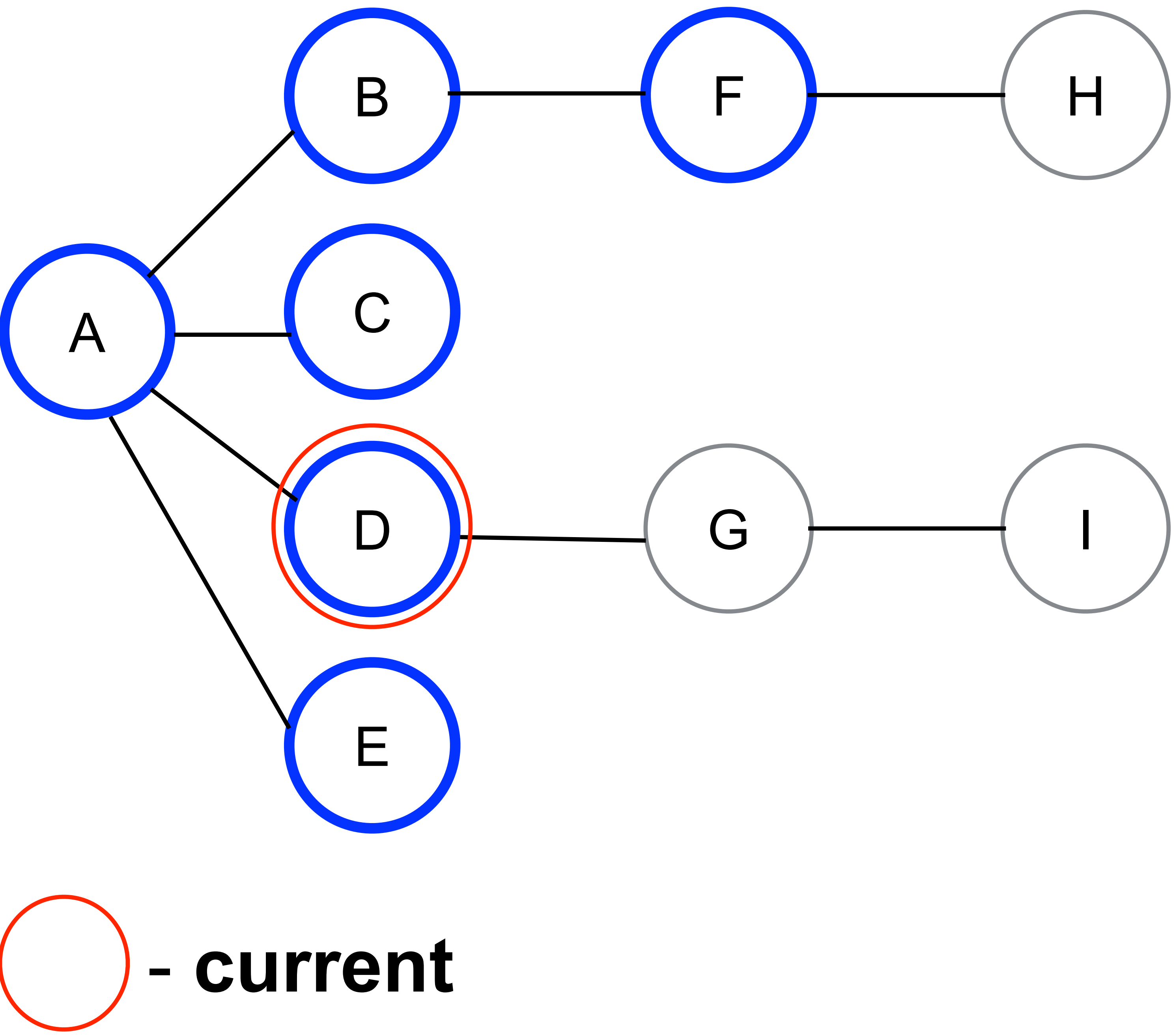


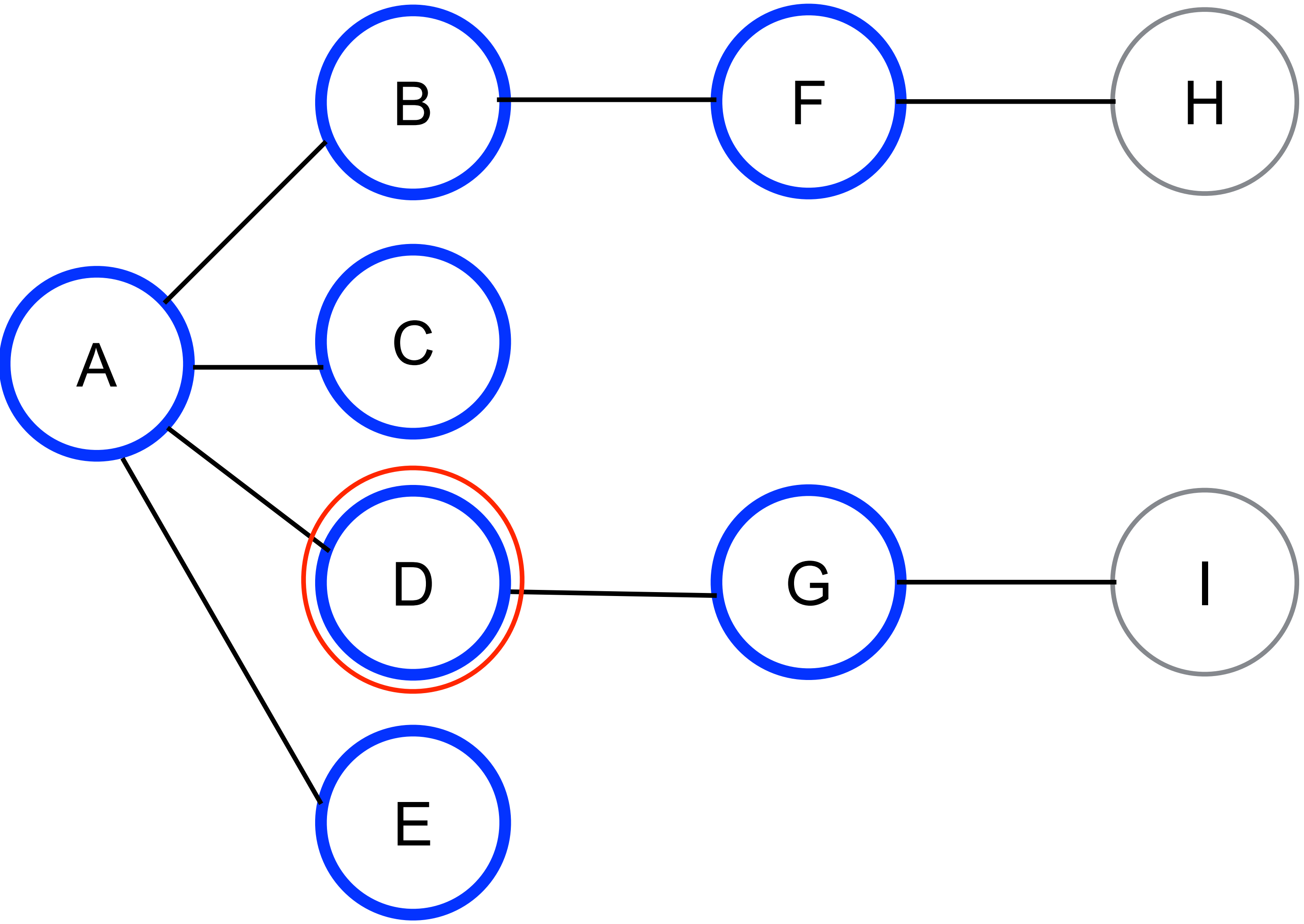
Will we follow BA?

