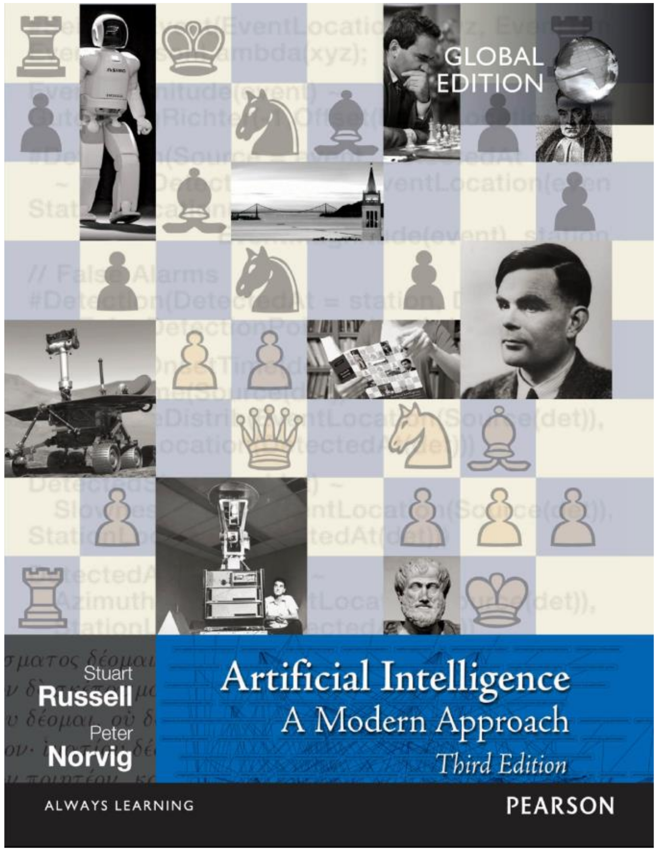
Artificial Intelligence

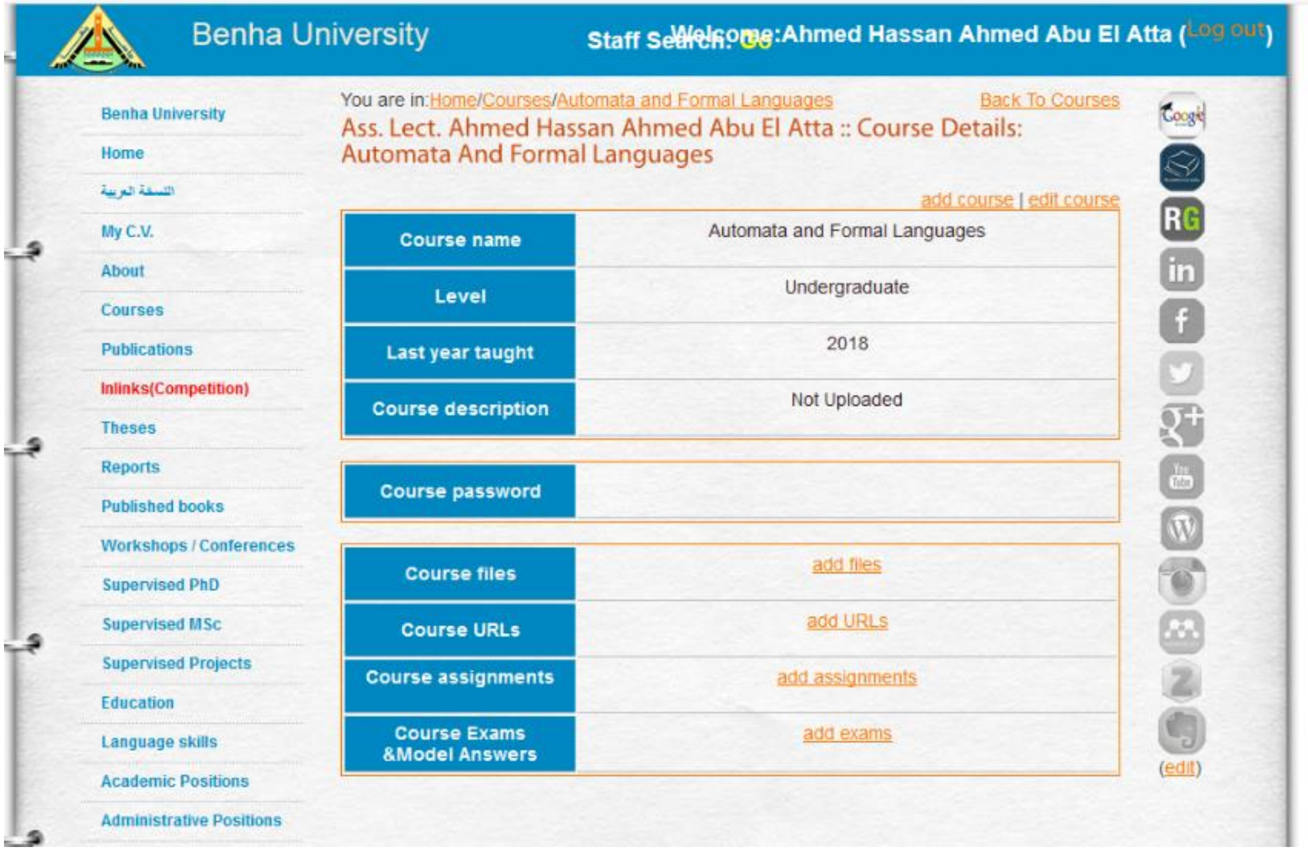
Lecture 02

Books


GLOBAL EDITION
Stuart Russell
Peter Norvig
Artificial Intelligence
A Modern Approach
Third Edition
ALWAYS LEARNING
PEARSON

PowerPoint

http://www.bu.edu.eg/staff/ahmedaboalatah14-courses/14767



You are in: Home/Courses/Automata and Formal Languages

Ass. Lect. Ahmed Hassan Ahmed Abu El Atta :: Course Details: Automata And Formal Languages

Course name	Automata and Formal Languages
Level	Undergraduate
Last year taught	2018
Course description	Not Uploaded

Course password	

Course files	add files
Course URLs	add URLs
Course assignments	add assignments
Course Exams &Model Answers	add exams

Problem-solving

SOLVING PROBLEMS BY SEARCHING I

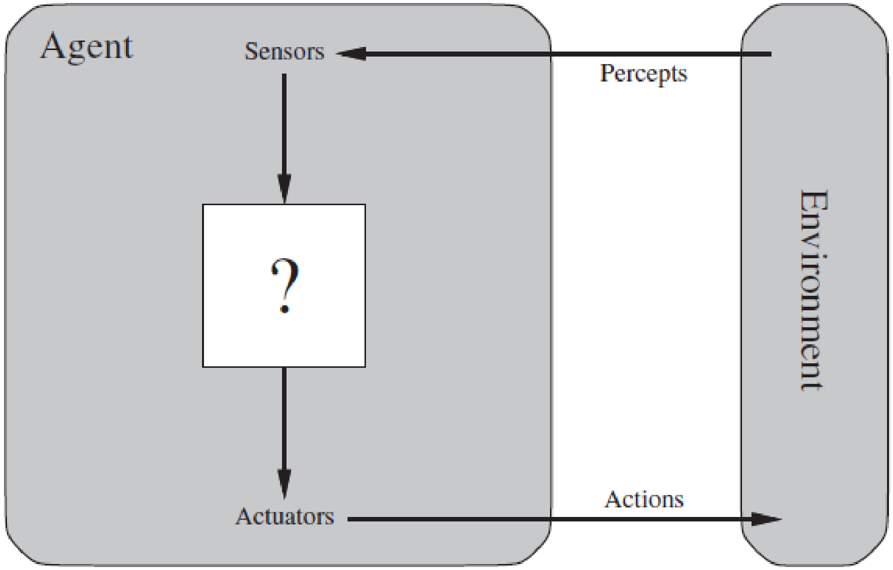
Agent and Environment

An **agent** is something that perceives and acts in an environment.

The **agent function** for an agent specifies the action taken by the agent in response to any percept sequence.

The **performance measure** evaluates the behavior of the agent in an environment.

A **rational agent** acts so as to maximize the expected value of the performance measure, given the percept sequence it has seen so far.



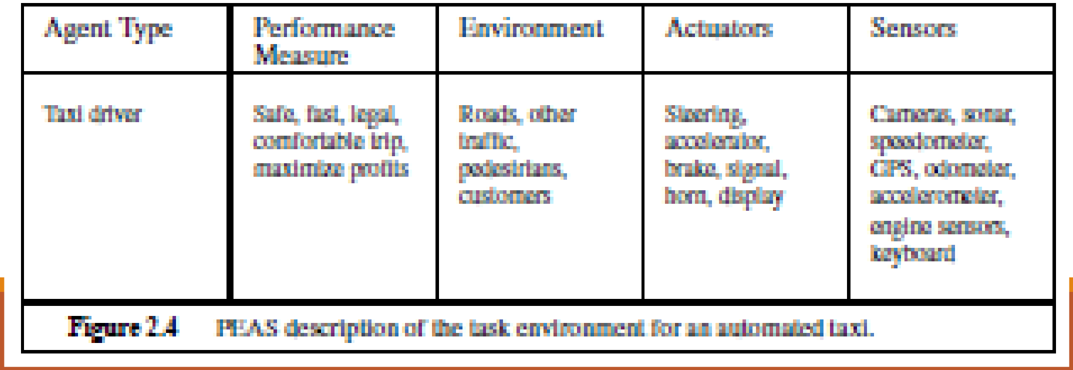
Agent and Environment

A **task environment** specification includes the performance measure, the external environment, the actuators, and the sensors.

The **agent program** implements the agent function.

- There exists a variety of basic agent-program designs reflecting the kind of information made explicit and used in the decision process.

Goal-based agents act to achieve their goals.



Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

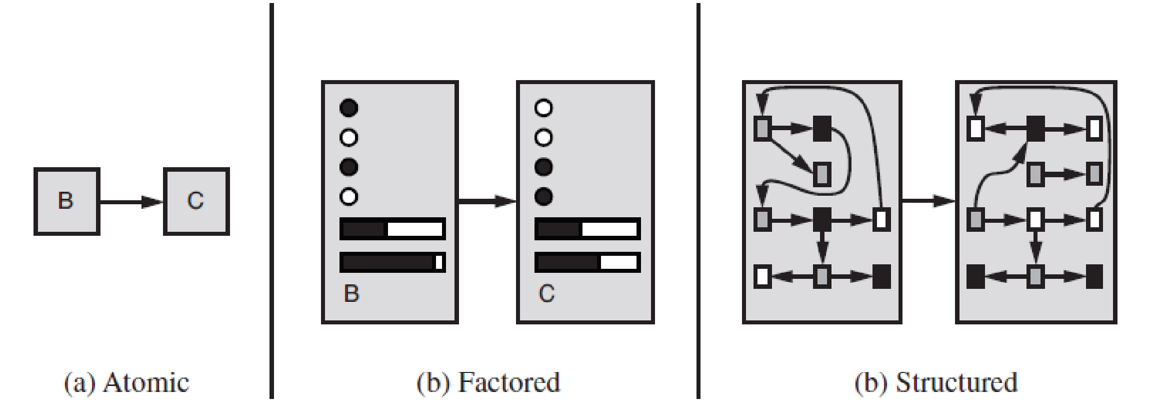
Figure 2.4 PEAS description of the task environment for an automated taxi.

Environment Representations
atomic, **factored**, and **structured**

(a) Atomic representation: a state (such as B or C) is a black box with no internal structure;

(b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols.

(c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.



PROBLEM-SOLVING AGENTS

A **problem-solving agent is** one kind of goal-based agent.

- Problem-solving agents use atomic representations—that is, states of the world are considered as wholes, with no internal structure visible to the problem solving algorithms.

Intelligent agents are supposed to maximize their performance measure.

Achieving this is sometimes simplified if the agent can adopt a **goal** and aim at satisfying it.

PROBLEM-SOLVING AGENTS

Goal formulation is the first step in problem solving. (based on the current situation and the agent’s performance measure)

Consider a **goal** to be a **set of world states**—exactly those states in which the goal is satisfied.

- The agent’s task is to find out how to act, now and in the future, so that it reaches a goal state.

Problem formulation is the process of deciding what actions and states to consider, given a goal.

The solution to any problem is a fixed sequence of actions.

The process of looking for a sequence of actions that reaches the goal is called **search**.

A problem can be defined formally by five components:

The **initial state** that the agent starts in.

A description of the possible **actions** available to the agent.

- Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions is **applicable** in s.

A description of what each action does; the formal name for this is the **transition model**,

- specified by a function RESULT(s, a) that returns the state that results from doing action a in state s. We also use the term successor to refer to any state reachable from a given state by a single action.

The **goal test**, which determines whether a given state is a goal state.

- Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.

A **path cost function** that assigns a numeric cost to each path.

- The problem-solving agent chooses a cost function that reflects its own performance measure.

EXAMPLE PROBLEMS

The vacuum-cleaner

The vacuum-cleaner world shown in figure.

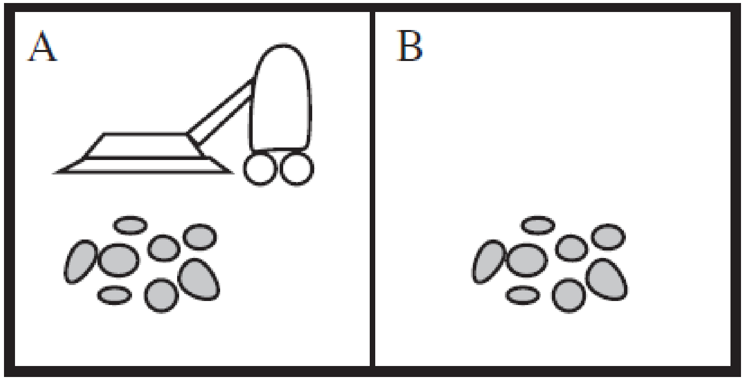
This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations.

This particular world has just two locations: squares A and B.

The vacuum agent perceives which square it is in and whether there is dirt in the square.

It can choose to move left, move right, suck up the dirt, or do nothing.

One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square.



The vacuum-cleaner

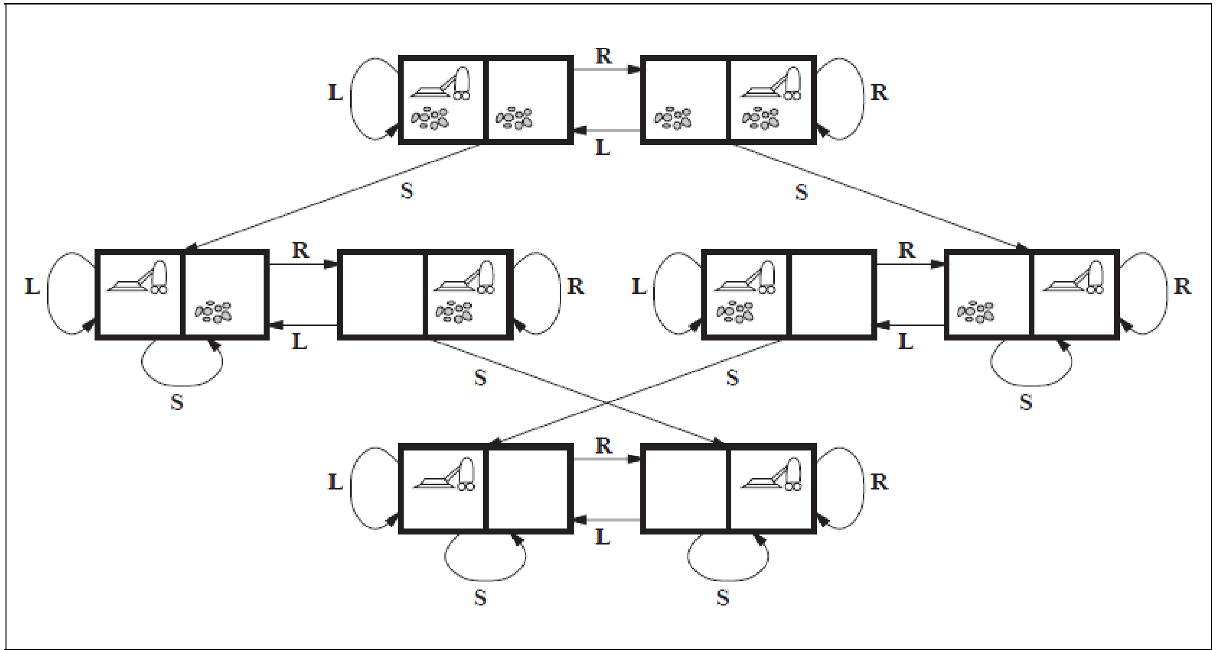


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

States: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are 2 × 2^2 = 8 possible world states.

The state space for the vacuum world

Initial state: Any state can be designated as the initial state.

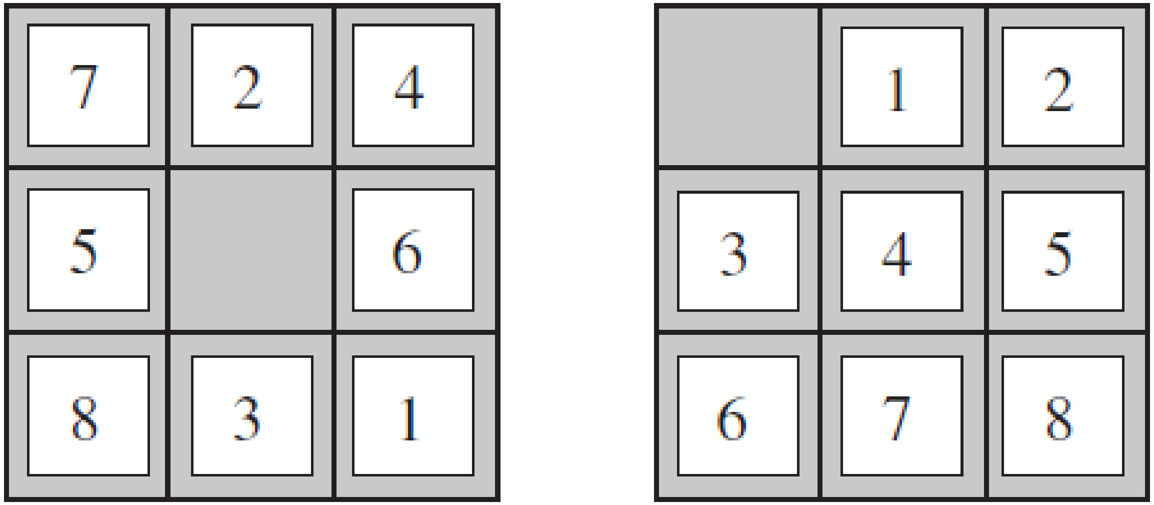
Actions: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.

Transition model: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.

Goal test: This checks whether all the squares are clean.

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

The 8-puzzle



7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

The 8-puzzle has 9!/2=181, 440 reachable states and is easily solved.

The 8-puzzle

Initial state: Any state can be designated as the initial state.

Actions: The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.

Transition model: Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.

Goal test: This checks whether the state matches the goal configuration.

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

The 8-puzzle

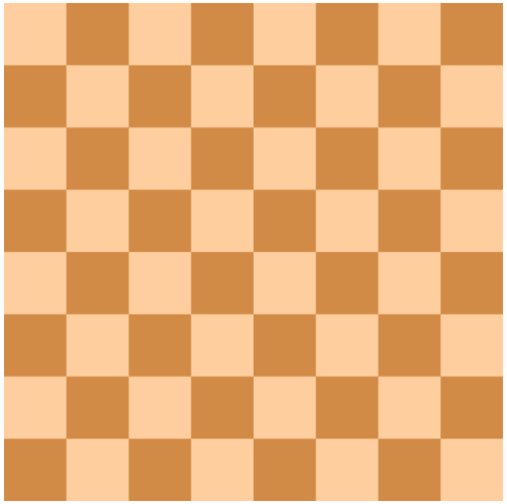
This family is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next.

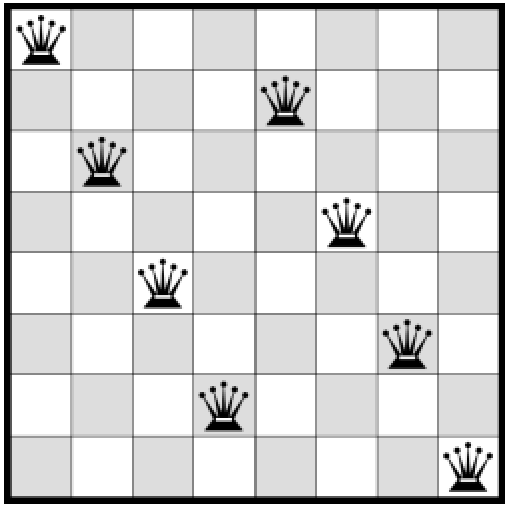
The 8-puzzle has 9!/2=181, 440 reachable states and is easily solved.

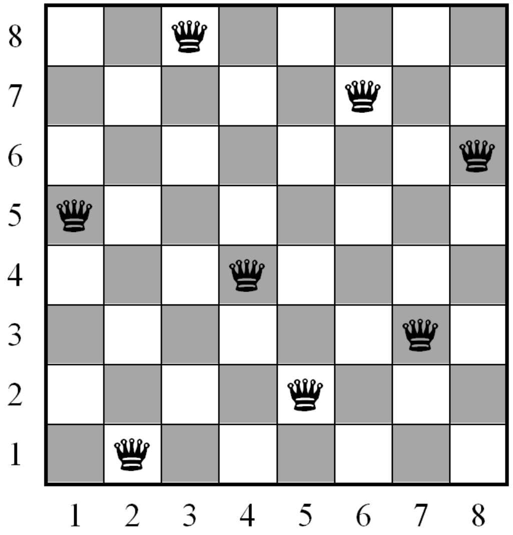
The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms.

The 24-puzzle (on a 5 × 5 board) has around 10^{25} states, and random instances take several hours to solve optimally.

8-queens problem







The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other.

States: Any arrangement of 0 to 8 queens on the board is a state.

8-queens problem

Initial state: No queens on the board.

Actions: Add a queen to any empty square.

Transition model: Returns the board with a queen added to the specified square.