Yes! But it will take us back to A, which is already visited!

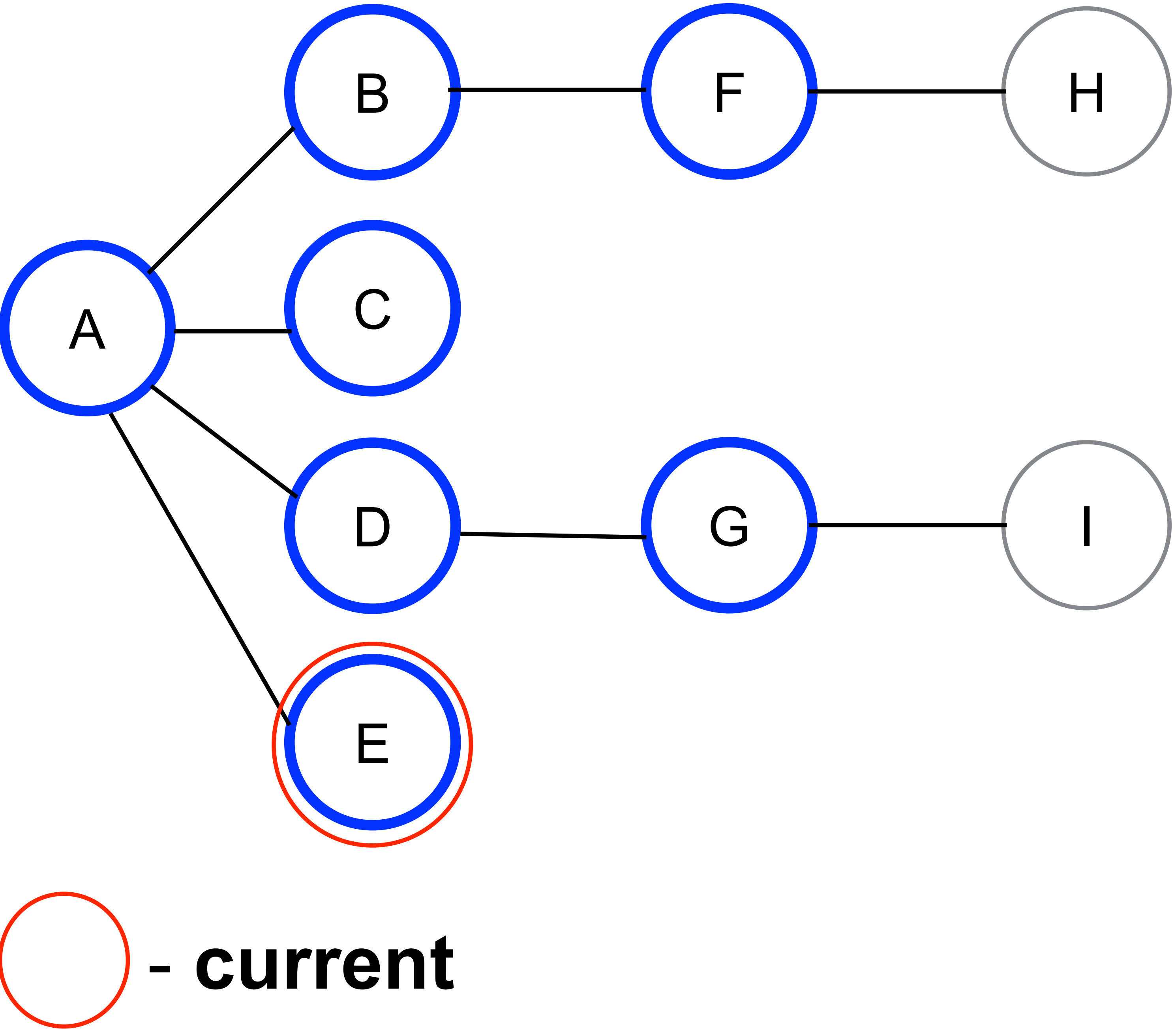
Thus each, vertex is visited once, and each edge twice!