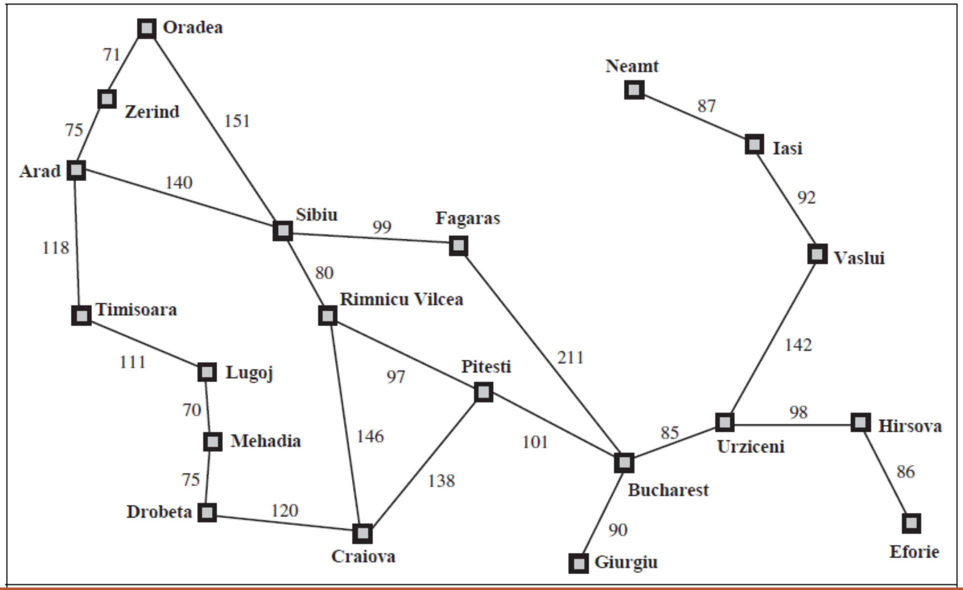
Goal test: 8 queens are on the board, none attacked.

In this formulation, we have $64! / (64-8)! = 64 \cdot 63 \cdot \cdot \cdot 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate.

Real-world problems route-finding problem

States: Each state obviously includes a location (e.g., an airport) and the current time.

Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.



Real-world problems route-finding problem

Initial state: This is specified by the user’s query.

Actions: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.

Transition model: The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.

Goal test: Are we at the final destination specified by the user?

Path cost: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

SEARCHING FOR SOLUTIONS

The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem.

Search Problem

Search space : The set of objects (sequence of states) among which we search for the solution.

Goal condition : What are the characteristics of the object (goal) we want to find in the search space.

Graph Search Problem

States : game positions, or locations in the map that are represented by nodes in the graph.

Initial state : start position.

Goal state : target position/s (node/s).

Actions and transition : connections between graph nodes.

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

Search Strategies

UNINFORMED SEARCH STRATEGIES (BLIND SEARCH)

- The term means that the strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the order in which nodes are expanded.

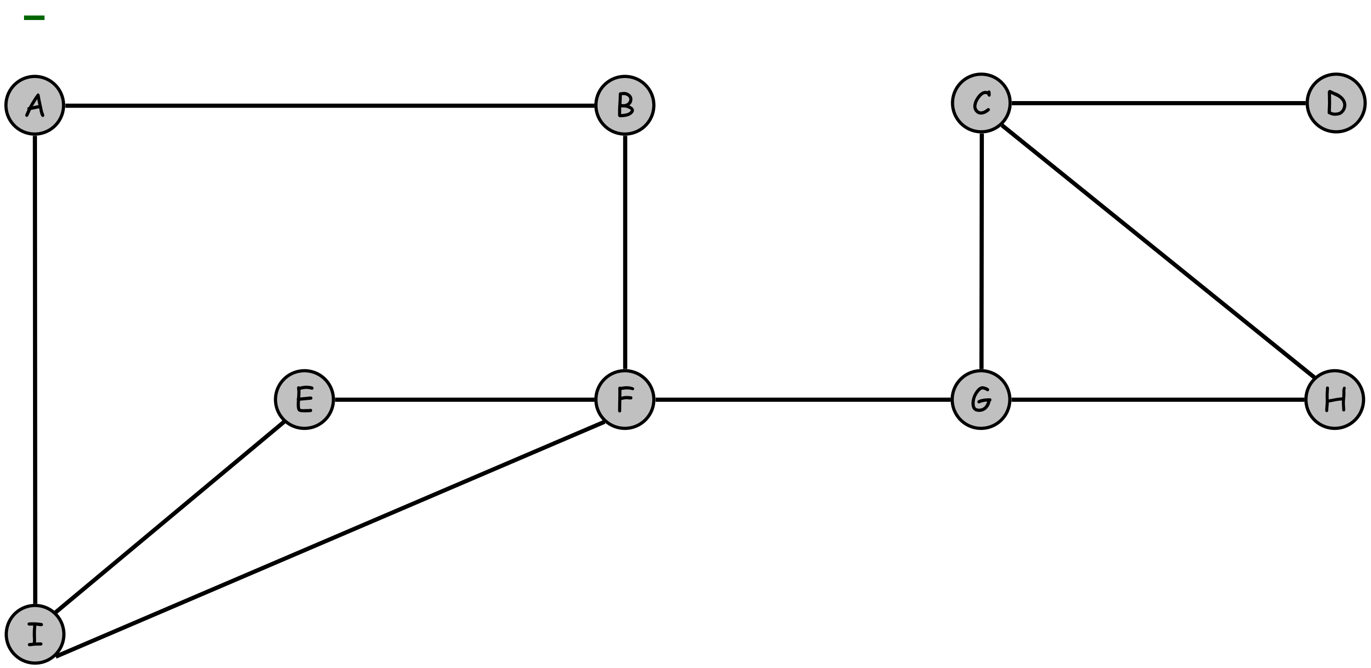
INFORMED (HEURISTIC) SEARCH STRATEGIES

- one that uses problem specific knowledge beyond the definition of the problem itself (node is selected for expansion based on an **evaluation function**).

Breadth-first search

```
procedure BFS(G, s)
    let Q be a queue
    label s as discovered
    Q.enqueue(s)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered then
                label w as discovered
                w.parent := v
                Q.enqueue(w)
```

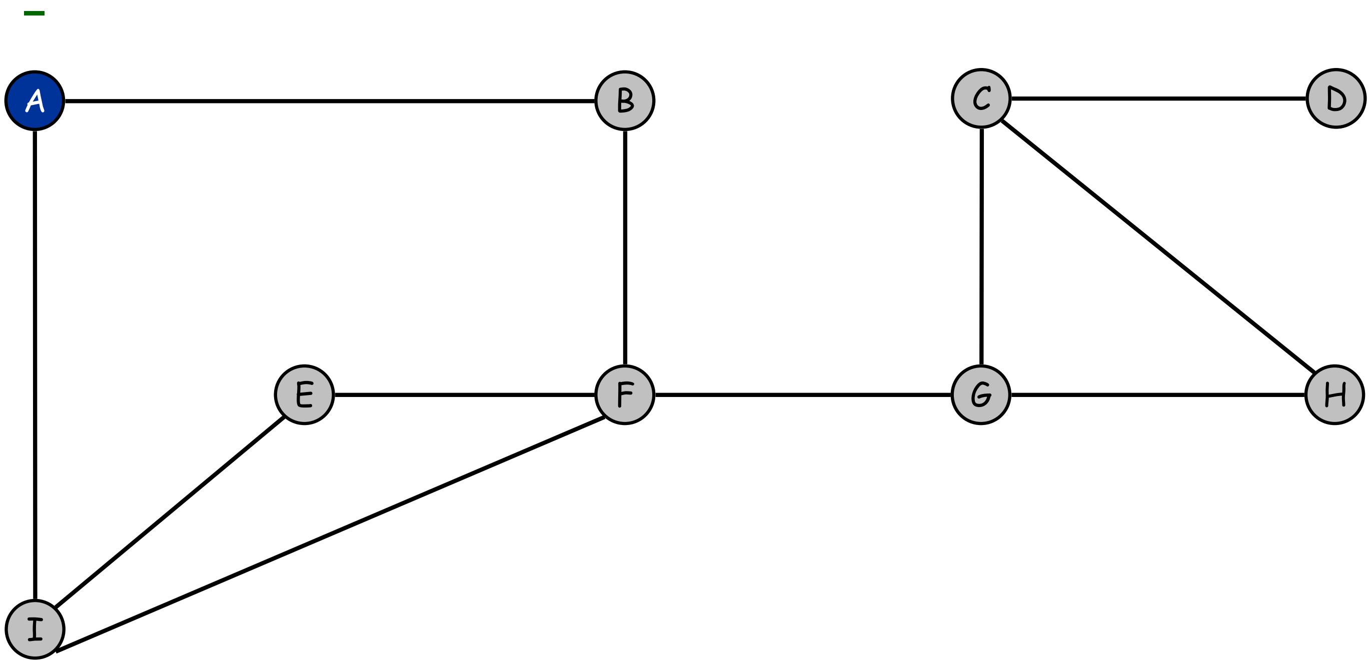
Breadth First Search



front

FIFO Queue

Breadth First Search

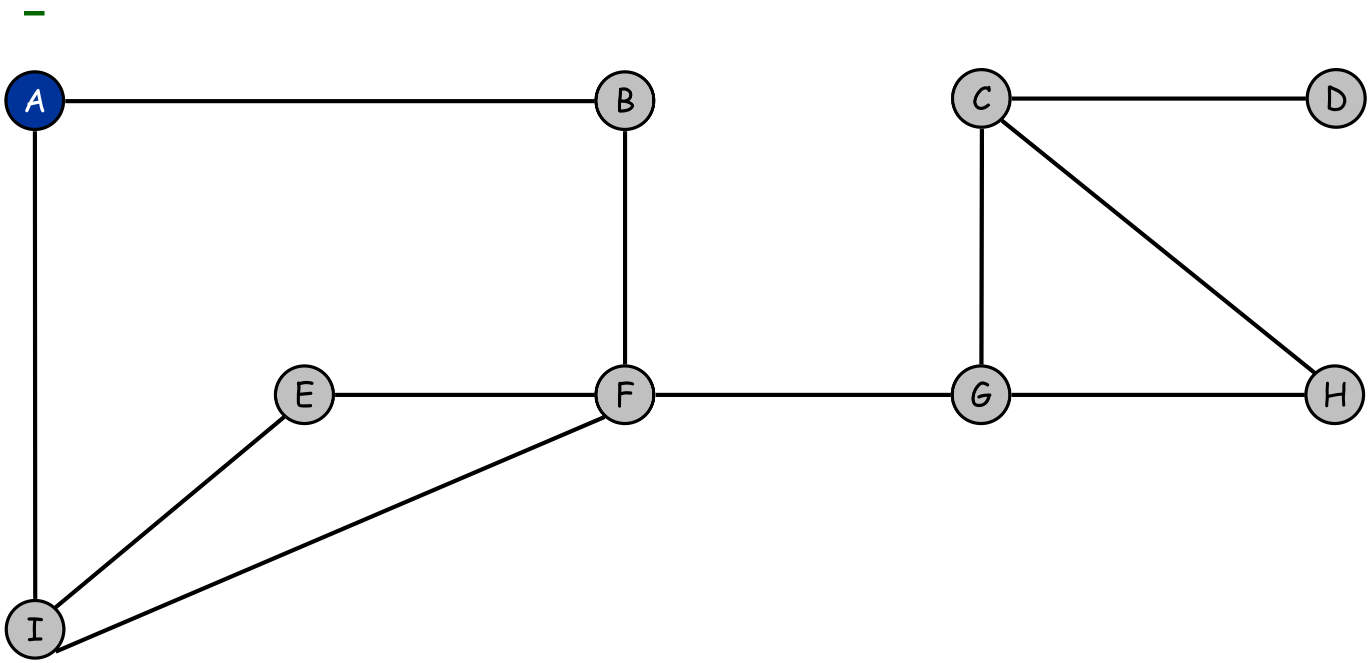


enqueue source node

front **A**

FIFO Queue

Breadth First Search

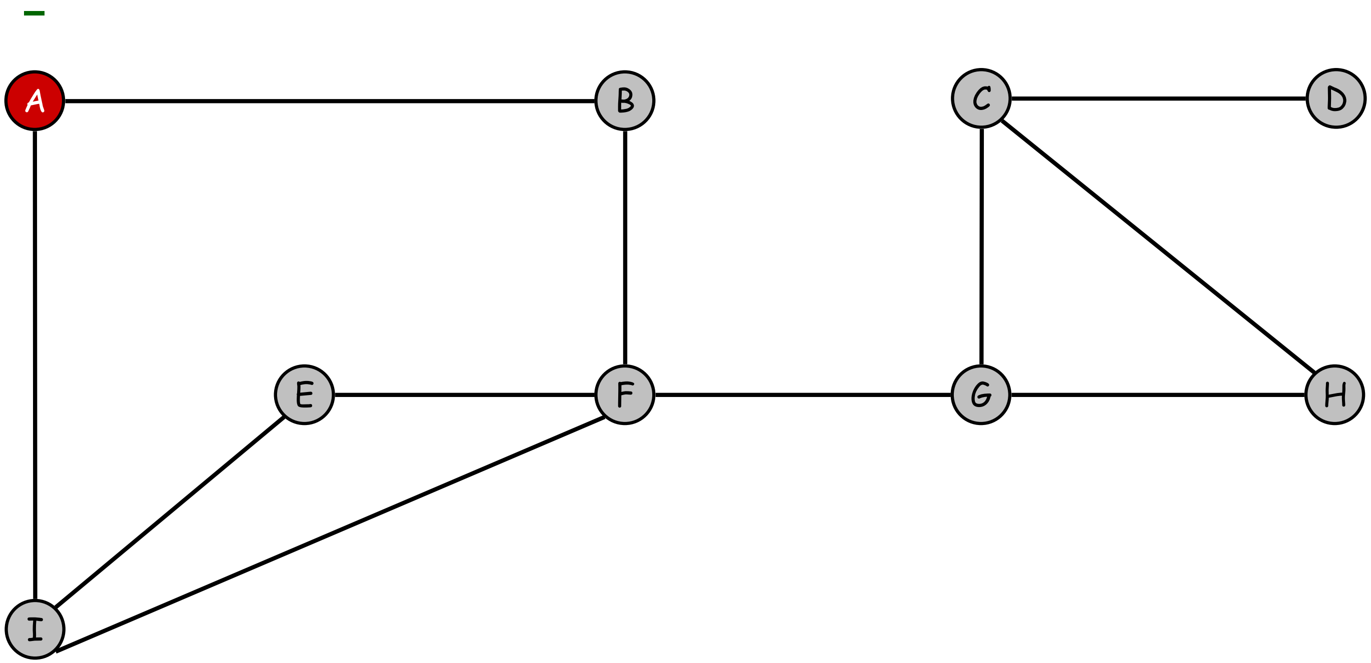


dequeue next vertex

front **A**

FIFO Queue

Breadth First Search

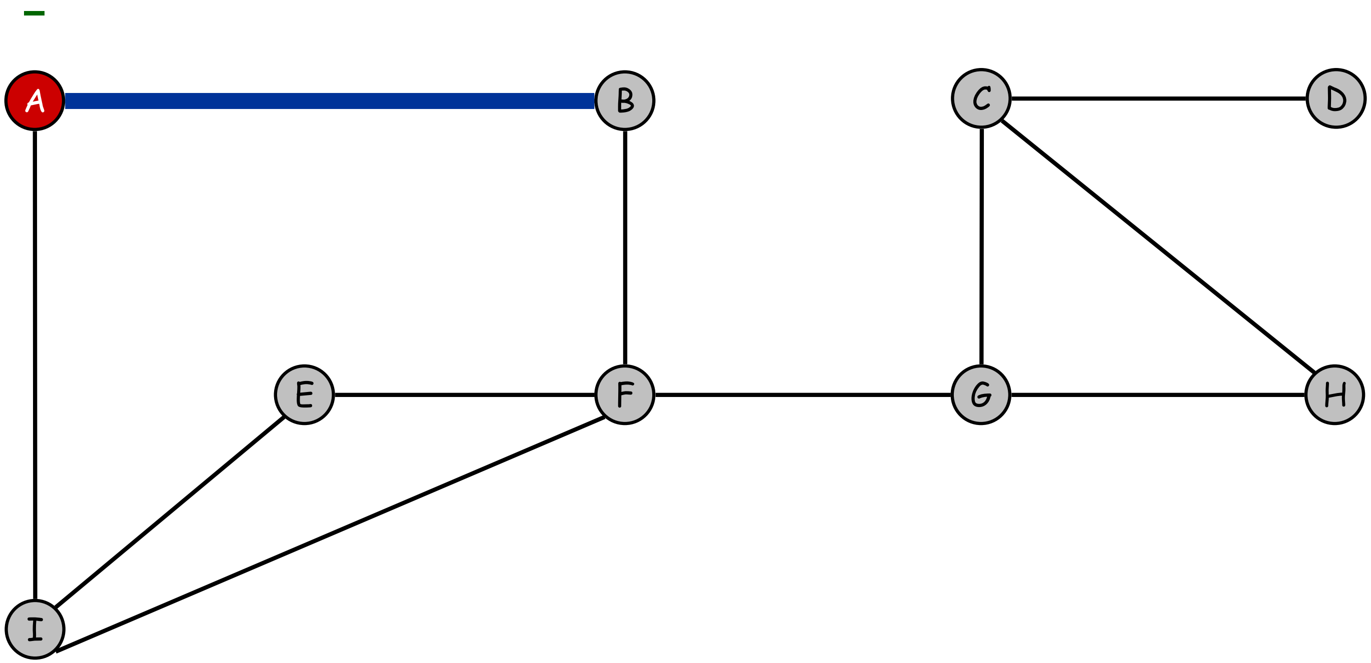


visit neighbors of A

front

FIFO Queue

Breadth First Search

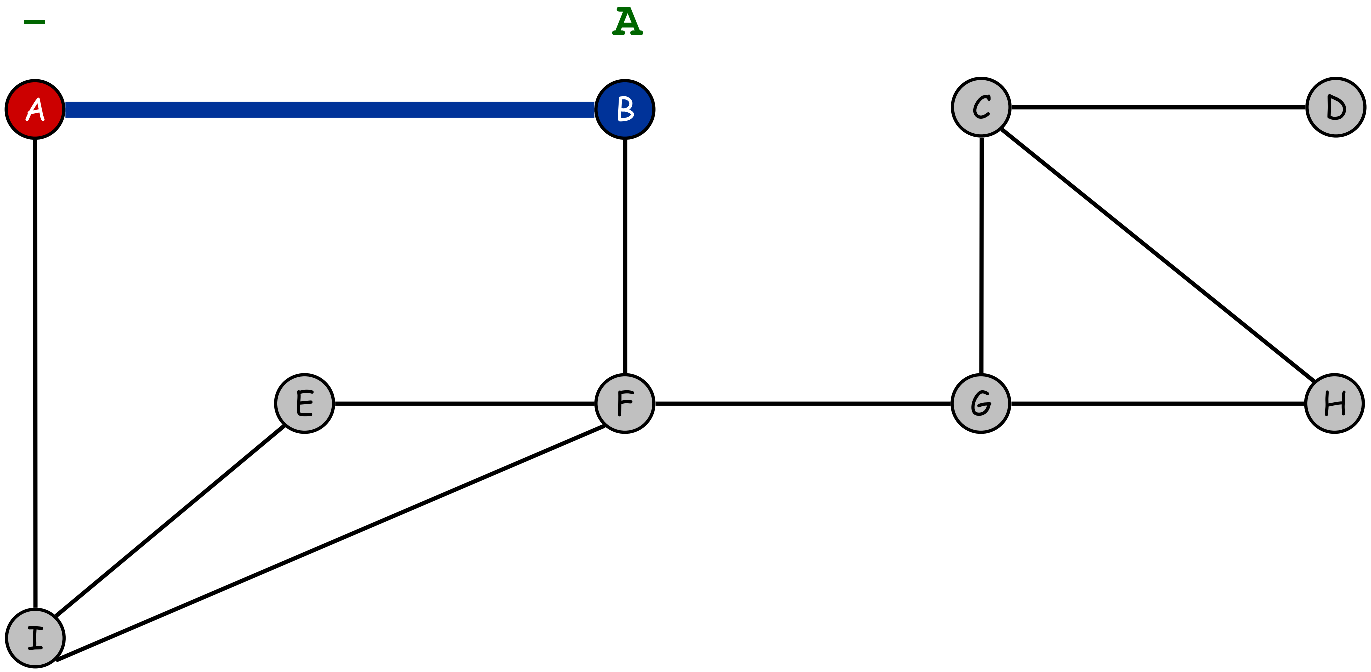


visit neighbors of A

front

FIFO Queue

Breadth First Search

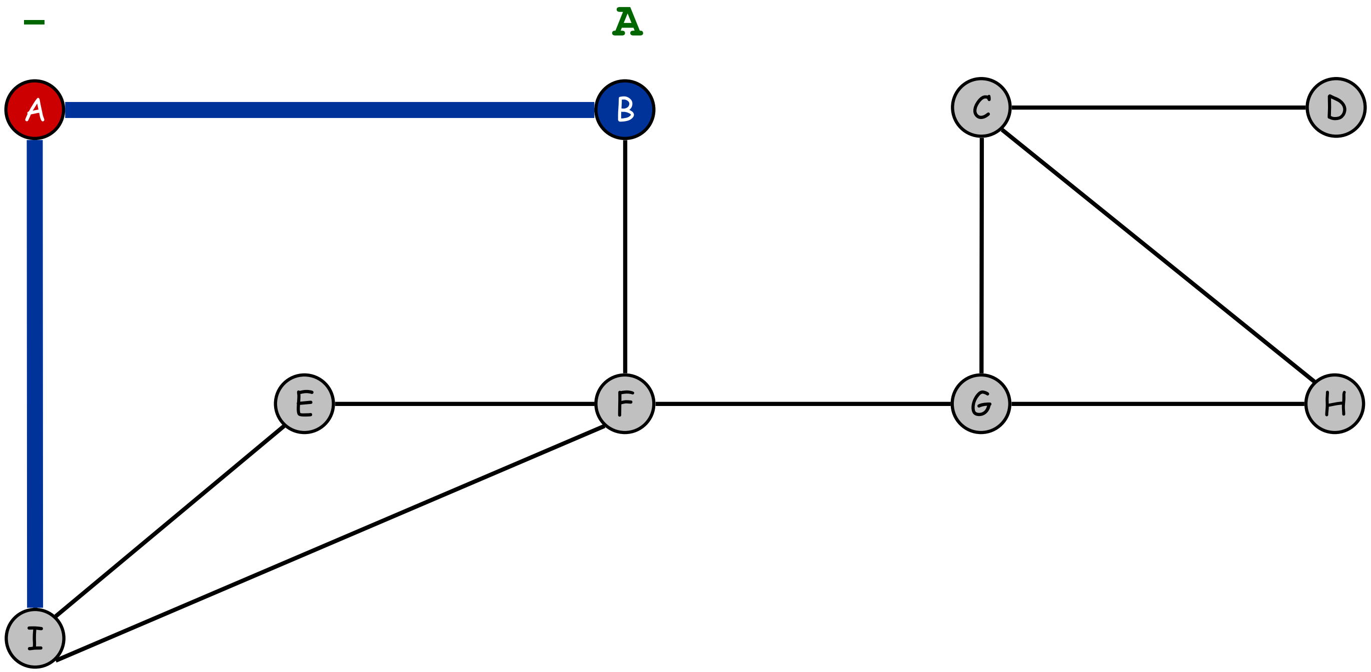


B discovered

front **B**

FIFO Queue

Breadth First Search



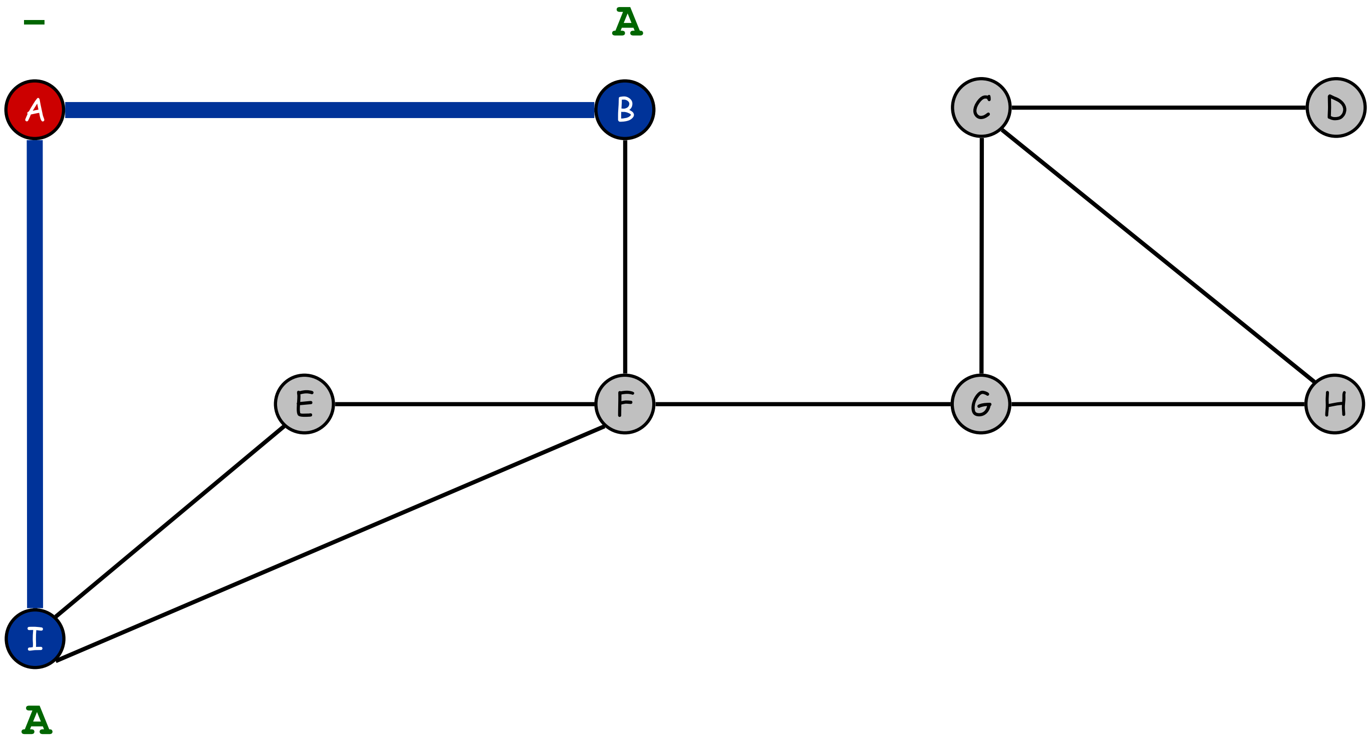
visit neighbors of A

front

B

FIFO Queue

Breadth First Search

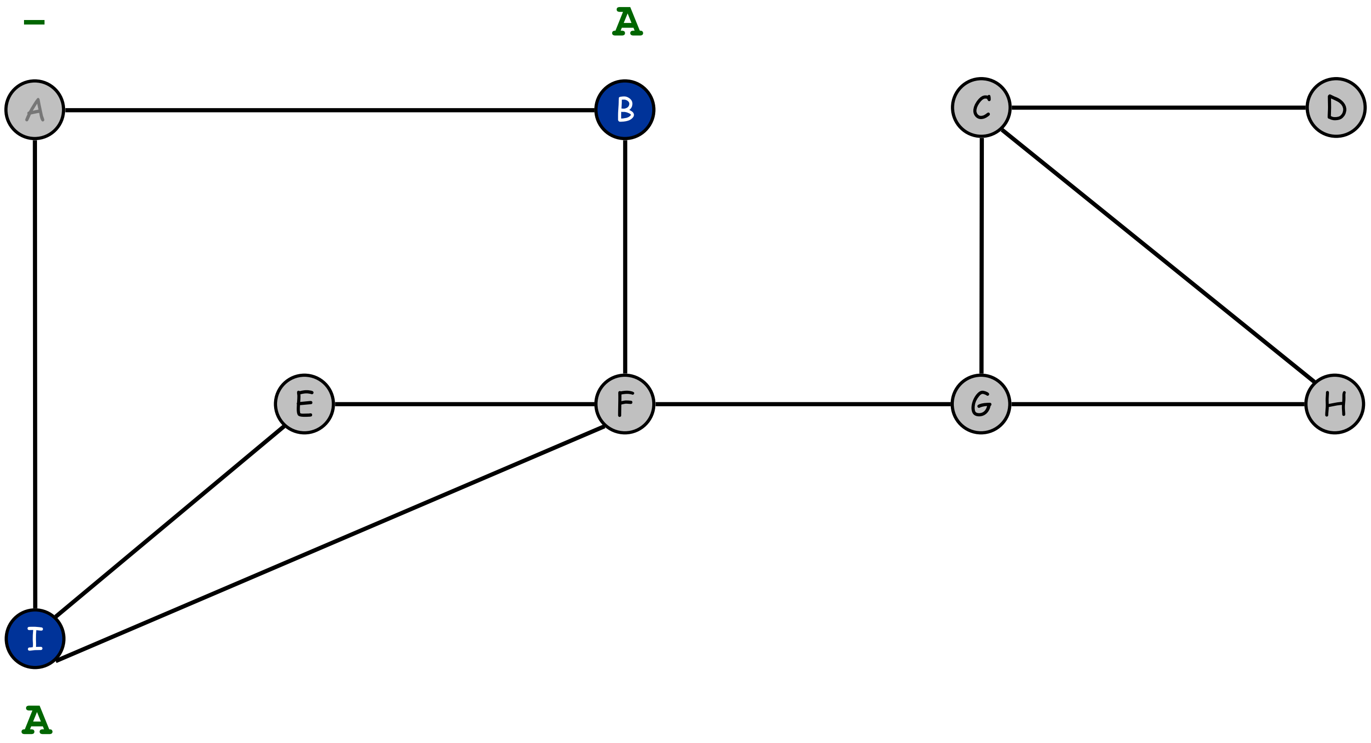


I discovered

front B I

FIFO Queue