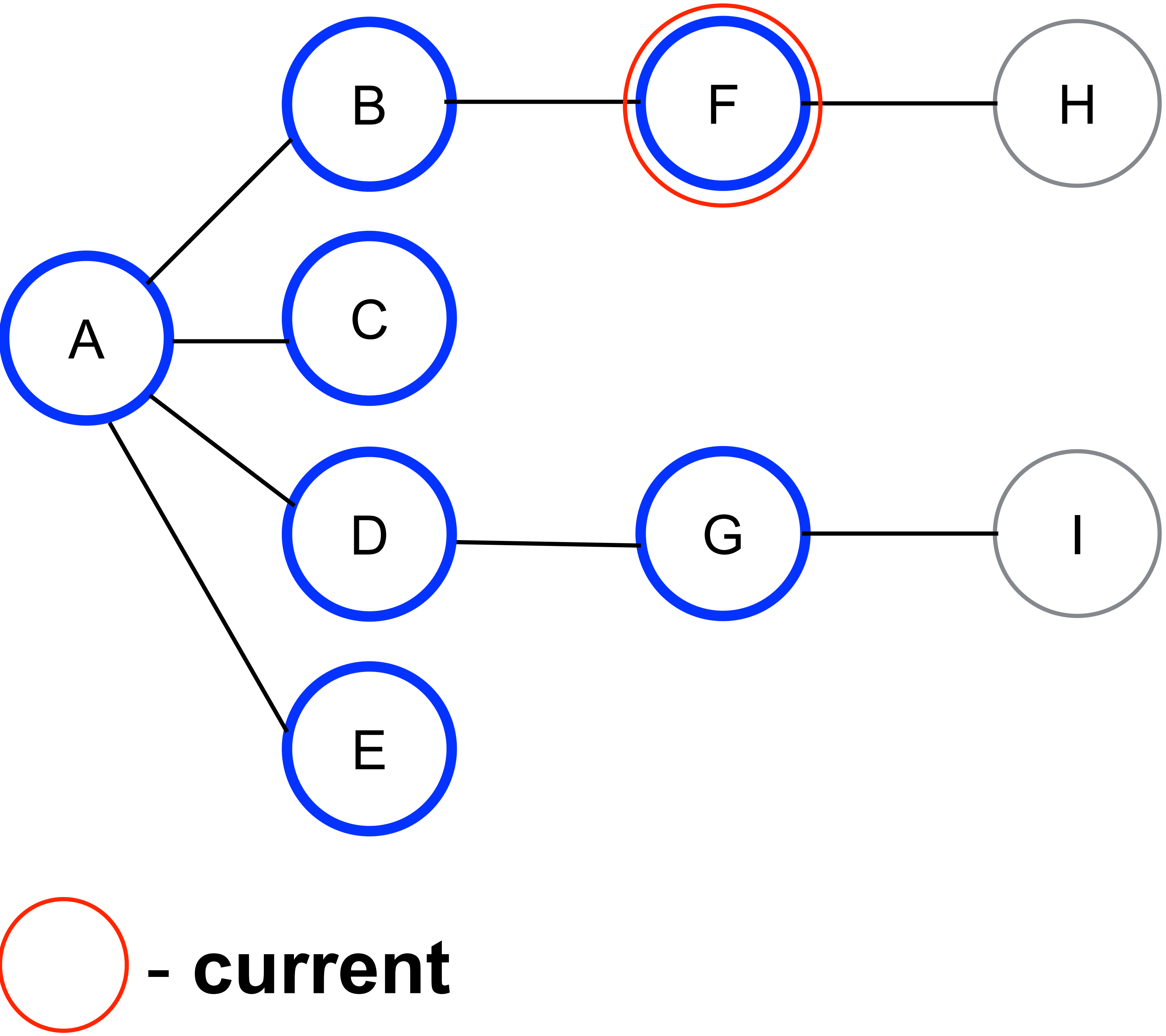

A
B
C
D
E
F
G
H
I
- current

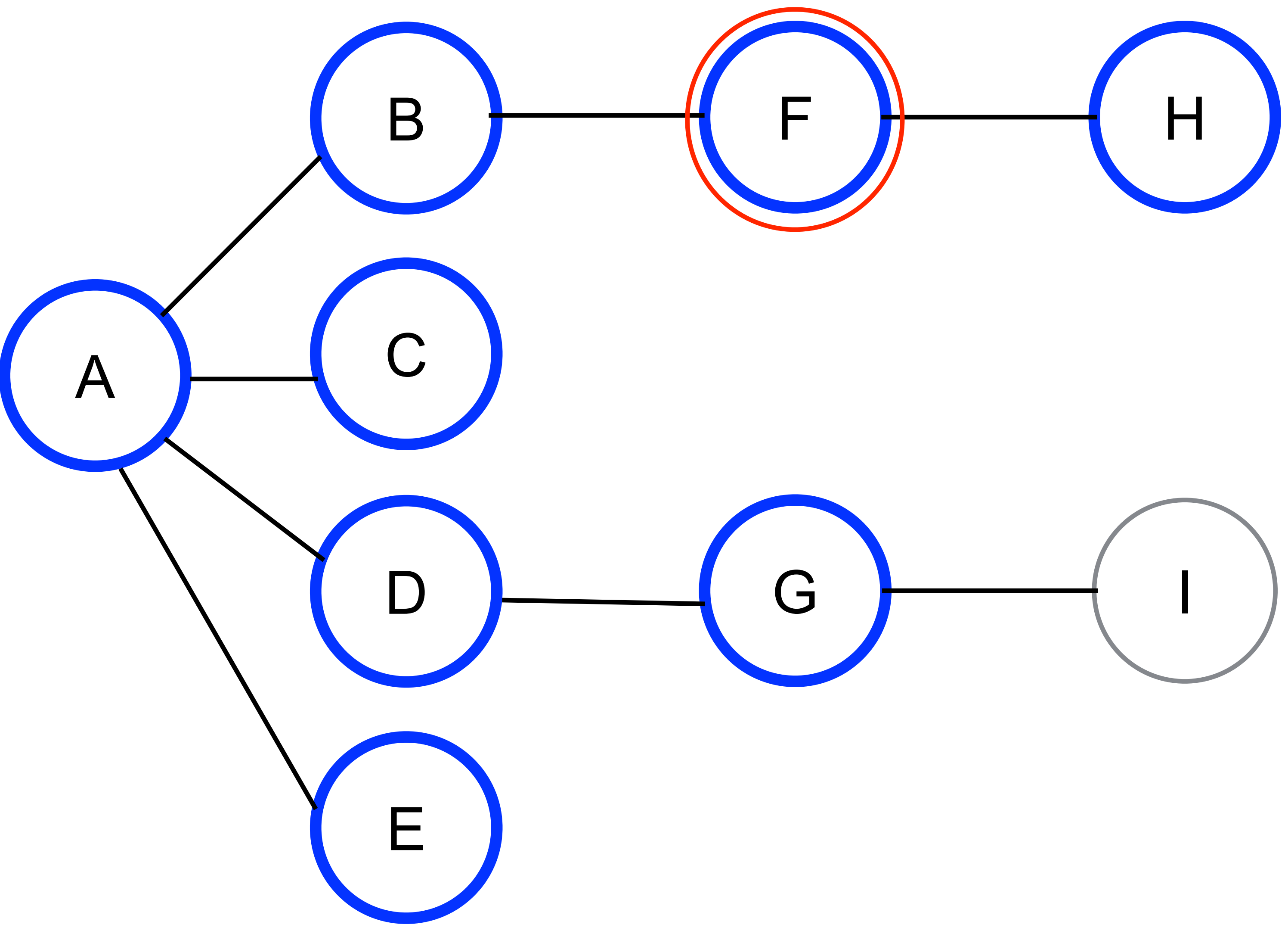

A
B
C
D
E
F
G
H
I
- current



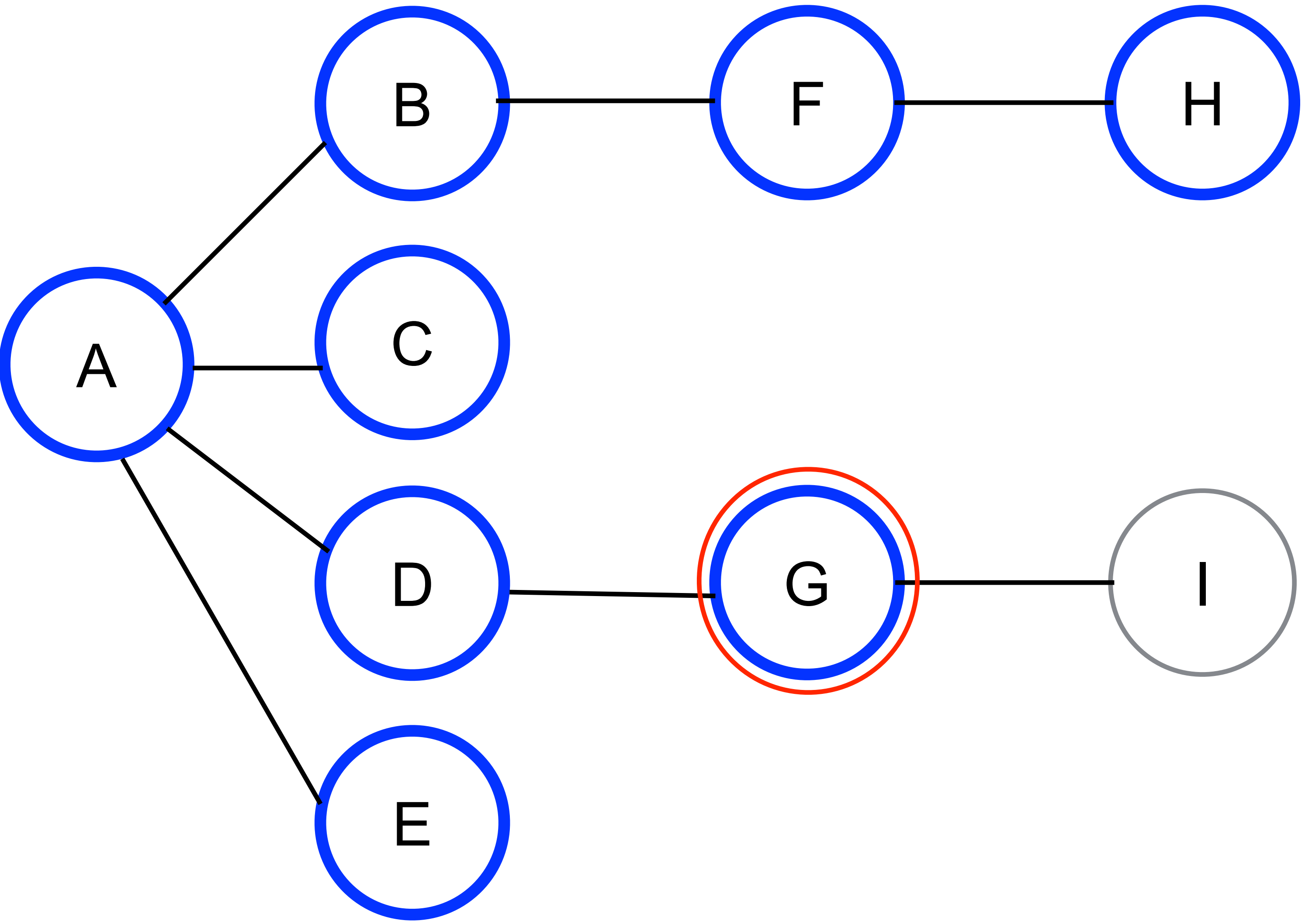
- current

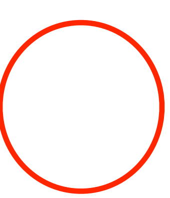

A
B
C
D
E
F
G
H
I
- current

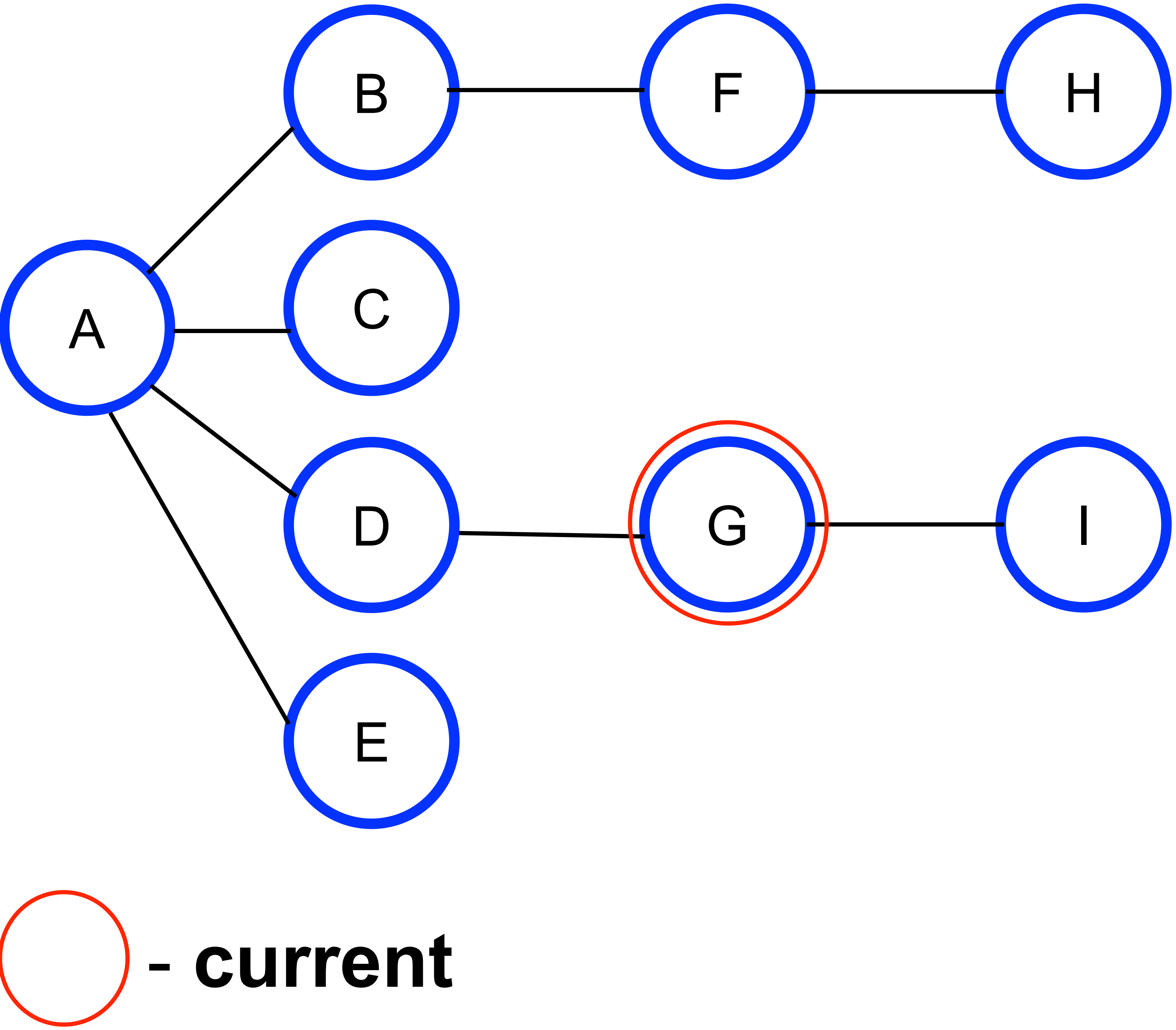

A
B
C
D
E