Breadth First Search

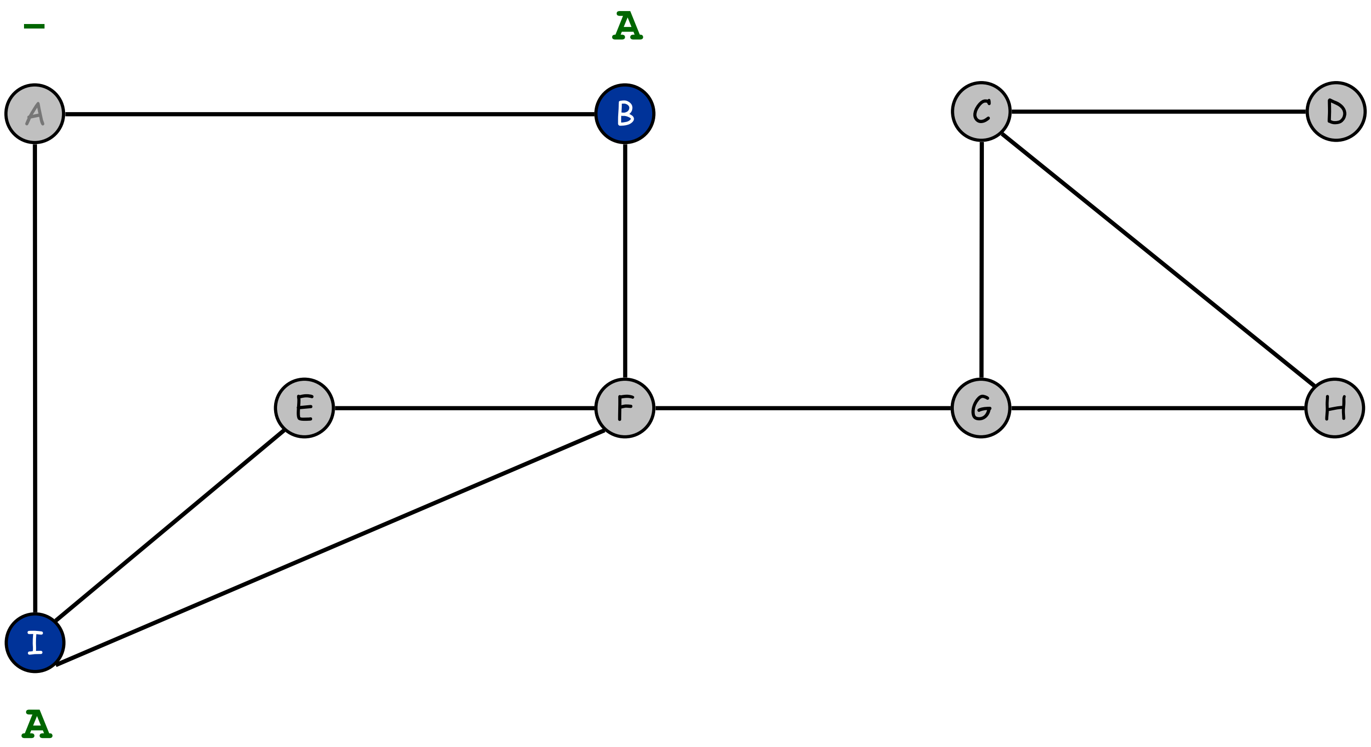


finished with A

front **B I**

FIFO Queue

Breadth First Search

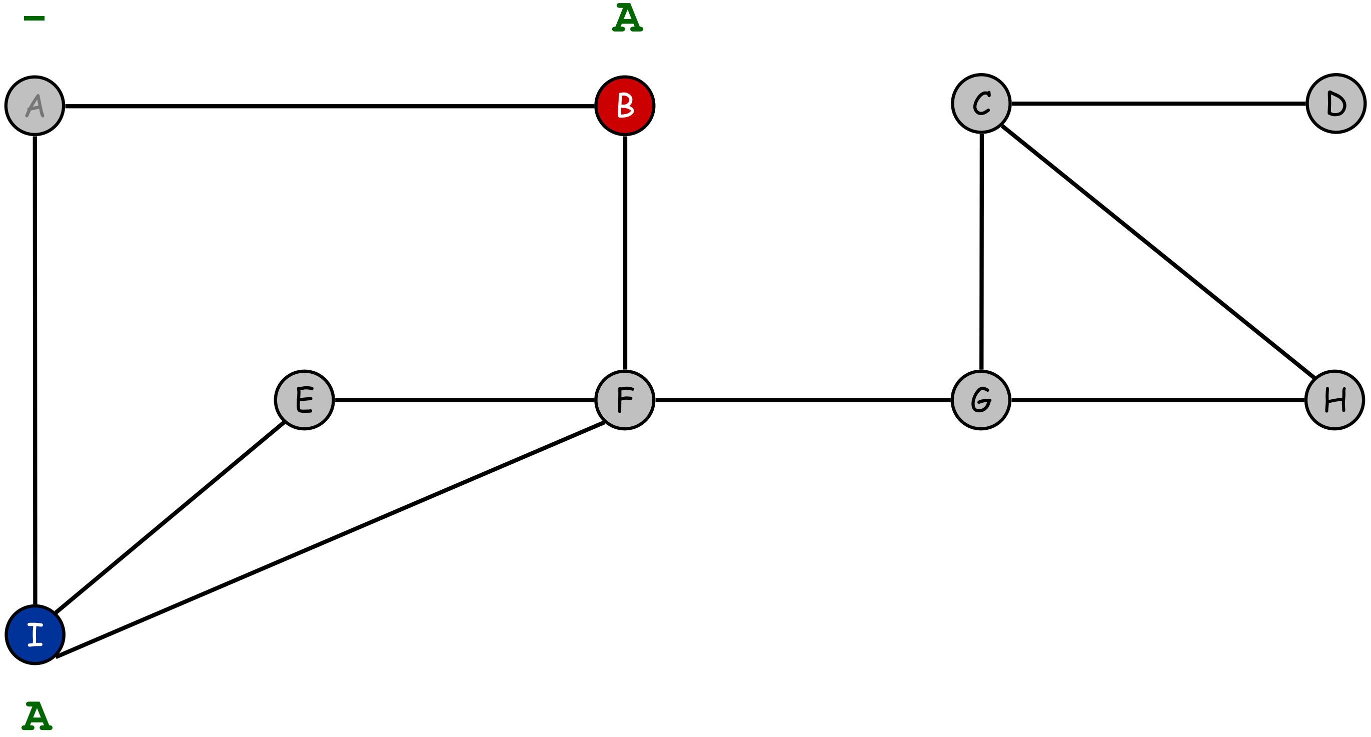


dequeue next vertex

front **B I**

FIFO Queue

Breadth First Search

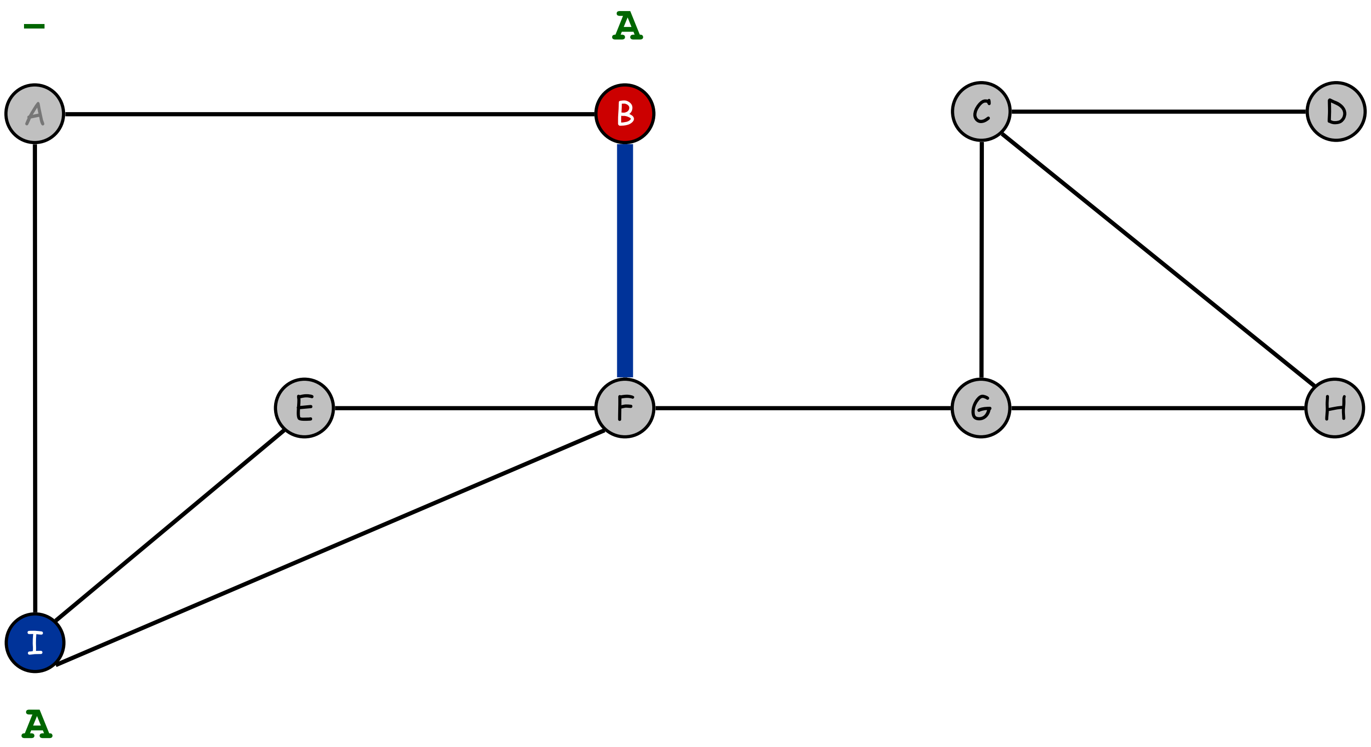


visit neighbors of B

front **I**

FIFO Queue

Breadth First Search

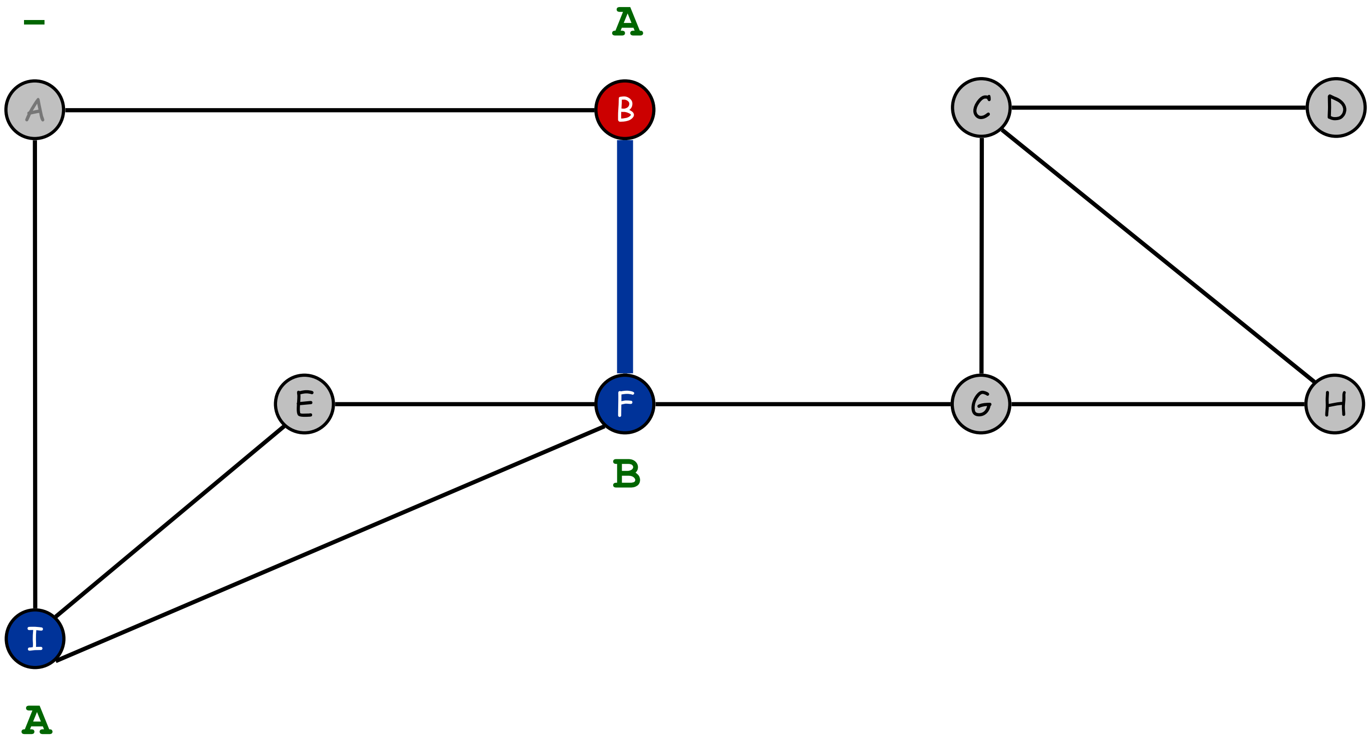


visit neighbors of B

front I

FIFO Queue

Breadth First Search

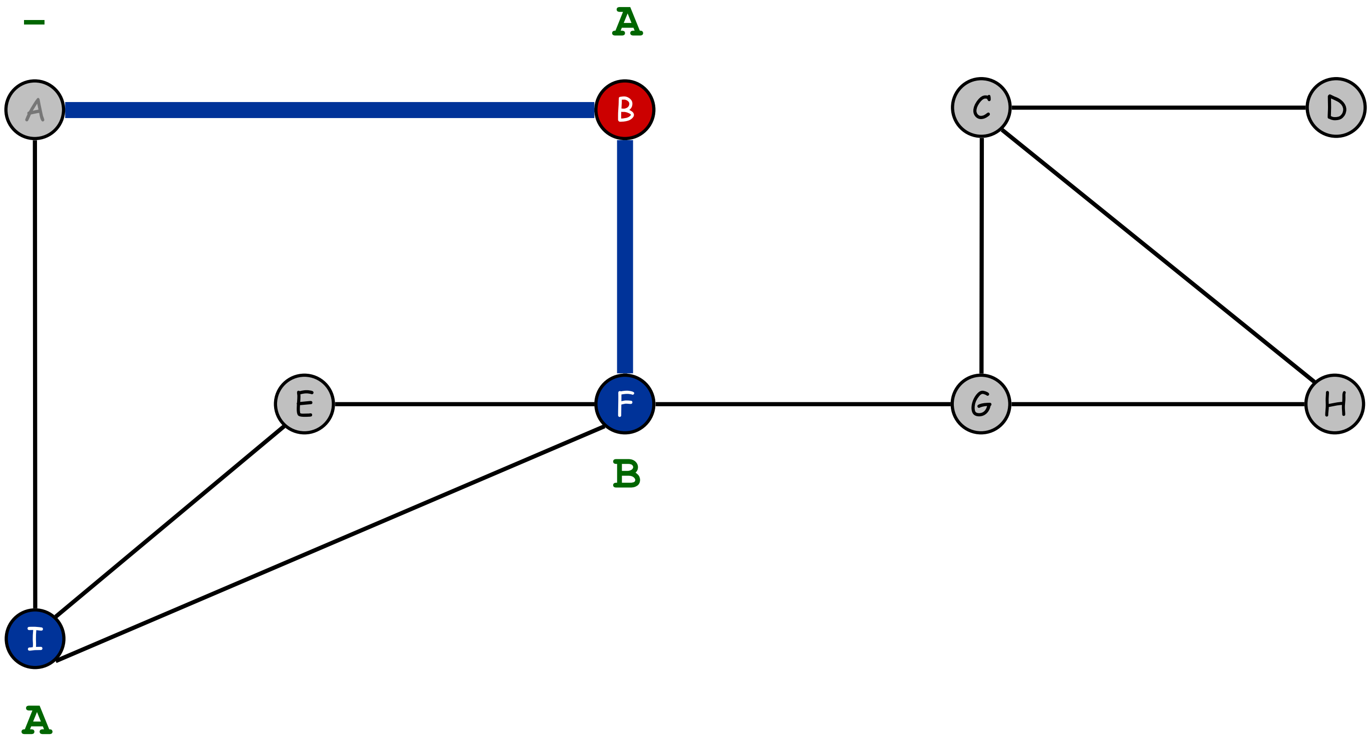


F discovered

front **I F**

FIFO Queue

Breadth First Search



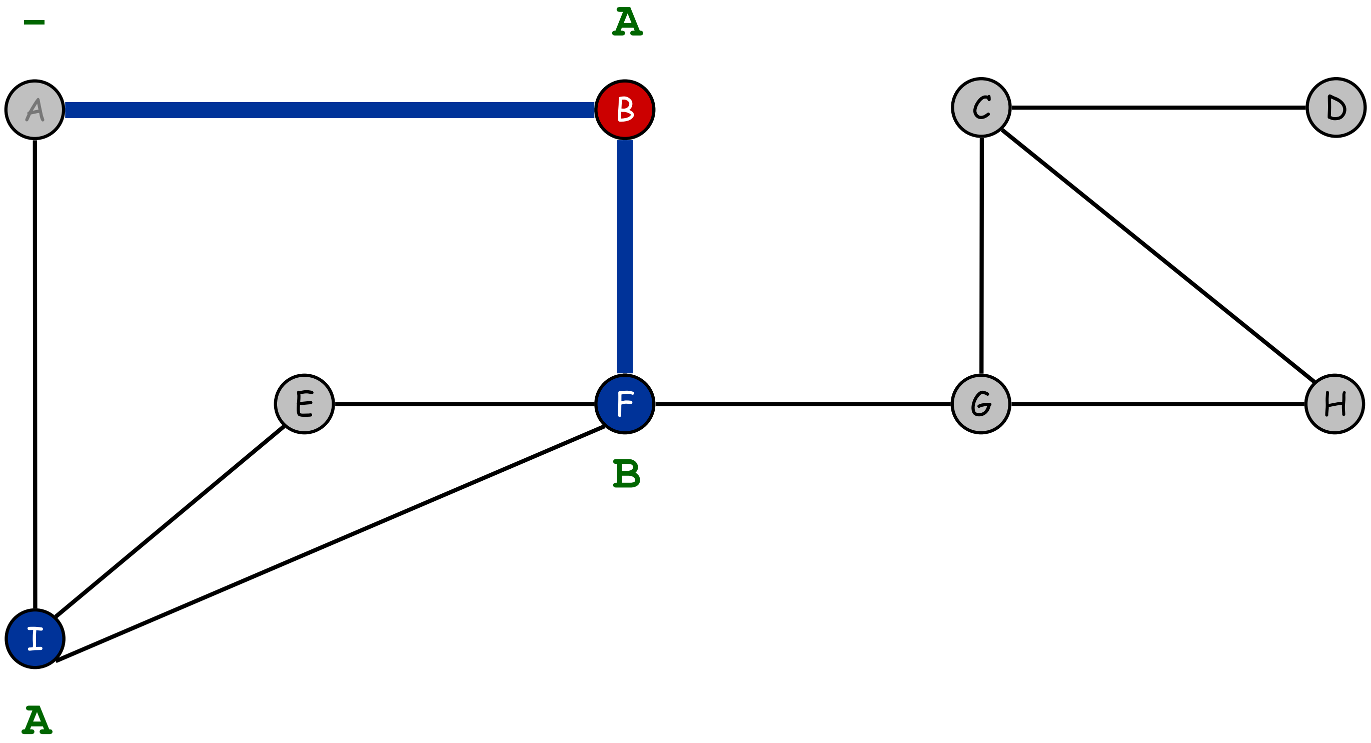
visit neighbors of B

front

I F

FIFO Queue

Breadth First Search



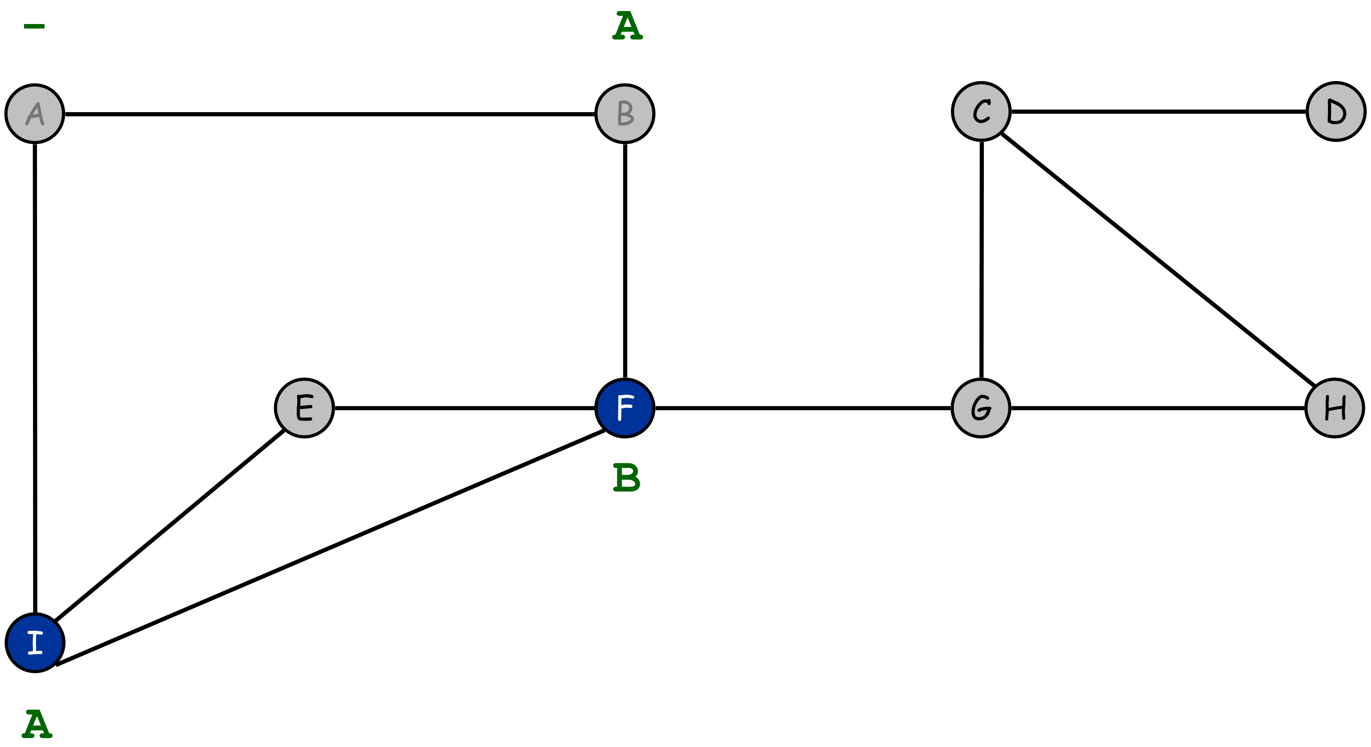
A already discovered

front

I F

FIFO Queue

Breadth First Search

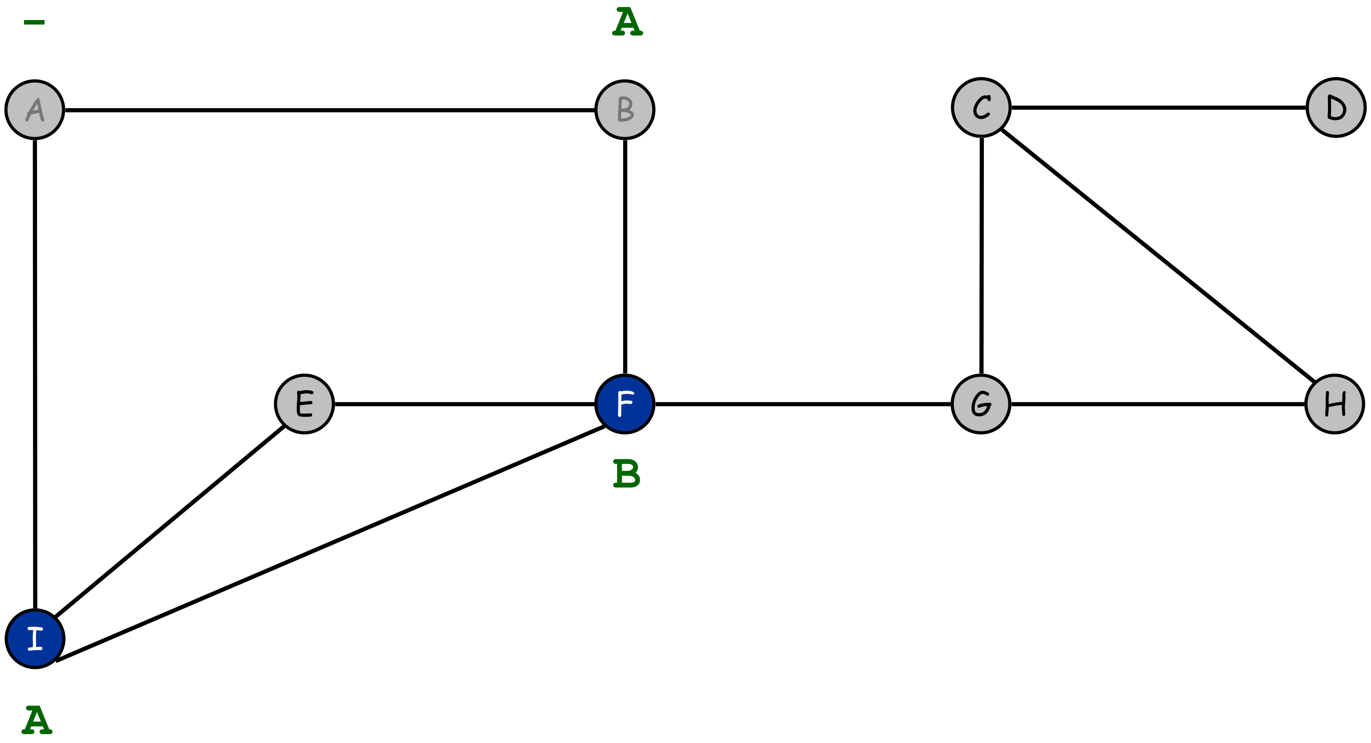


finished with B