F
G
H
I
- current

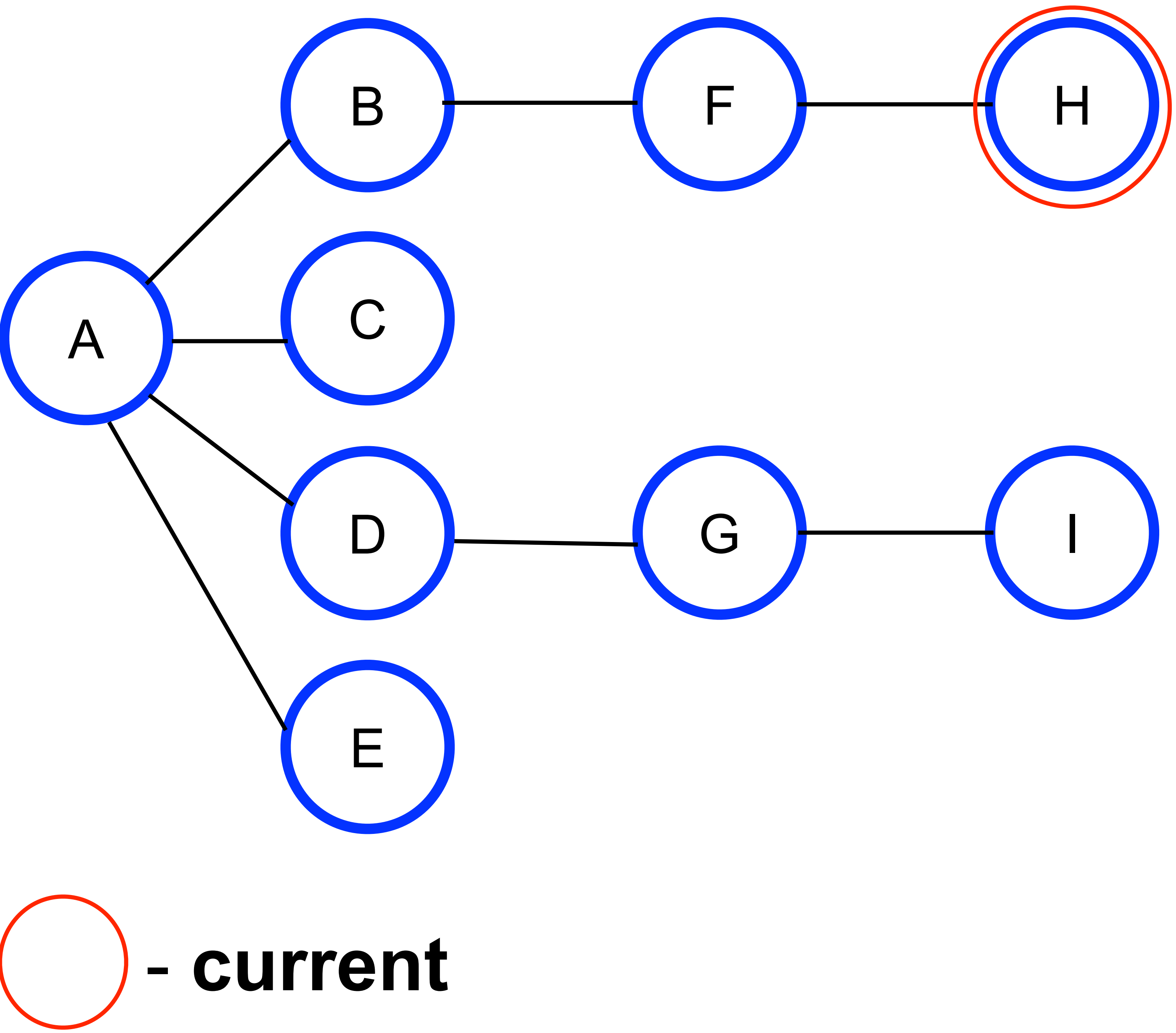


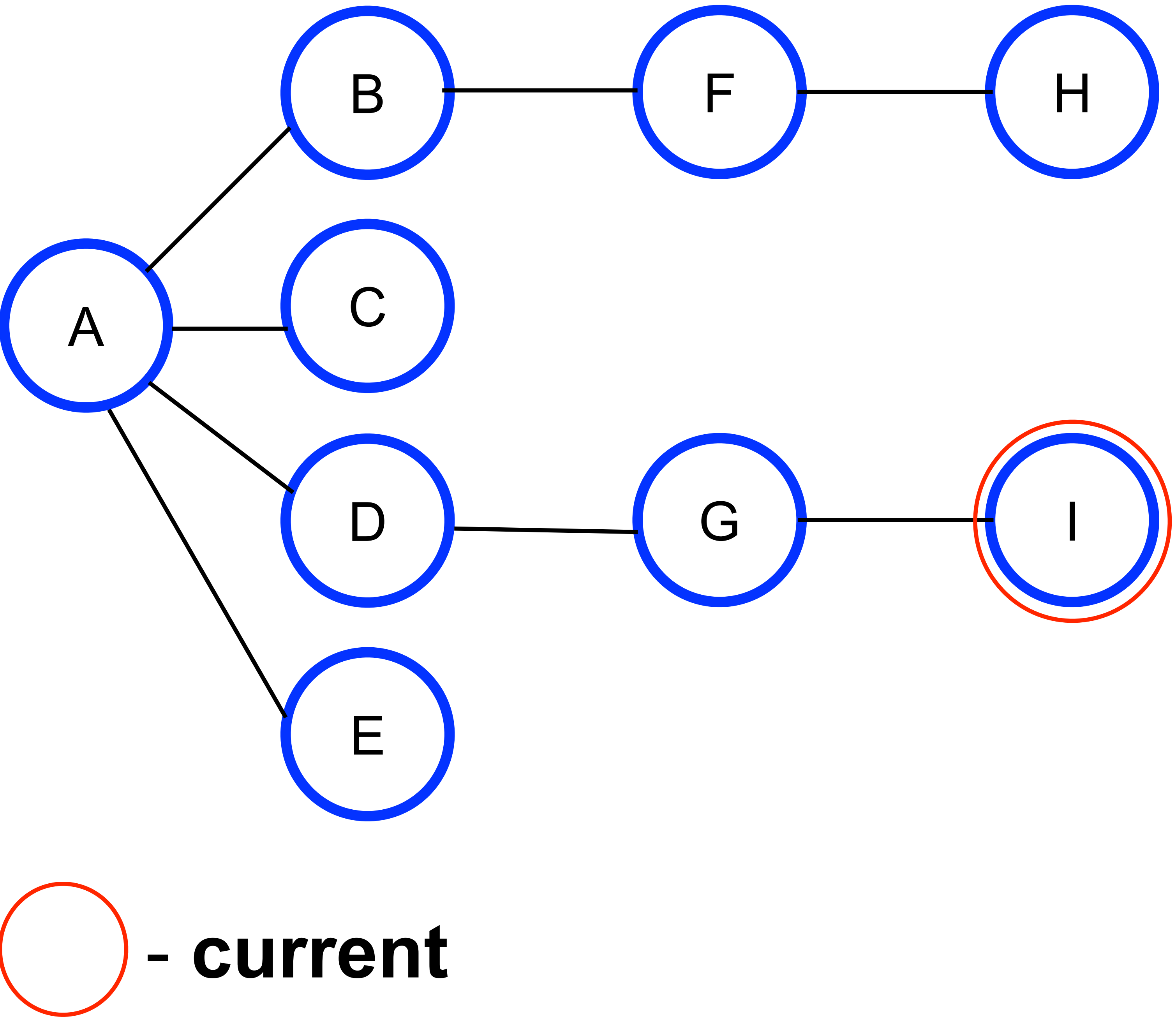
- current

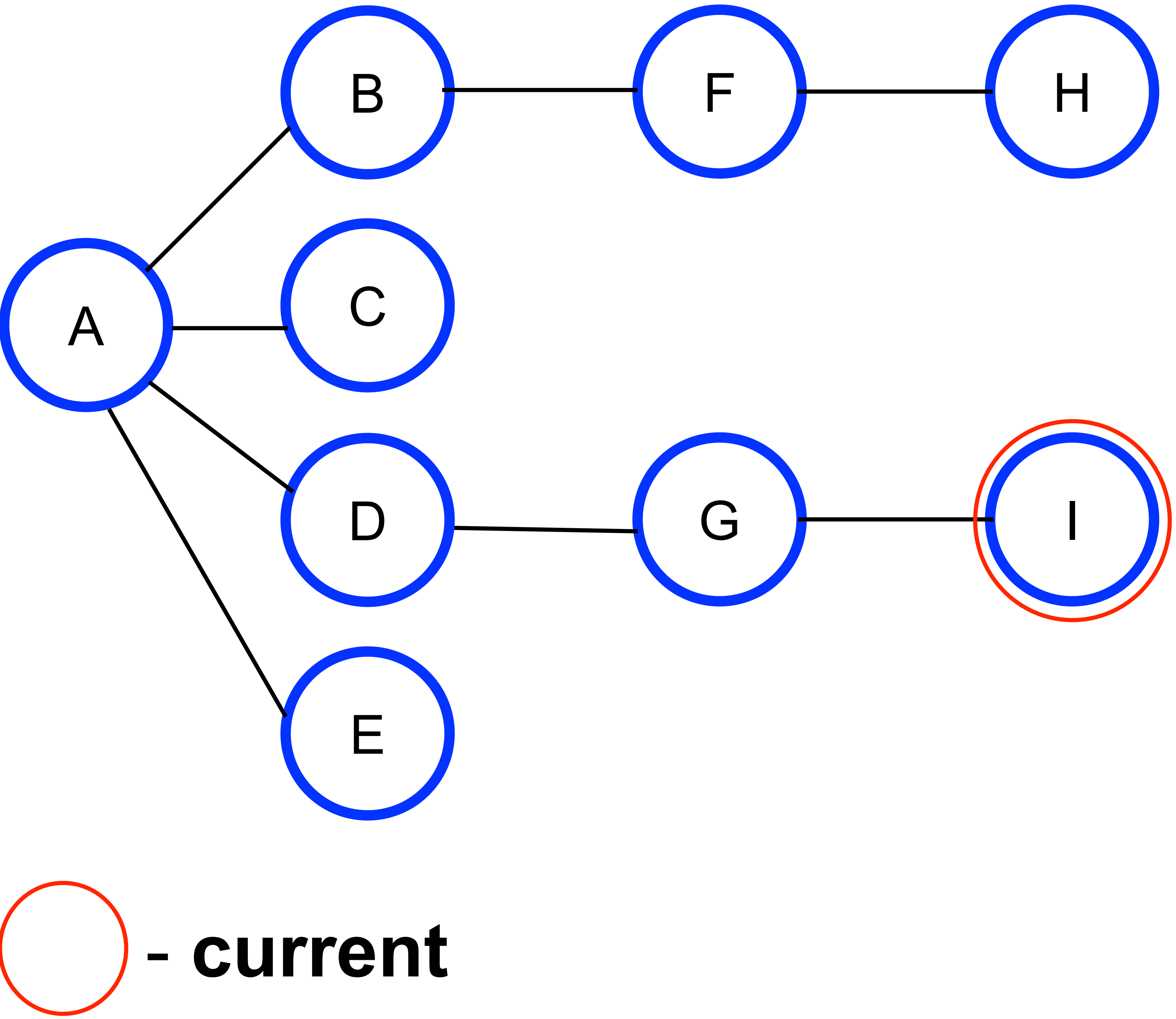