front **I F**

FIFO Queue

Breadth First Search



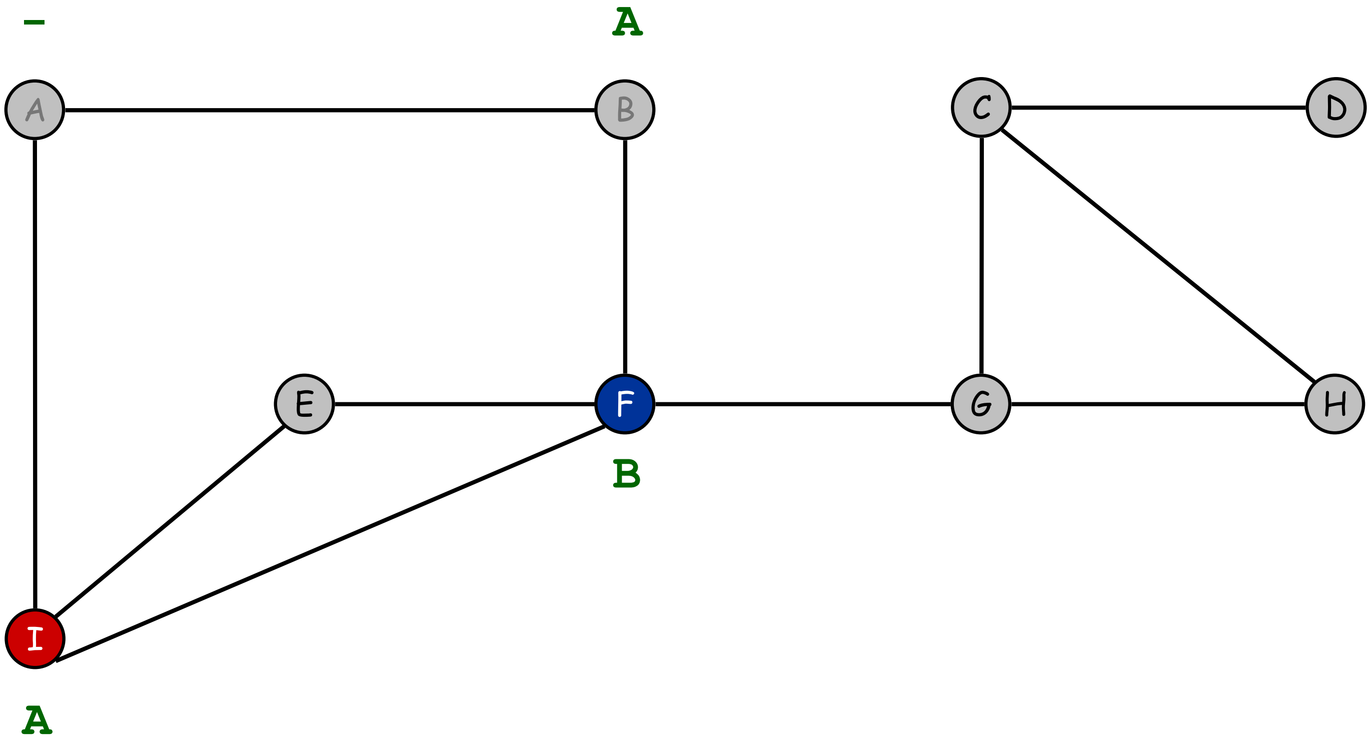
dequeue next vertex

front

I F

FIFO Queue

Breadth First Search

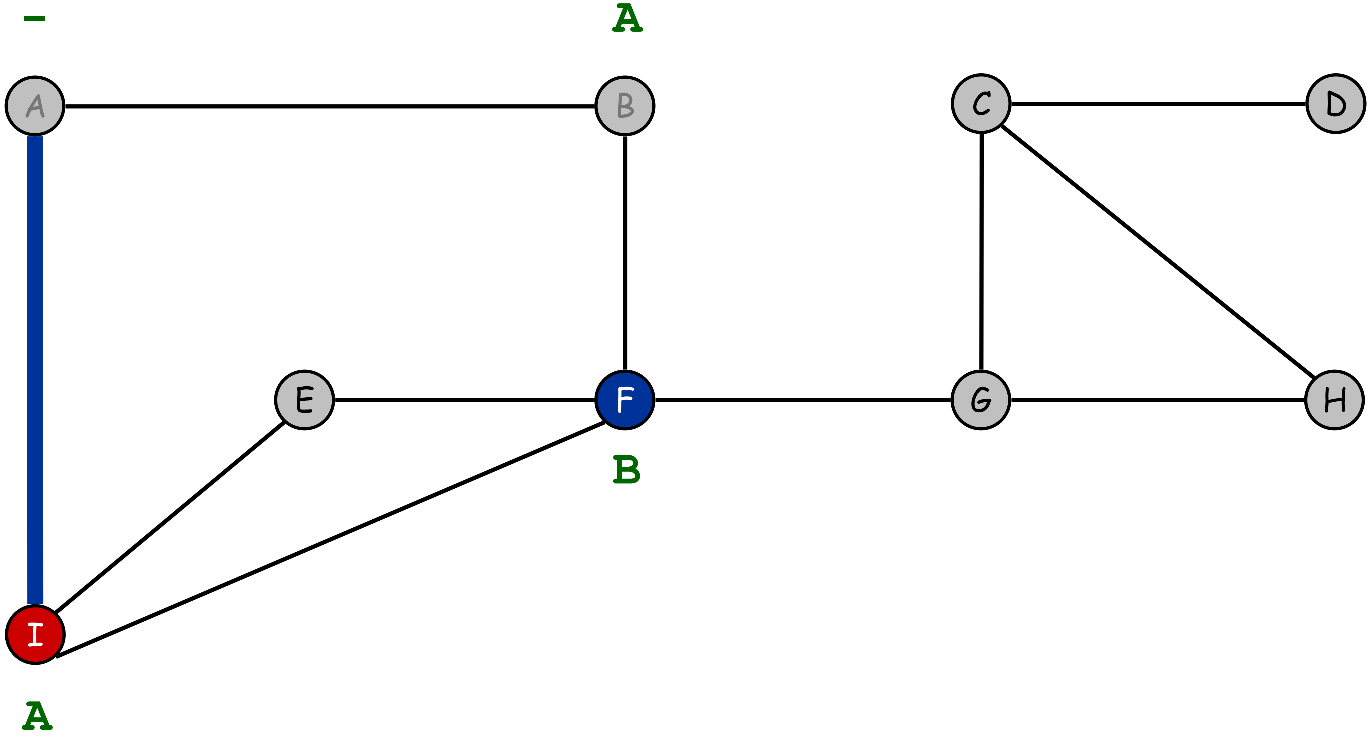


visit neighbors of I

front **F**

FIFO Queue

Breadth First Search



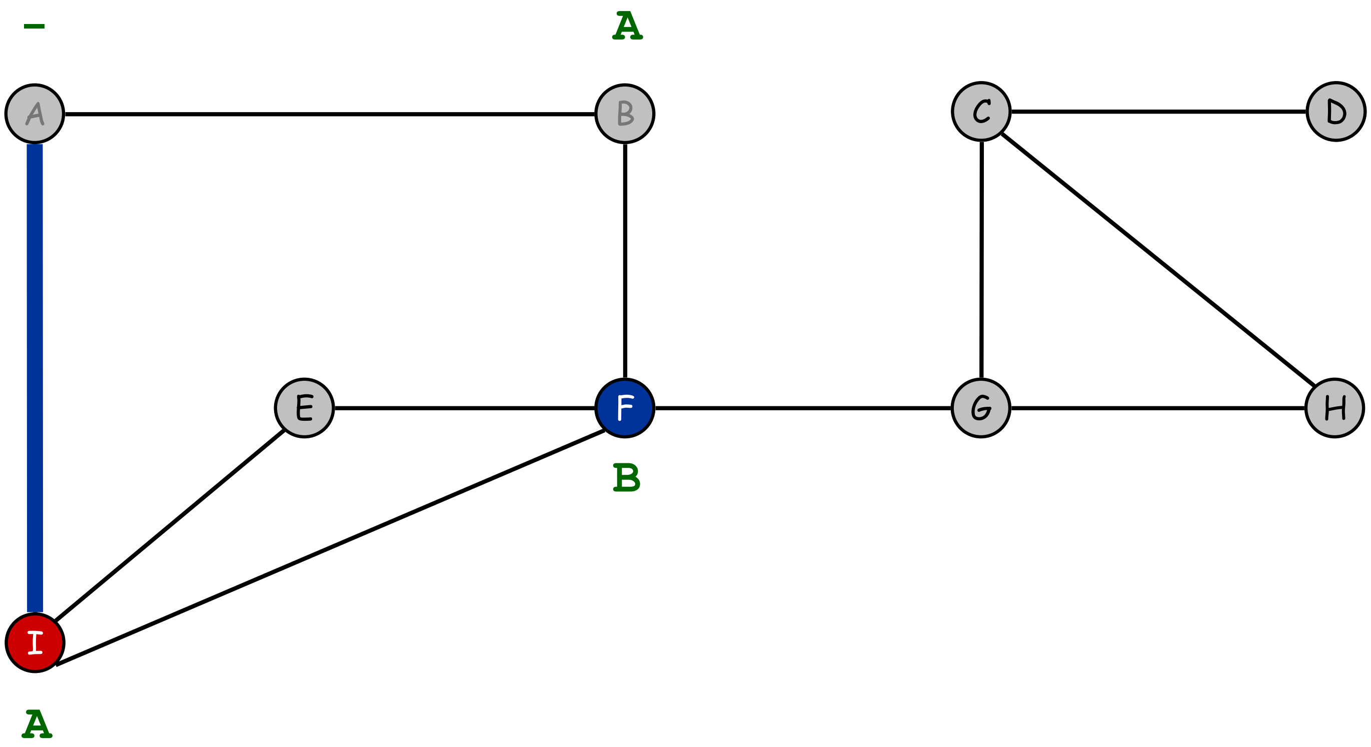
visit neighbors of I

front

F

FIFO Queue

Breadth First Search

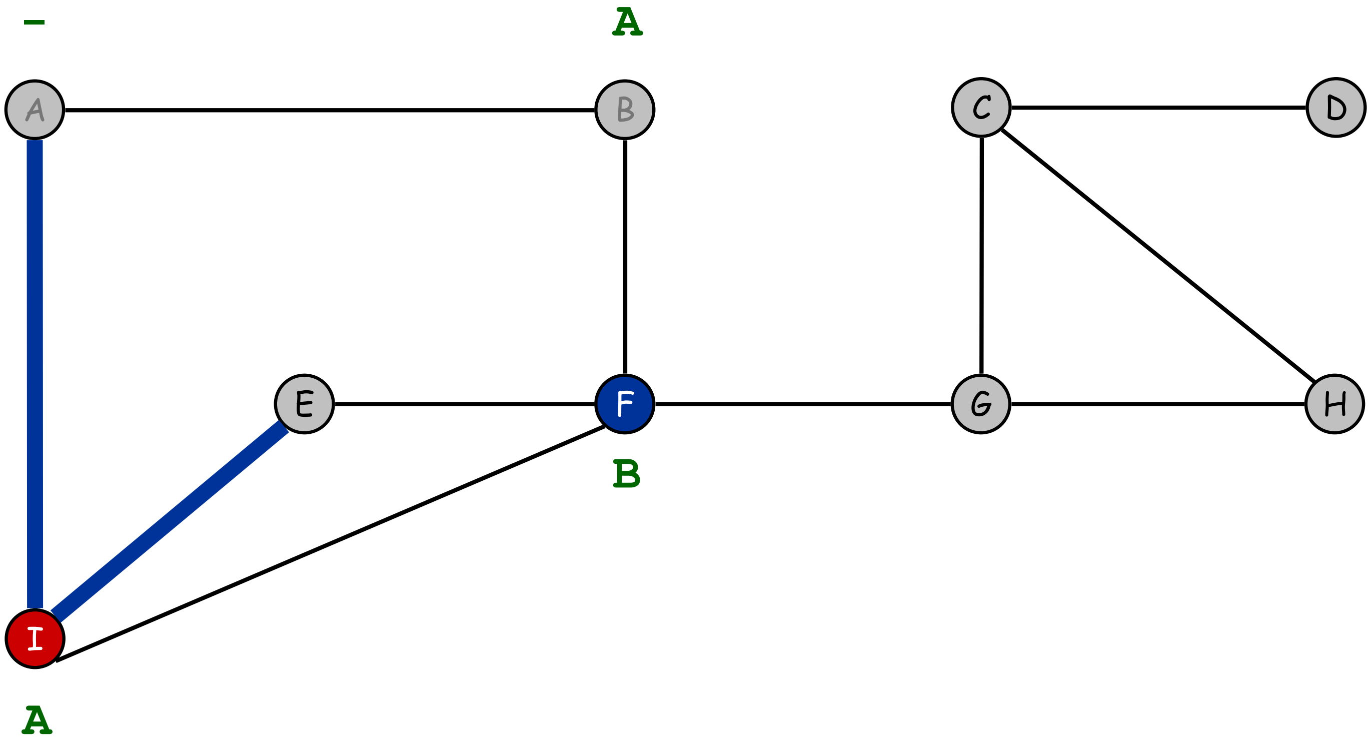


A already discovered

front F

FIFO Queue

Breadth First Search

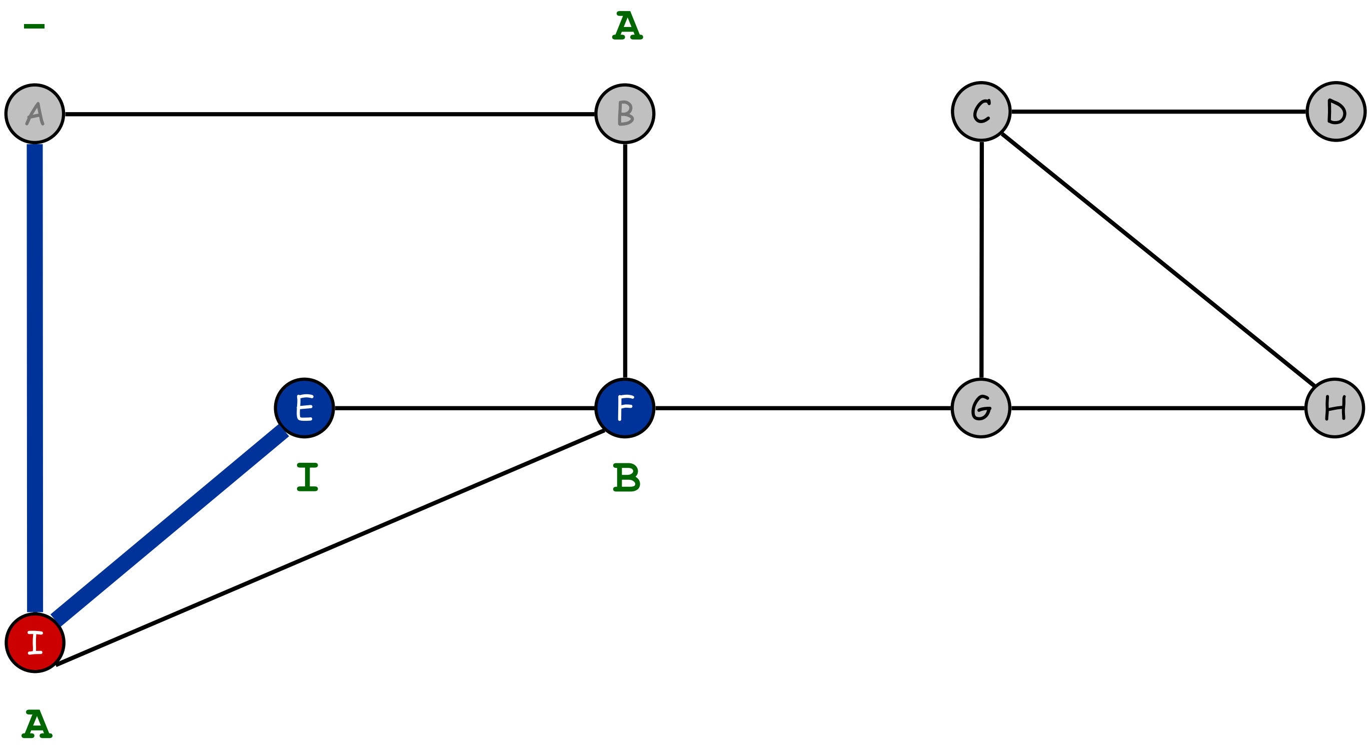


visit neighbors of I

front **F**

FIFO Queue

Breadth First Search



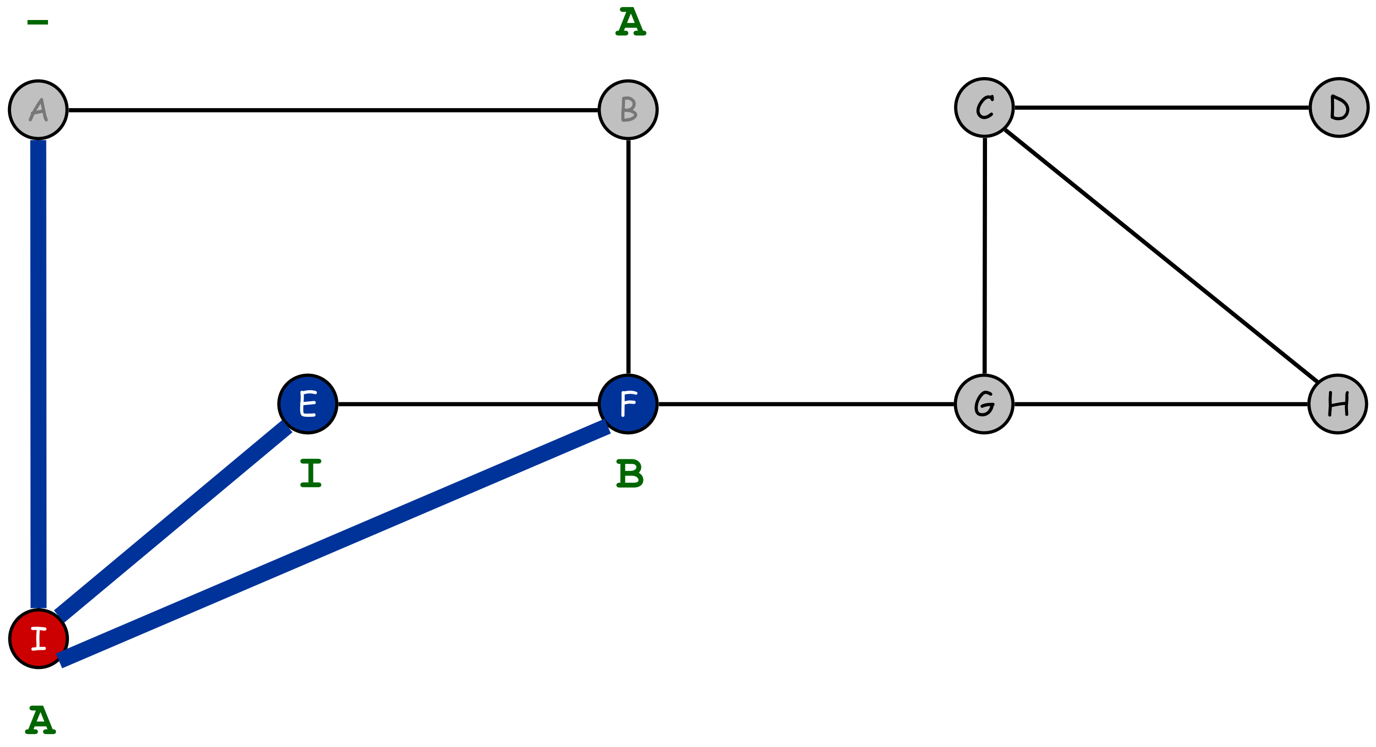
E discovered

front

F E

FIFO Queue

Breadth First Search



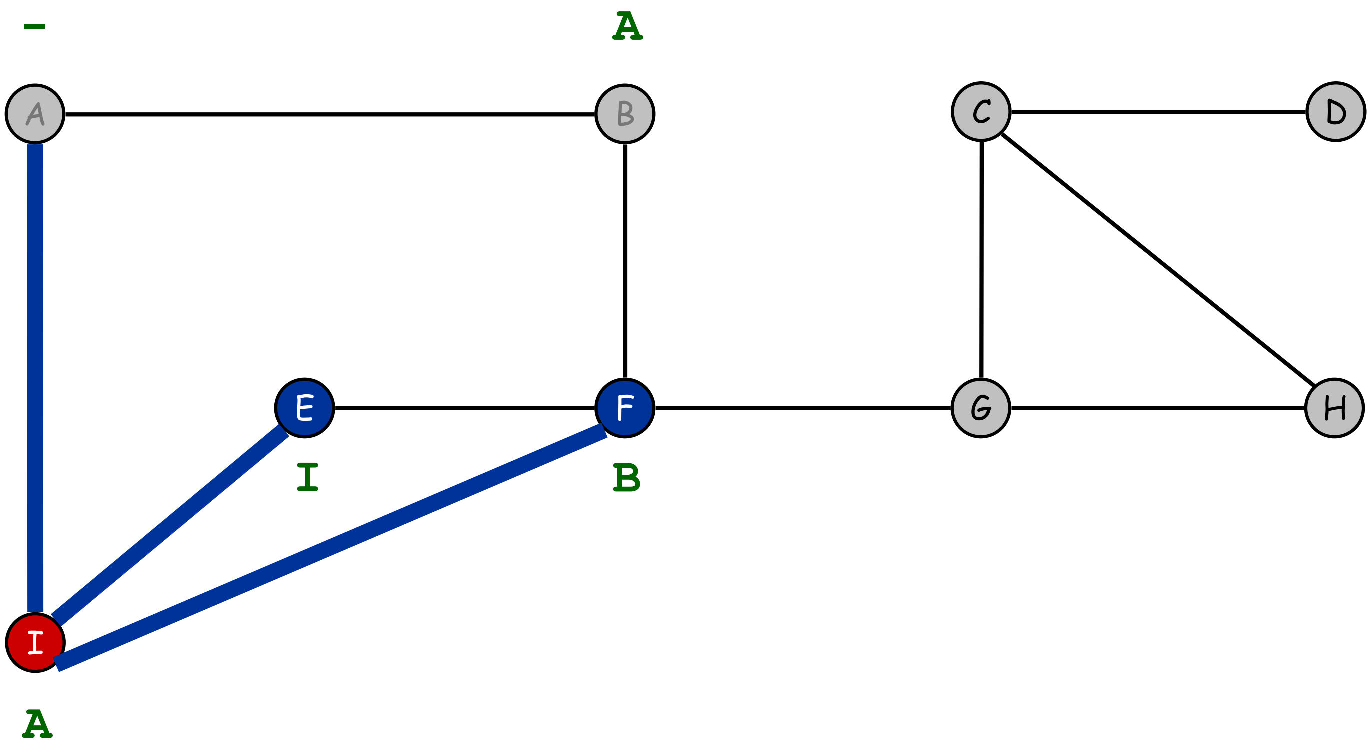
visit neighbors of I

front

F E

FIFO Queue

Breadth First Search

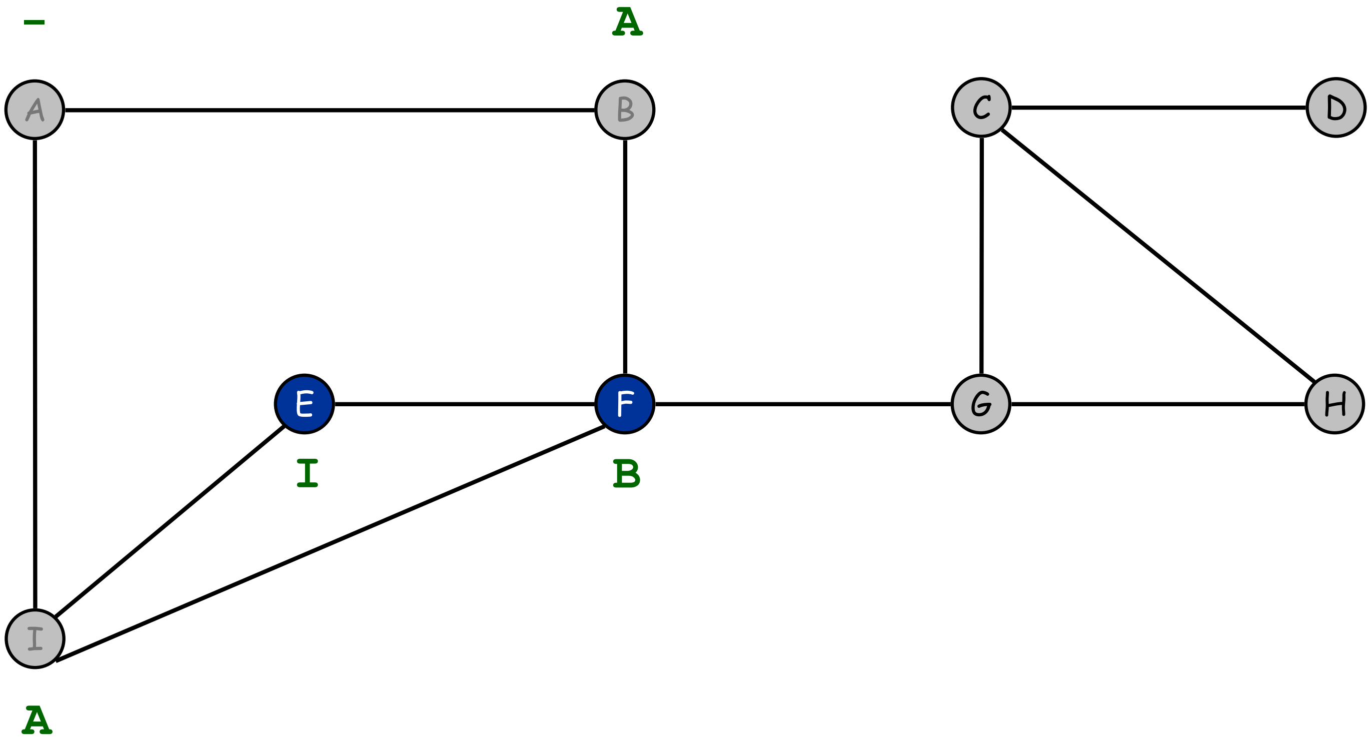


F already discovered

front **F E**

FIFO Queue

Breadth First Search

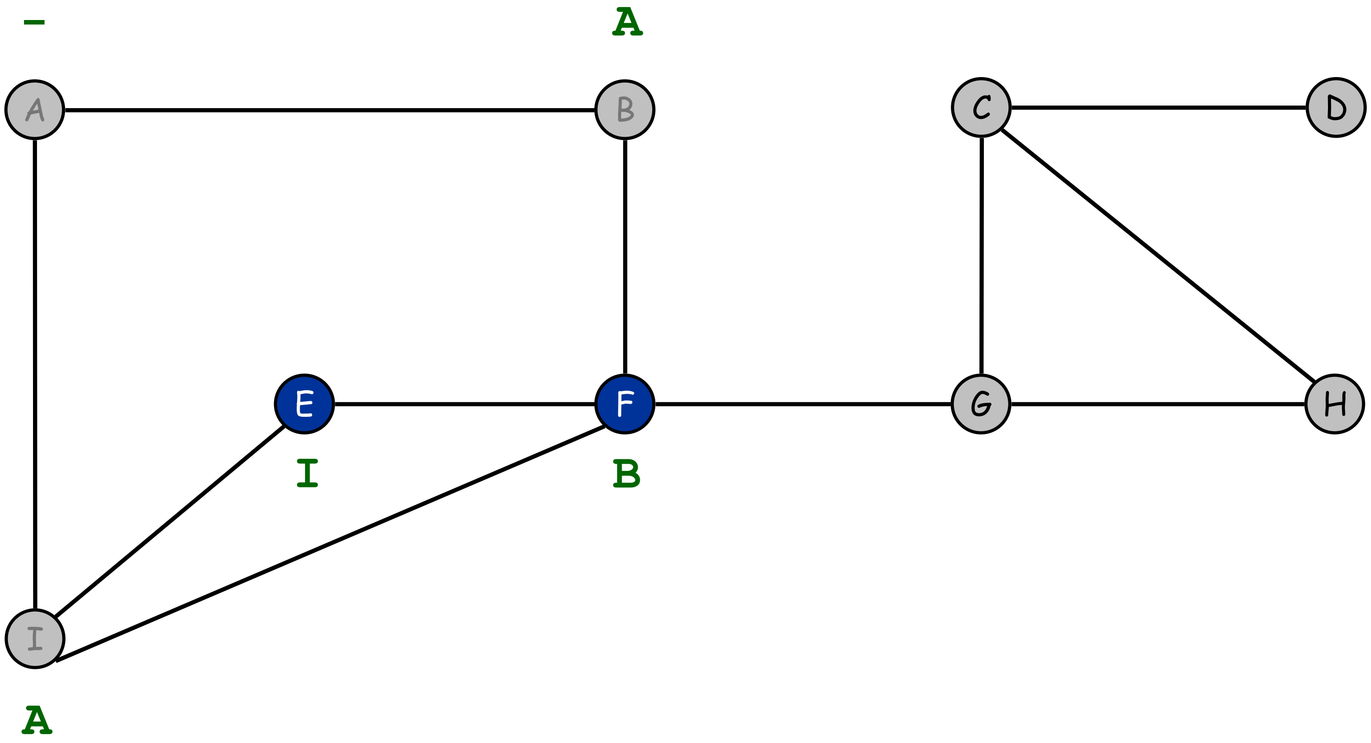


I finished

front F E

FIFO Queue

Breadth First Search

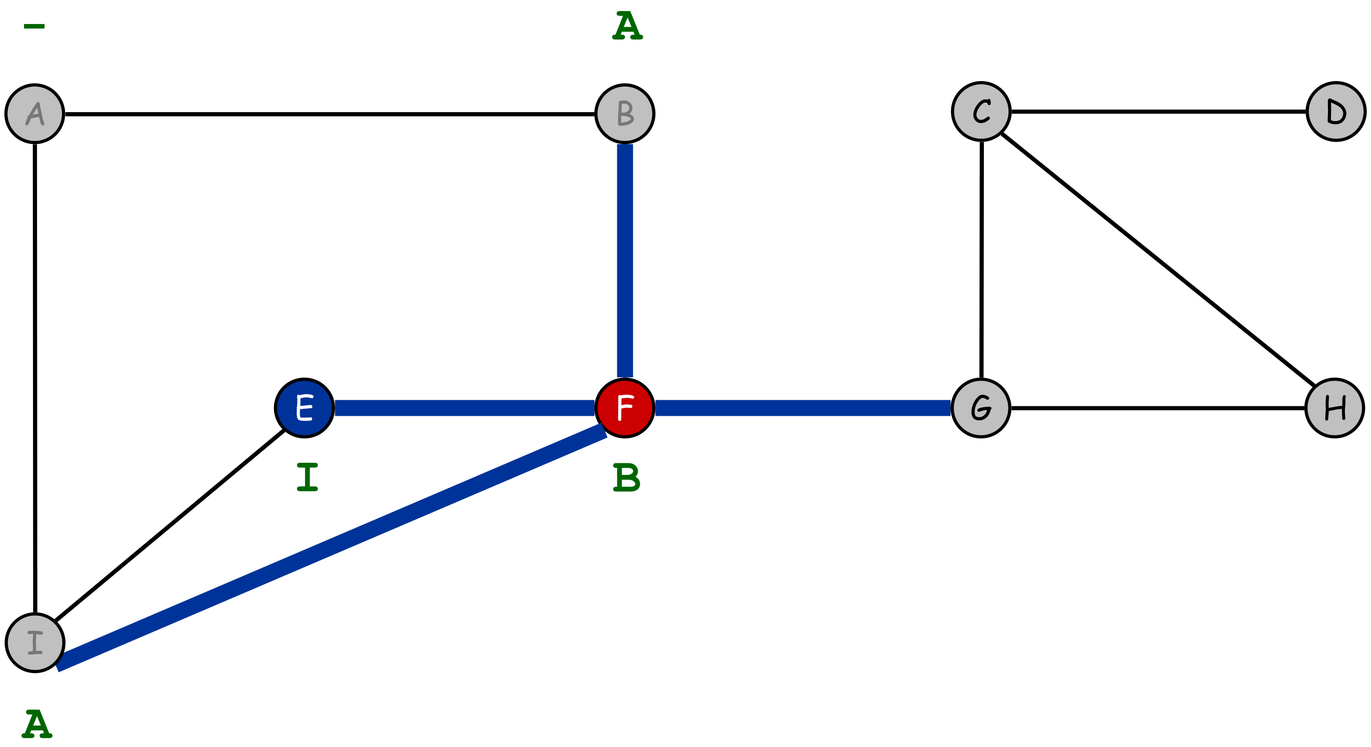


dequeue next vertex

front **F E**

FIFO Queue

Breadth First Search

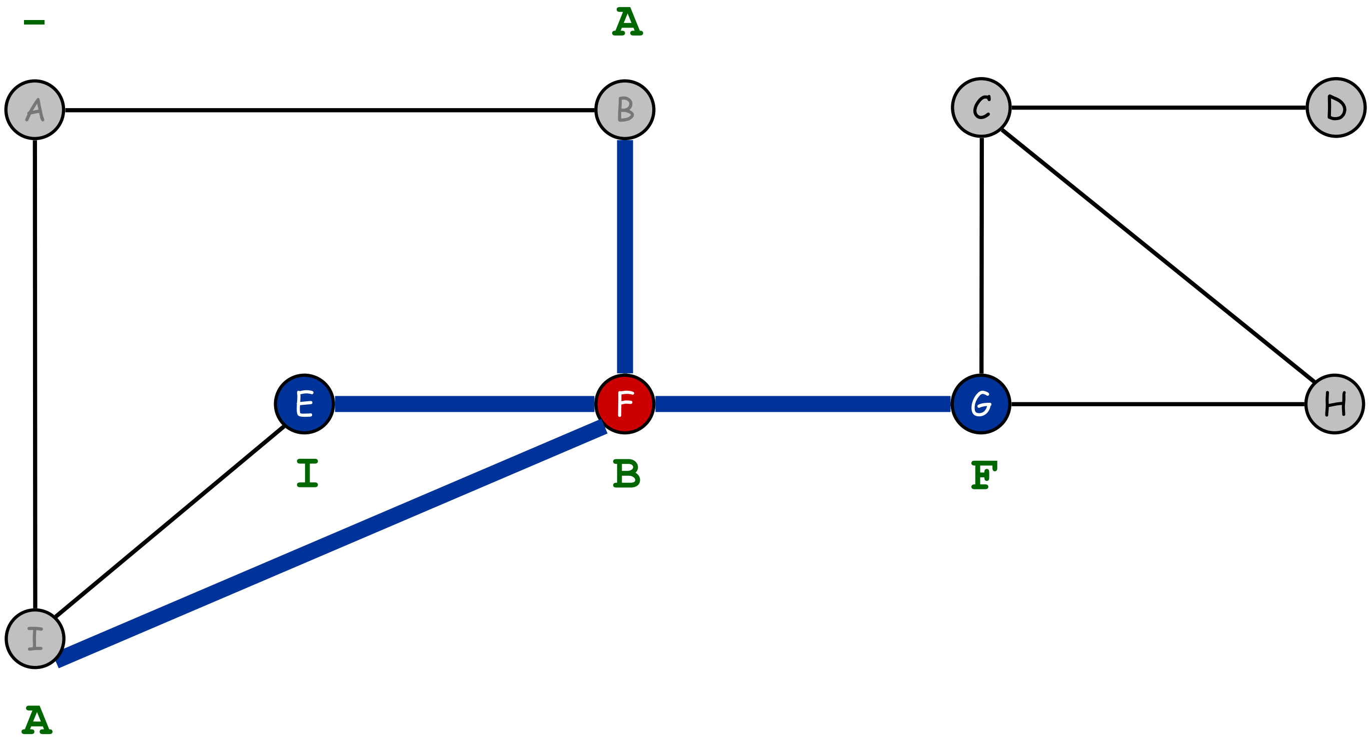


visit neighbors of F

front **E**

FIFO Queue

Breadth First Search



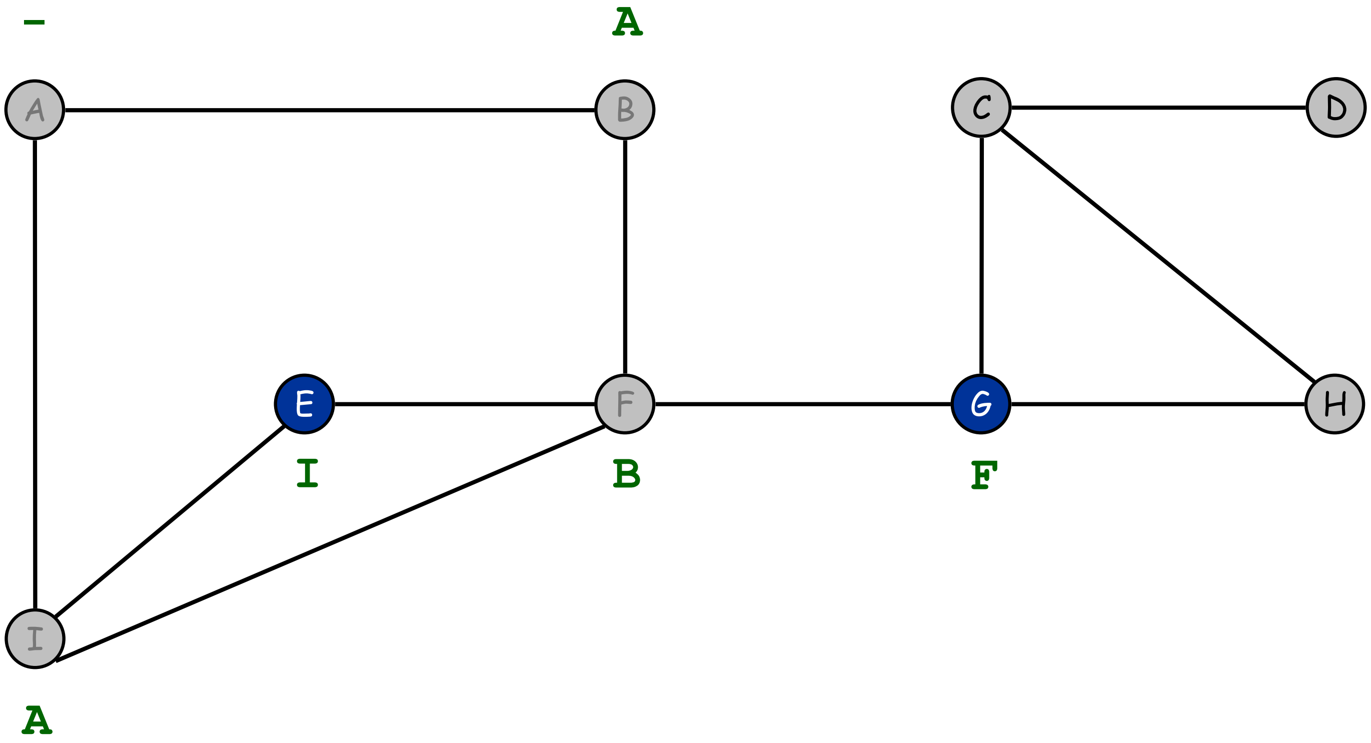
G discovered

front

E G

FIFO Queue

Breadth First Search



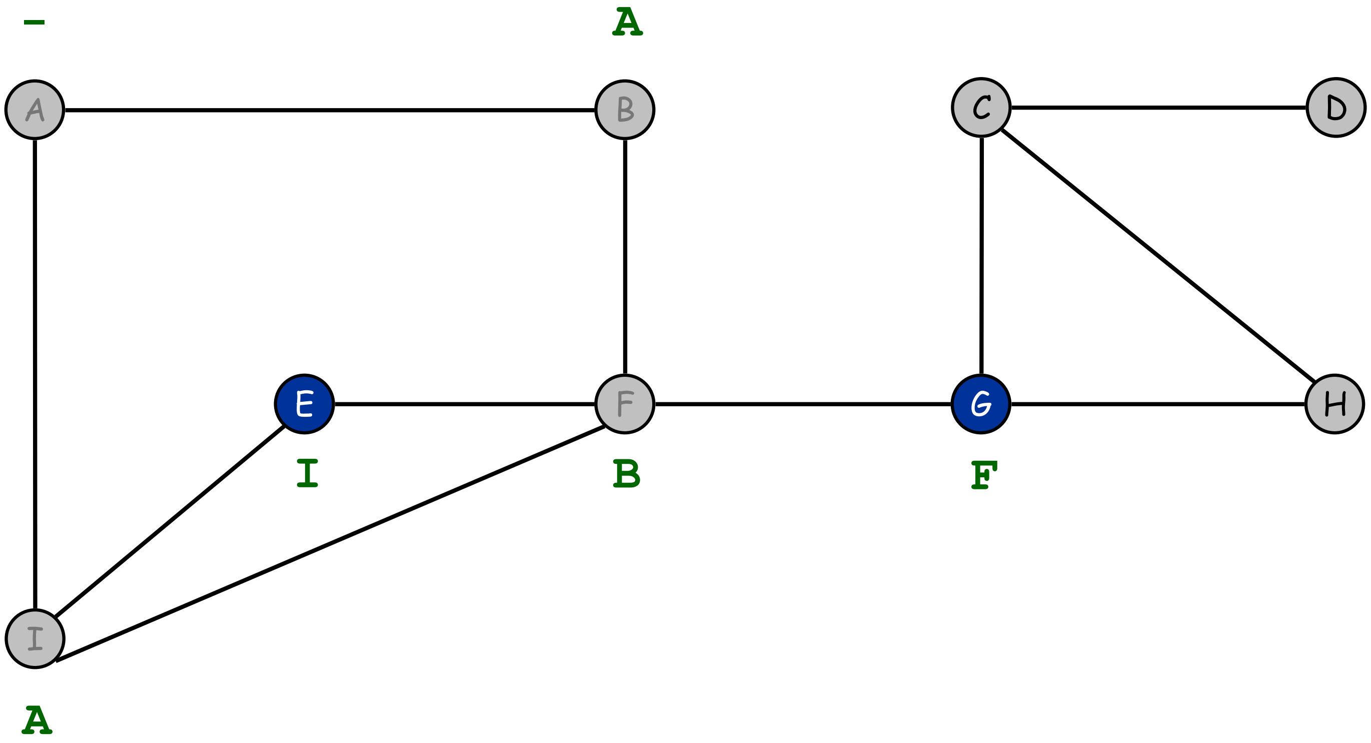
F finished

front

E G

FIFO Queue

Breadth First Search

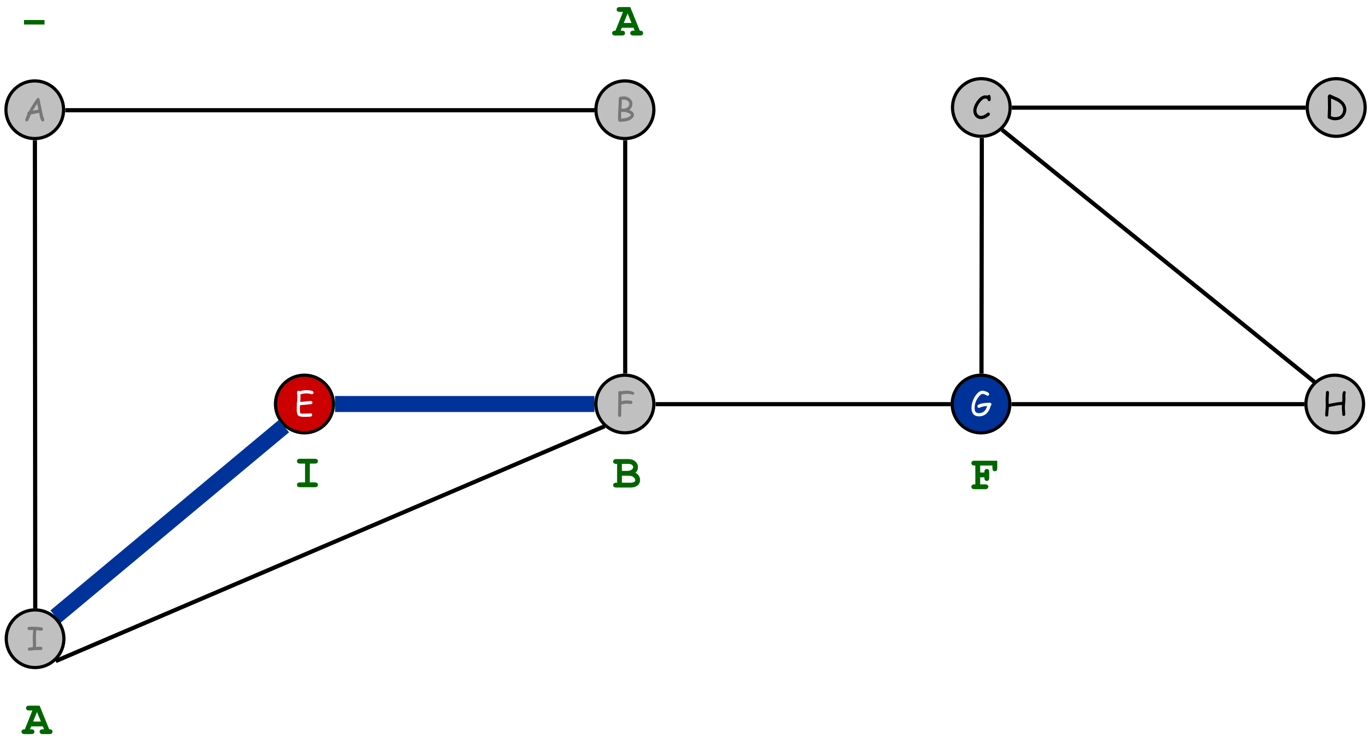


dequeue next vertex

front **E G**

FIFO Queue

Breadth First Search



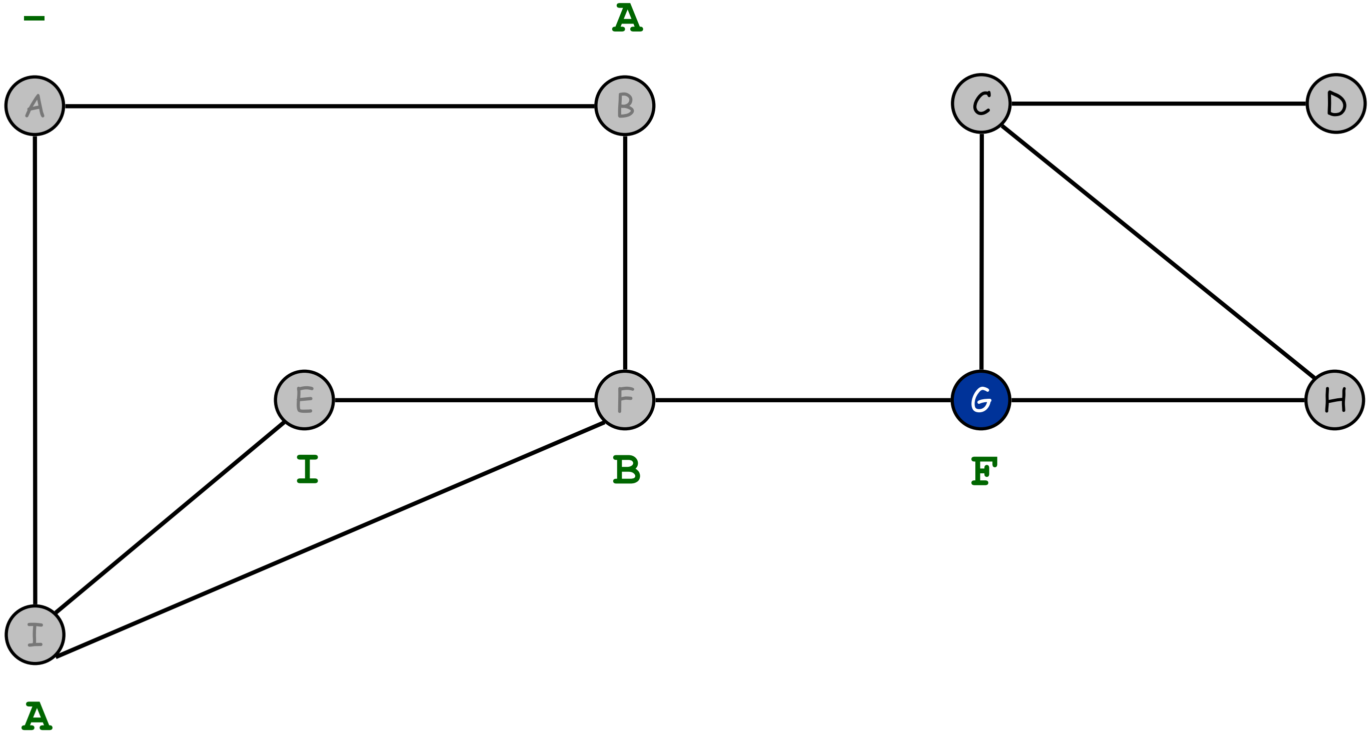
visit neighbors of E

front

G

FIFO Queue

Breadth First Search