 **- current**

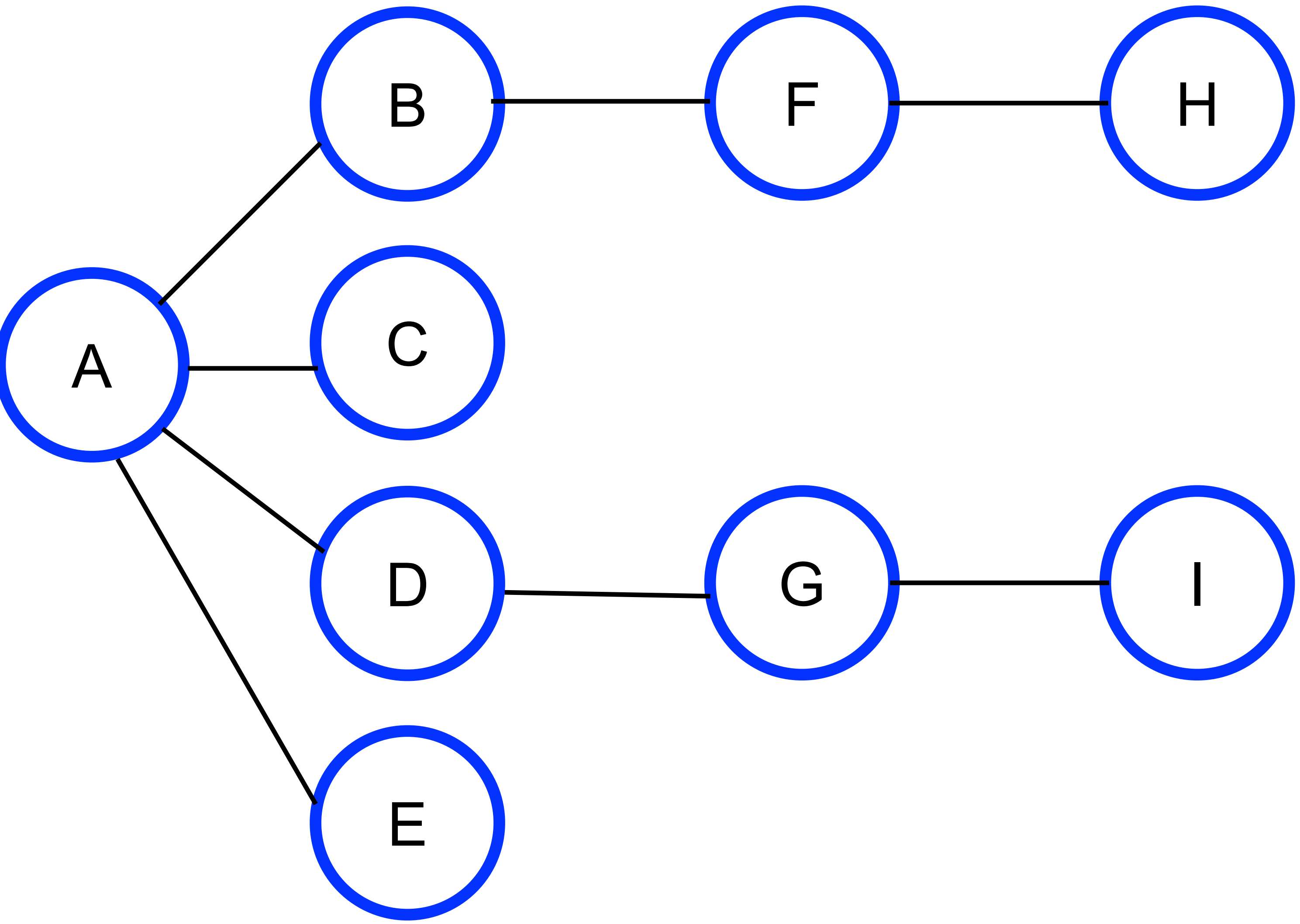

A
B
C
D
E
F
G
H
I
- current

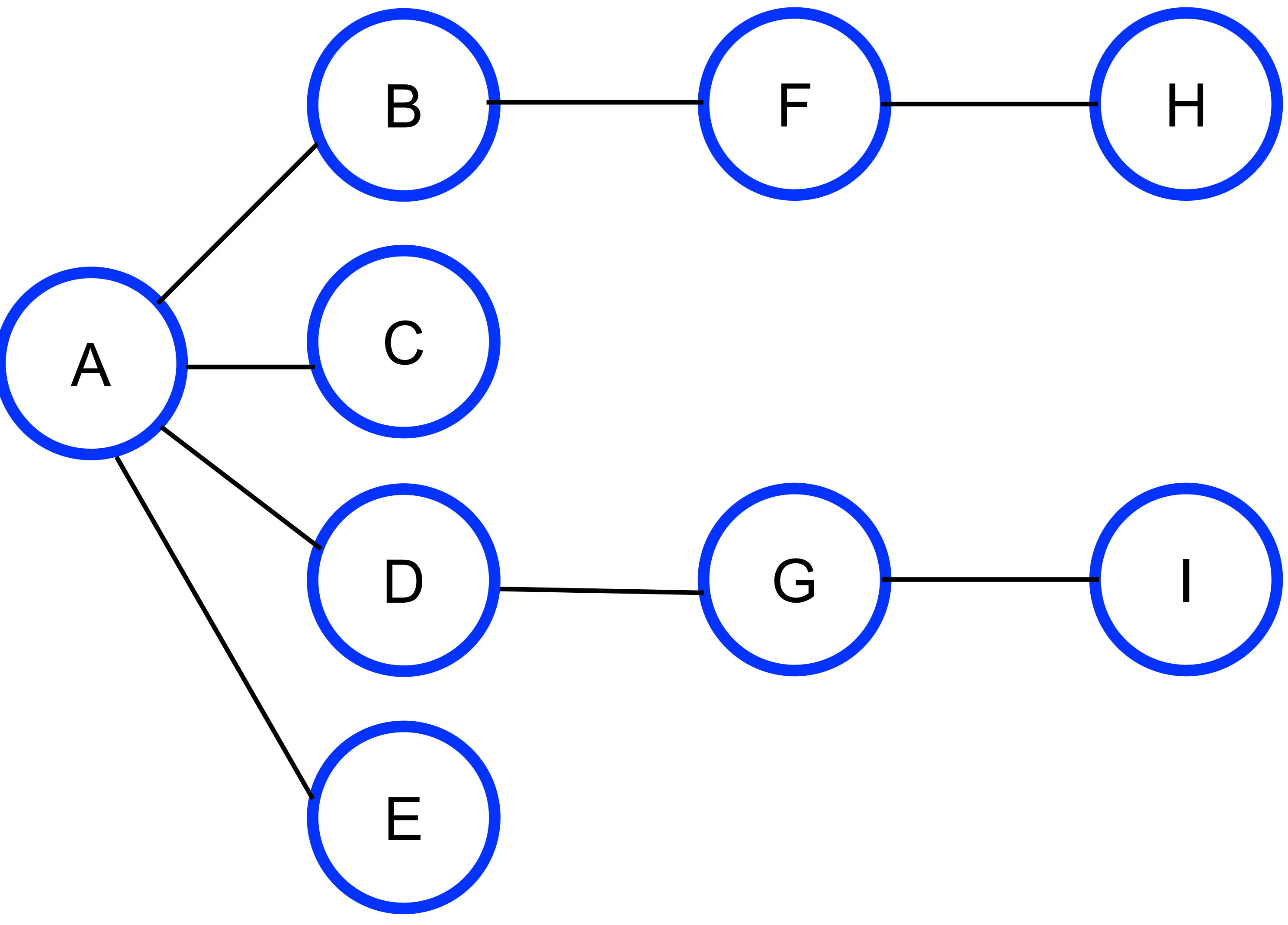

A
B
C
D
E
F
G
H
I
- current


A
B
C
D
E
F
G
H
I
- current



Now the queue is empty, so it is time for **Rule 3:**
"If you can't carry out Rule 2 because the queue is empty, you are finished"