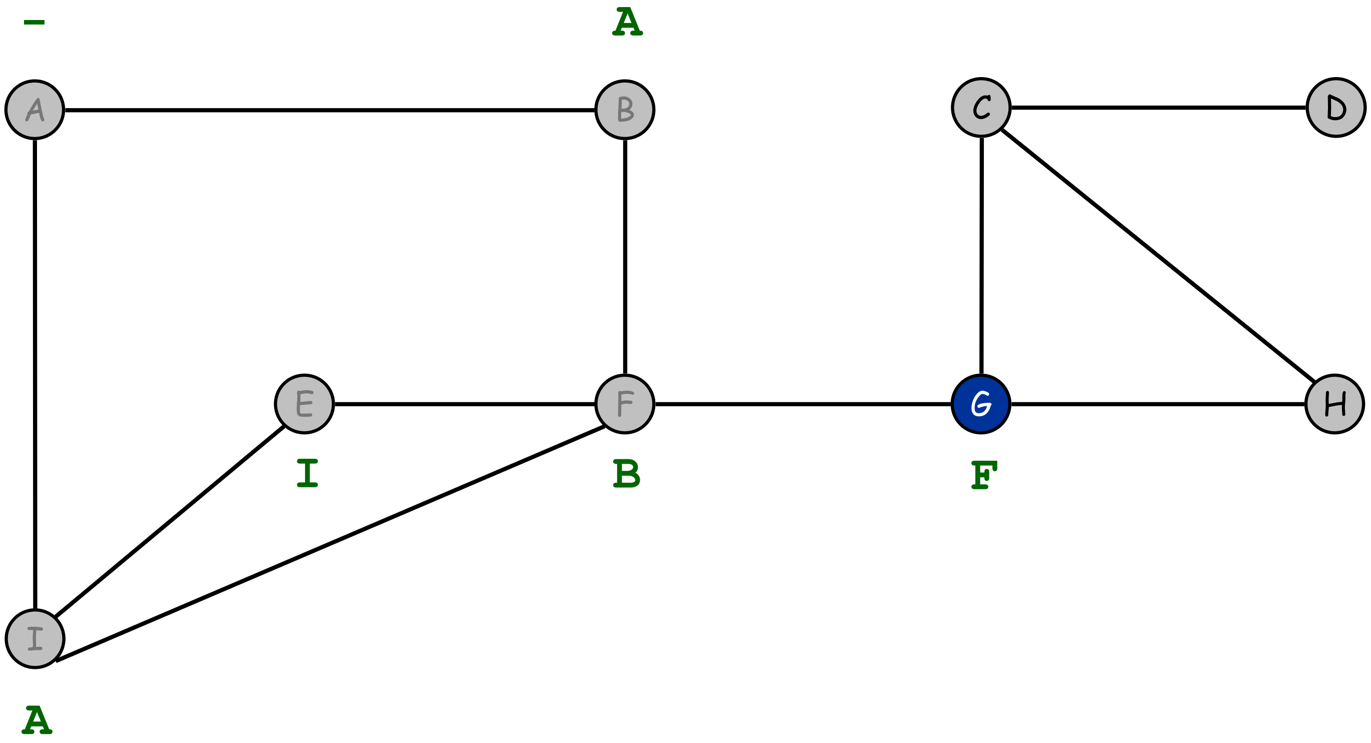


E finished

front G

FIFO Queue

Breadth First Search

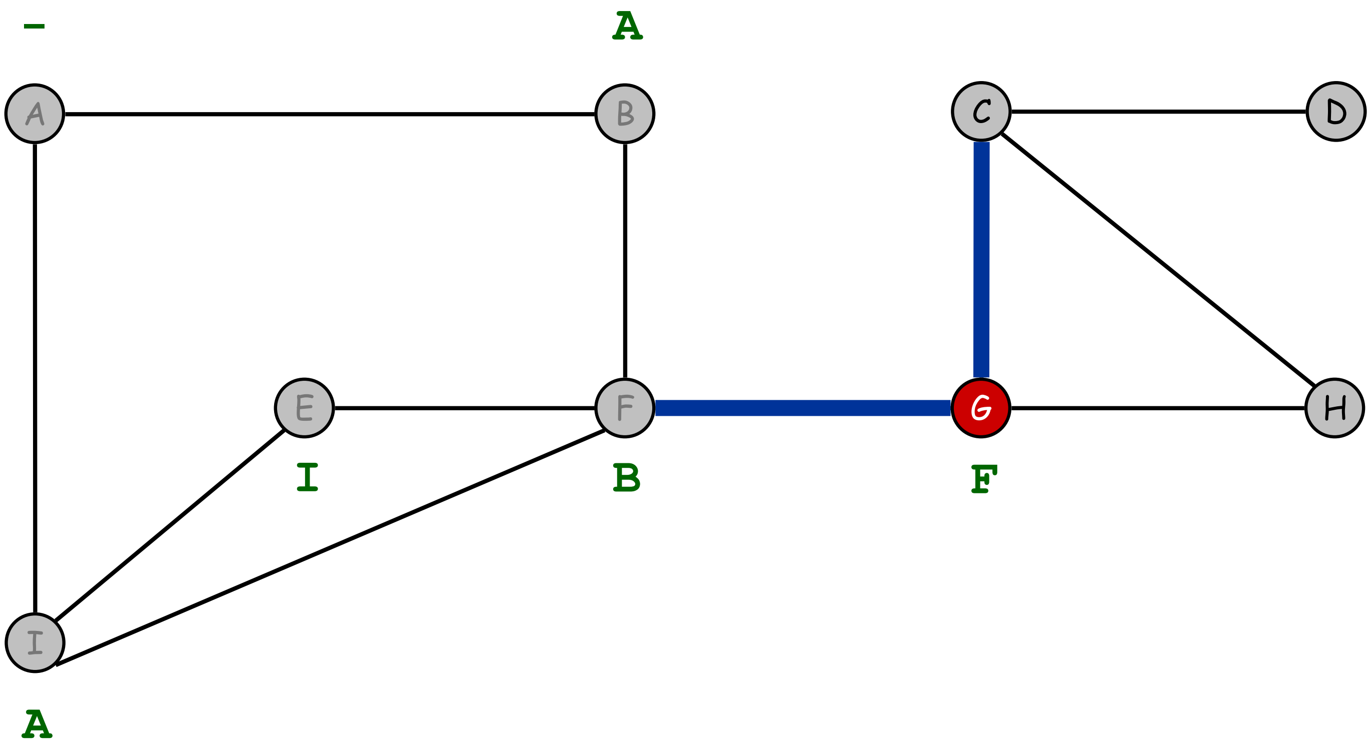


dequeue next vertex

front G

FIFO Queue

Breadth First Search

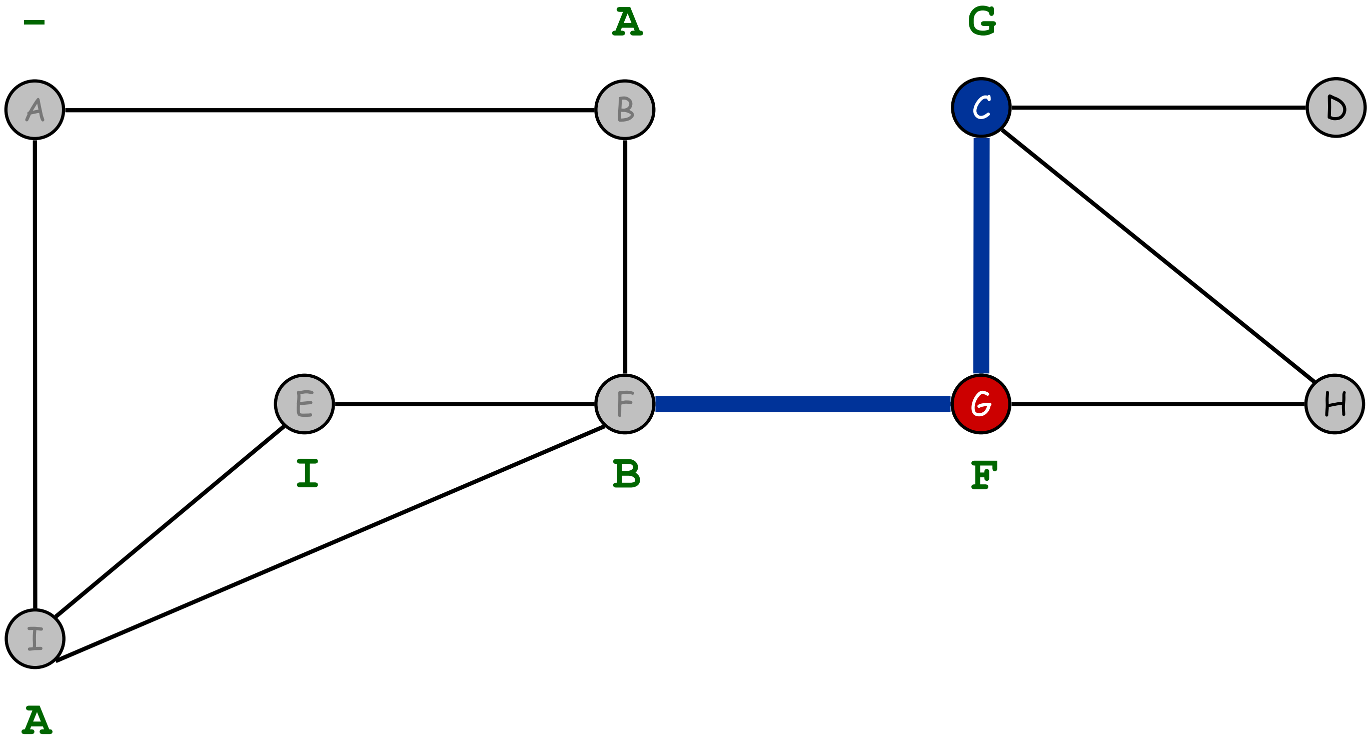


visit neighbors of G

front

FIFO Queue

Breadth First Search

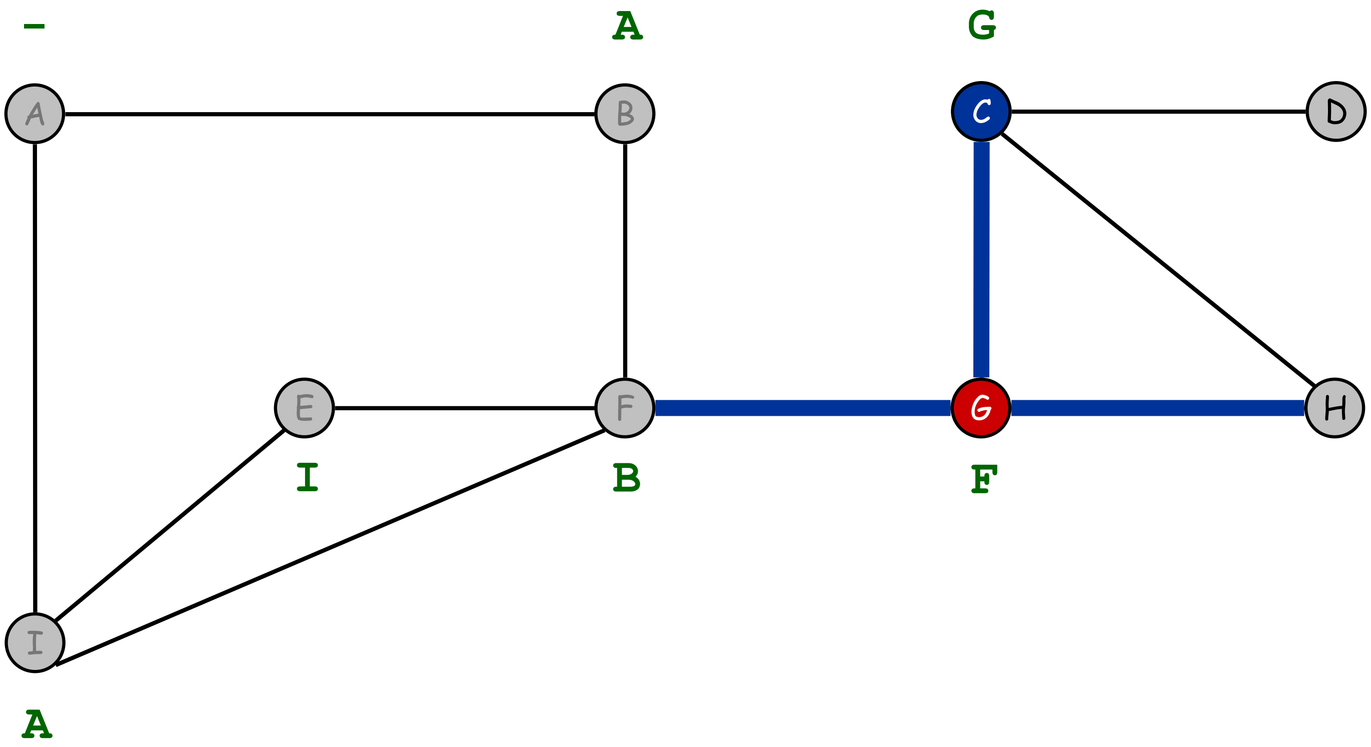


C discovered

front C

FIFO Queue

Breadth First Search



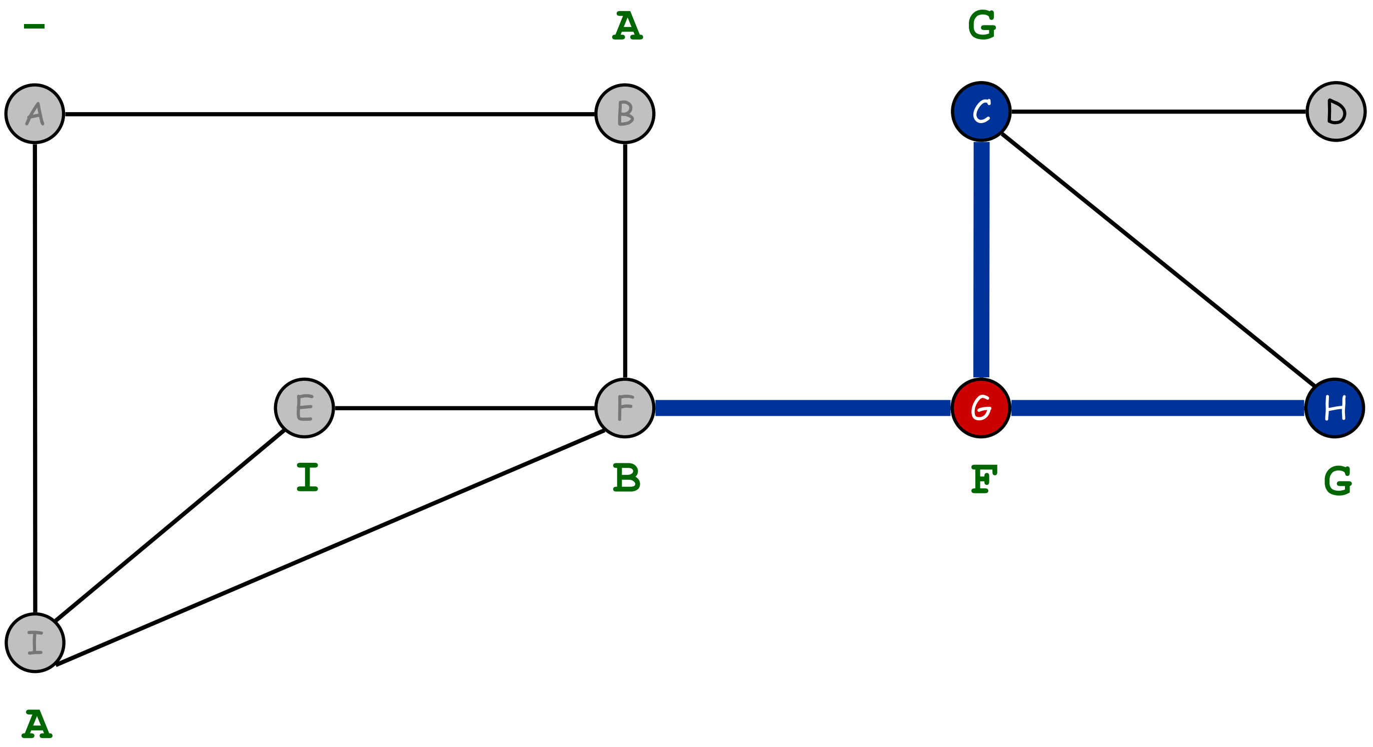
visit neighbors of G

front

C

FIFO Queue

Breadth First Search



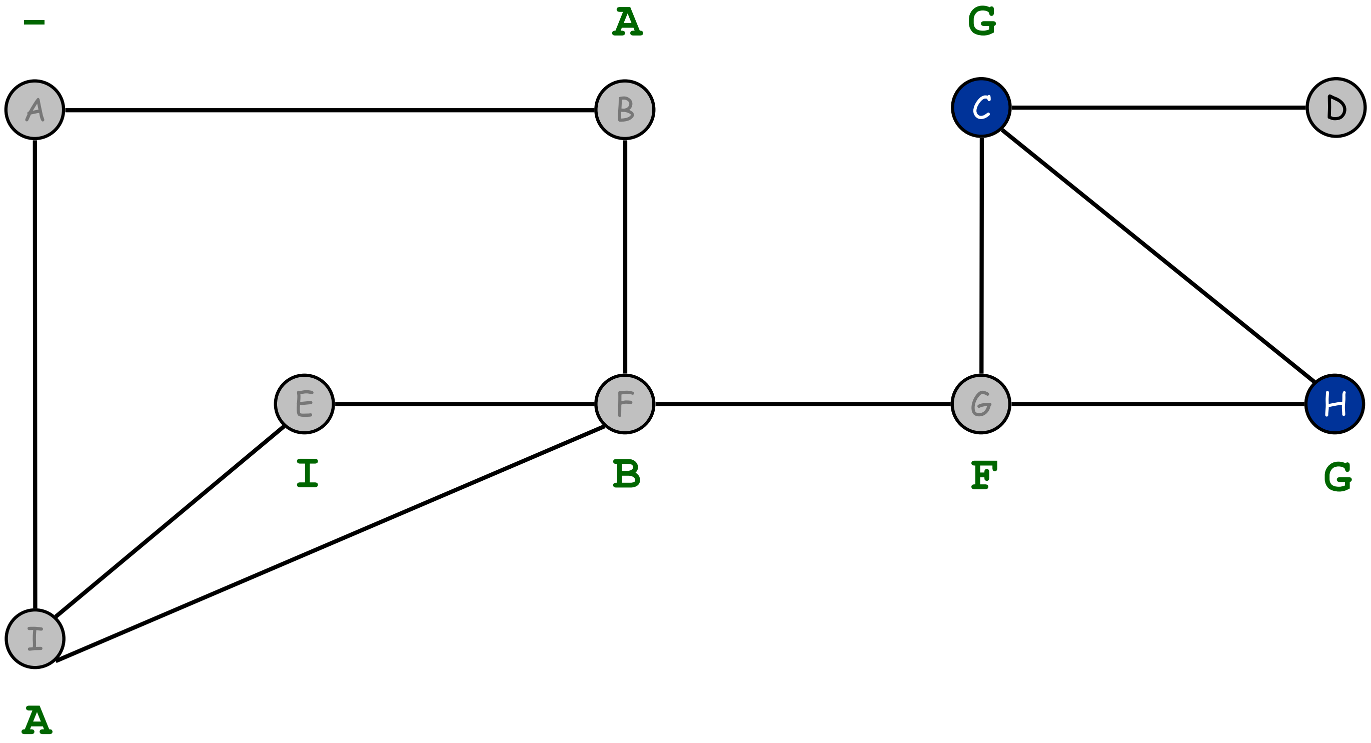
H discovered

front

C H

FIFO Queue

Breadth First Search



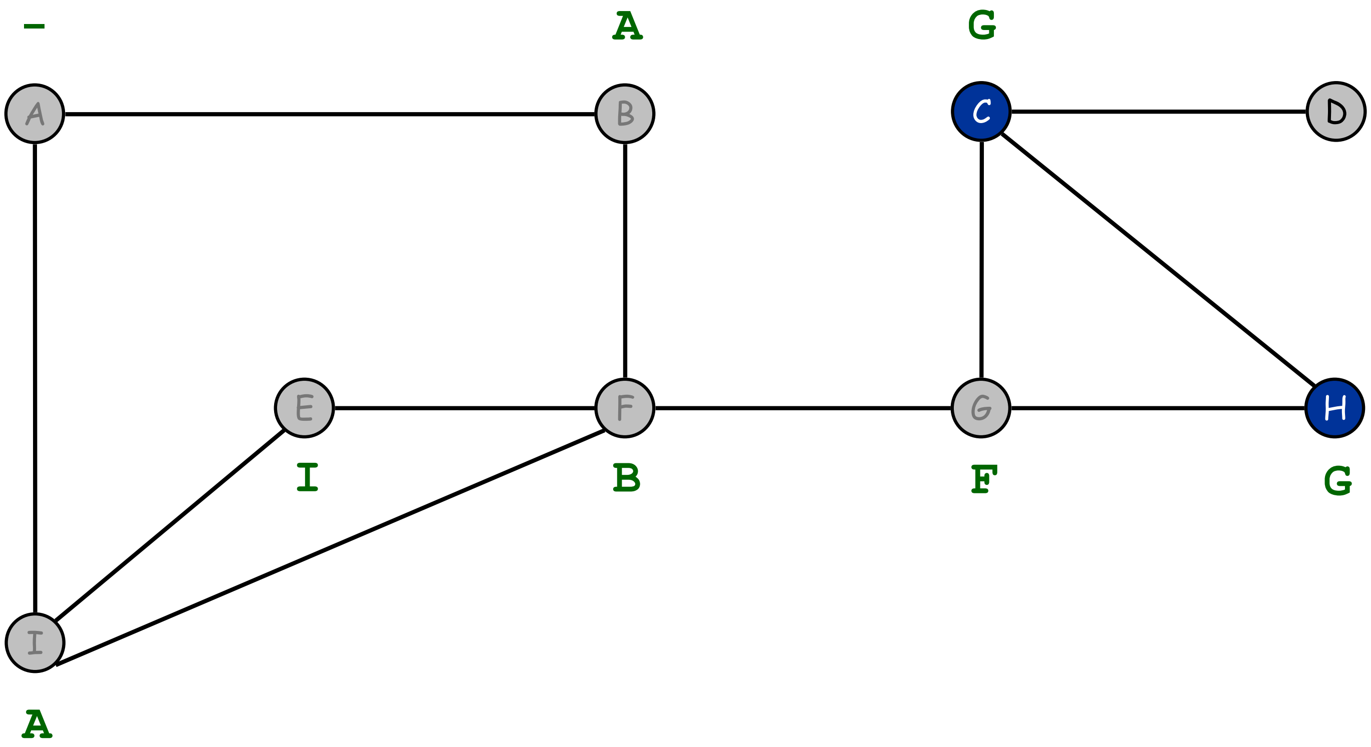
G finished

front

C	H

FIFO Queue

Breadth First Search



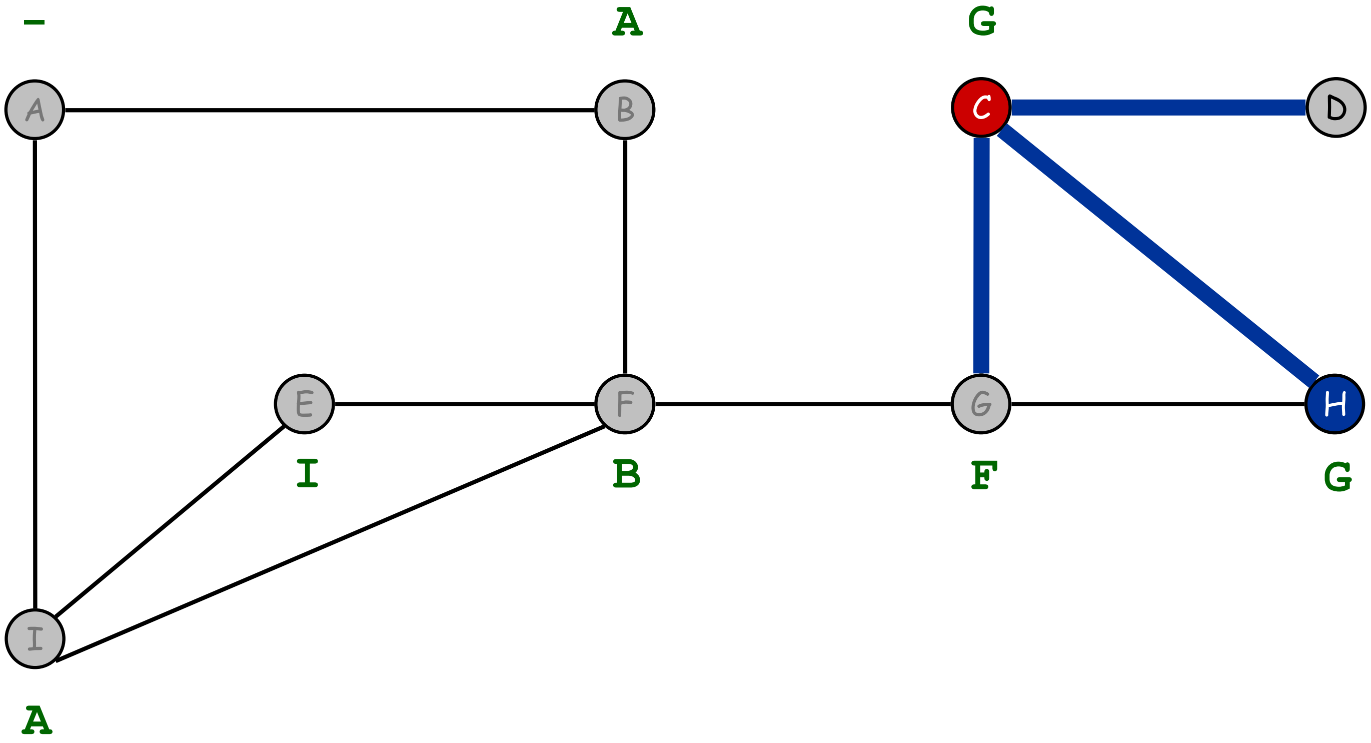
dequeue next vertex

front

C H

FIFO Queue

Breadth First Search