A
B
C
D
E
F
G
H
I



Order: ABCDEFGHI

Time: $O(|V| + |E|)$

BFS

- Notice that,
  - BFS tries to stay as close as possible to the starting point
  - Thus the name, **Breadth First Search**
- Implementation of BFS is left as an exercise

BFS

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

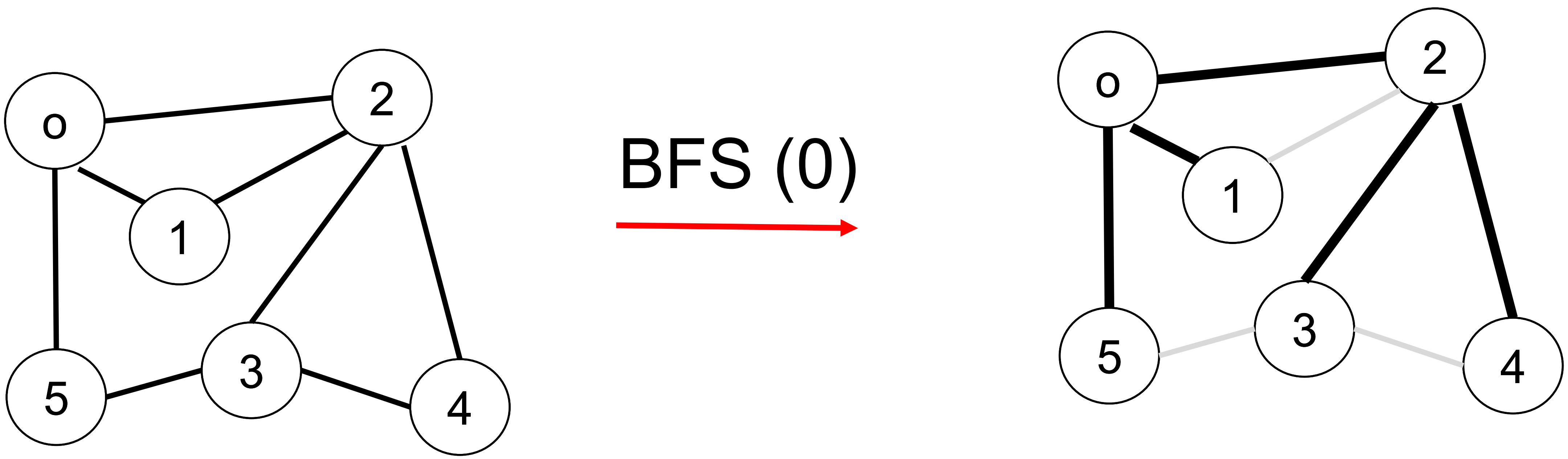
Cormen, Ch: 22

DFS & BFS

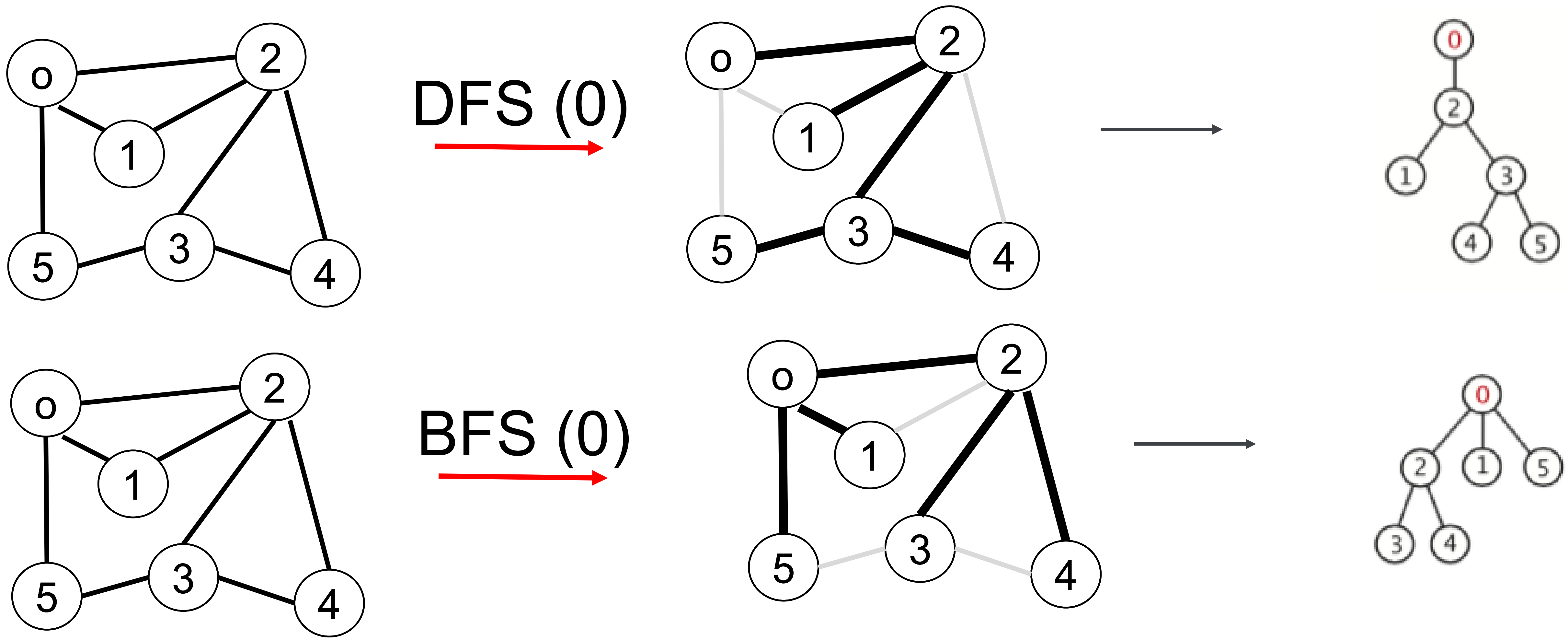
- Can be used to find:
  - whether there is a path between two vertices
  - whether a graph is connected
  - whether there is a cycle
  - connected components of a graph (slight modification or extension)

Final Remarks (1)

Final Remarks (2)



Final Remarks (3)



DFS finds a path, whereas BFS finds the shortest path (proof available in Cormen's: Chapter 22)

However, note that the graph is: unweighted (or same weight)

Did we achieve todays objectives?

1. Build a definition for the ”connected component of a graph”

2. Learn graph traversals

- Depth First Search (DFS)

- Breadth First Search (BFS)