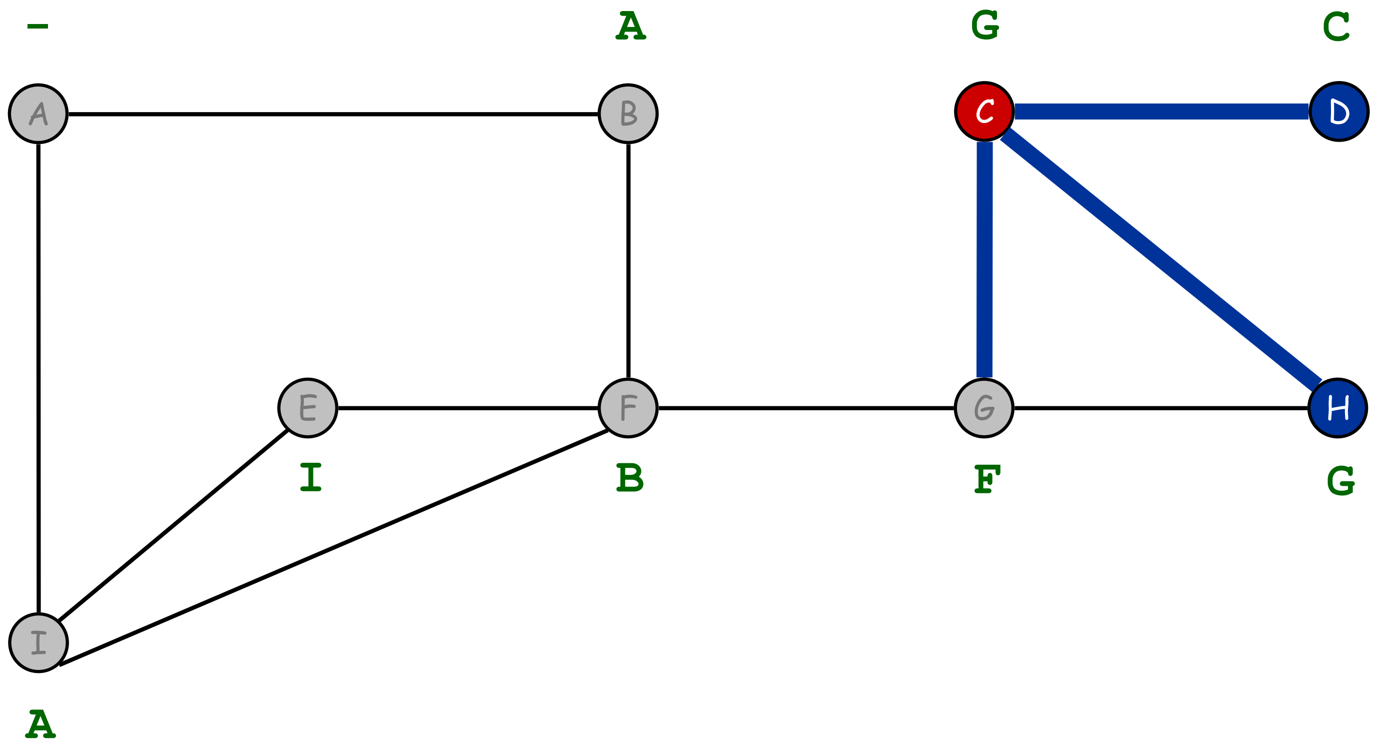


visit neighbors of C

front H

FIFO Queue

Breadth First Search

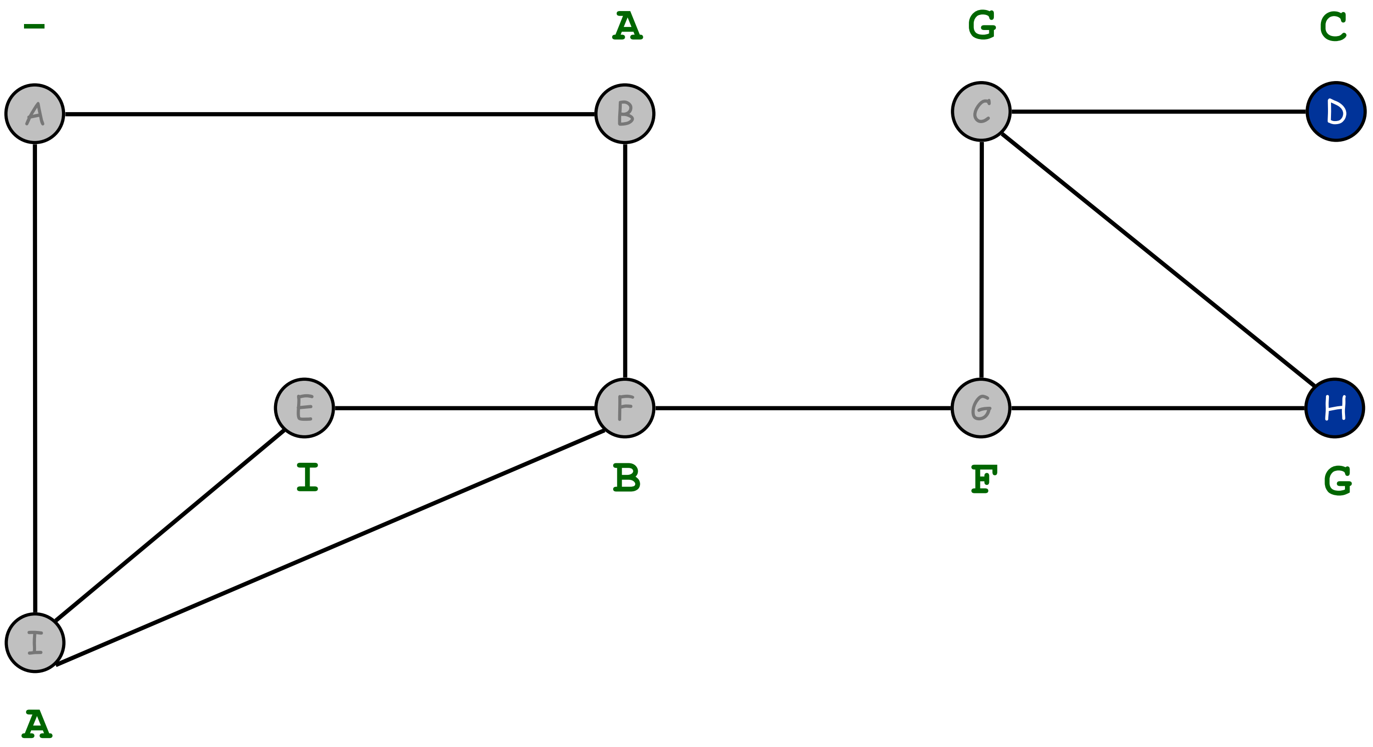


D discovered

front H D

FIFO Queue

Breadth First Search

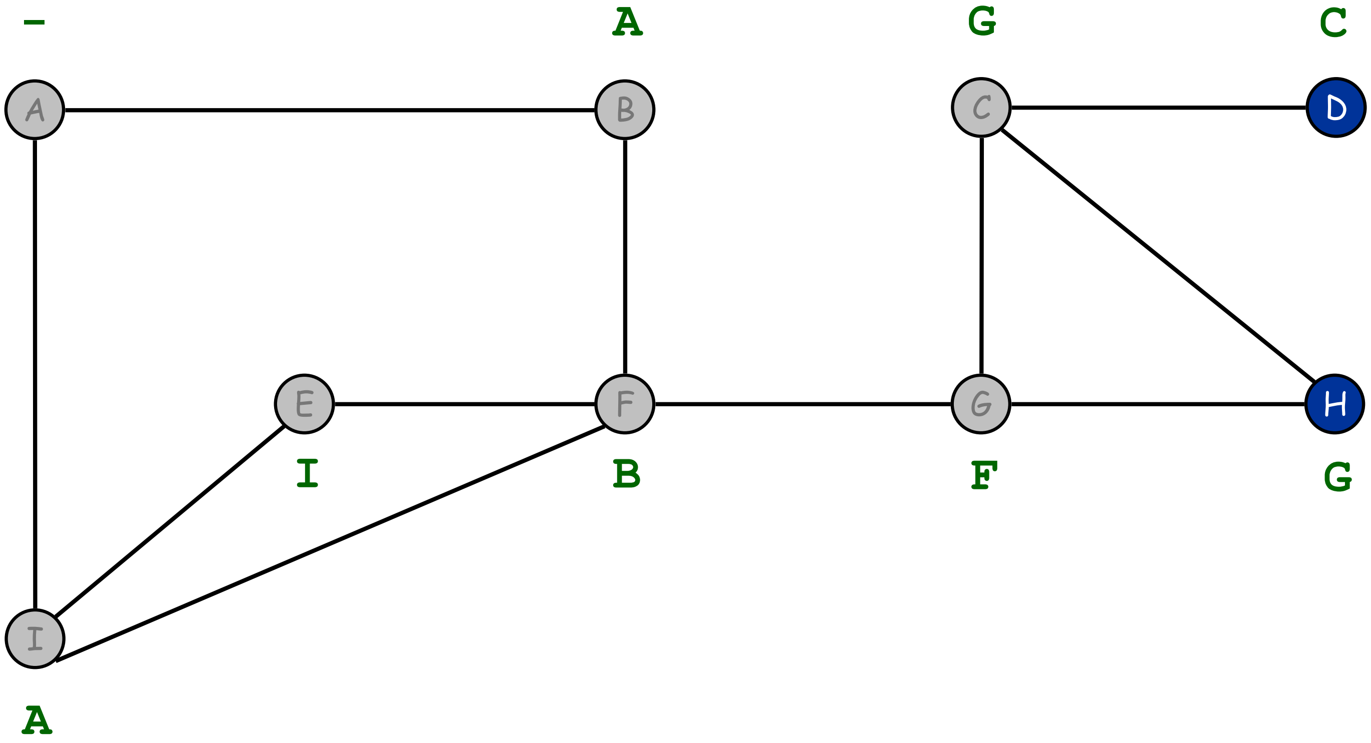


C finished

front **H D**

FIFO Queue

Breadth First Search

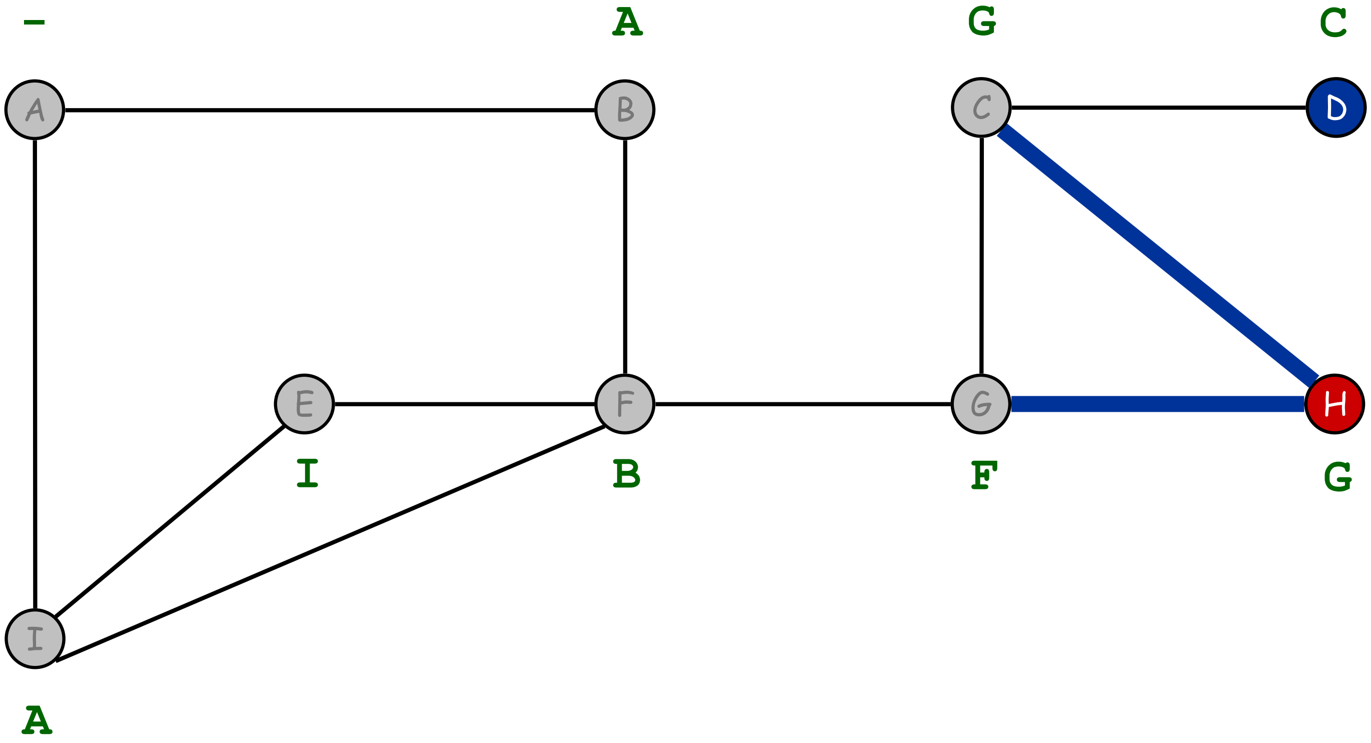


get next vertex

front **H D**

FIFO Queue

Breadth First Search



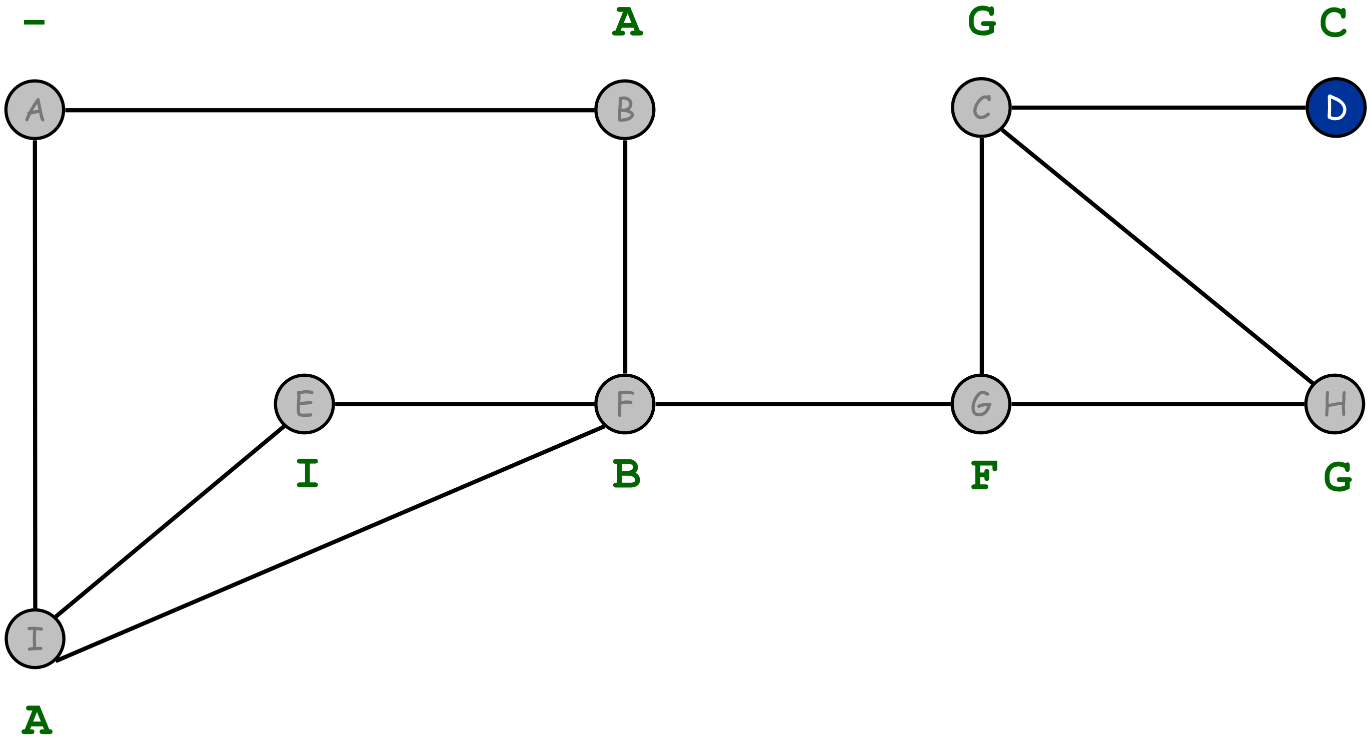
visit neighbors of H

front

D

FIFO Queue

Breadth First Search

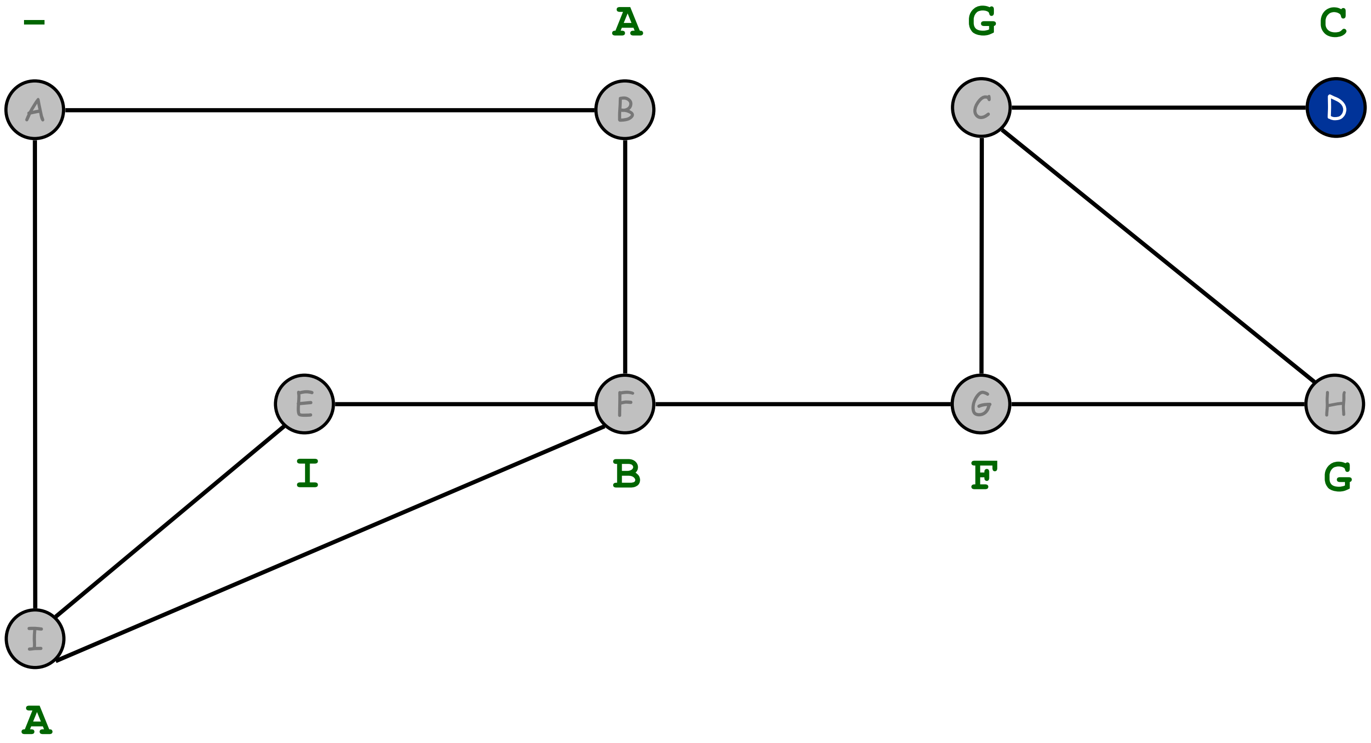


finished H

front D

FIFO Queue

Breadth First Search



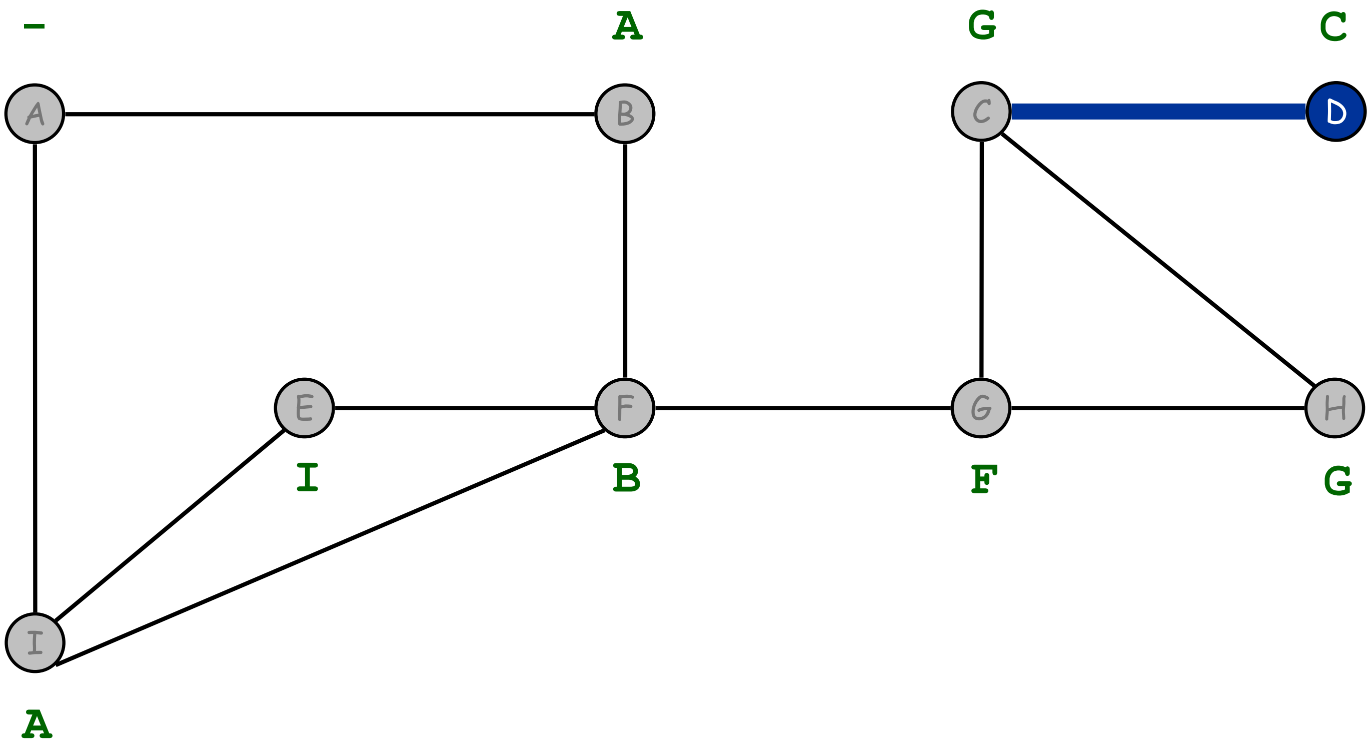
dequeue next vertex

front

D

FIFO Queue

Breadth First Search

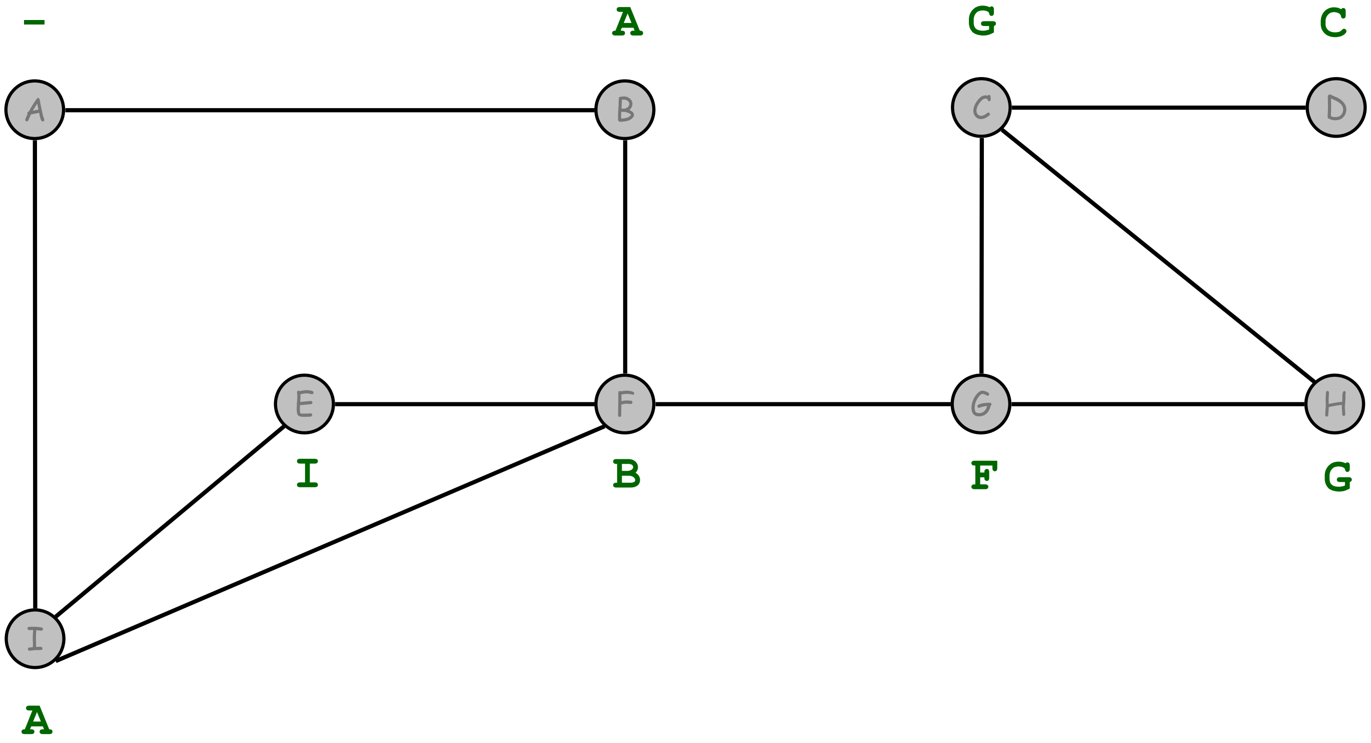


visit neighbors of D

front

FIFO Queue

Breadth First Search

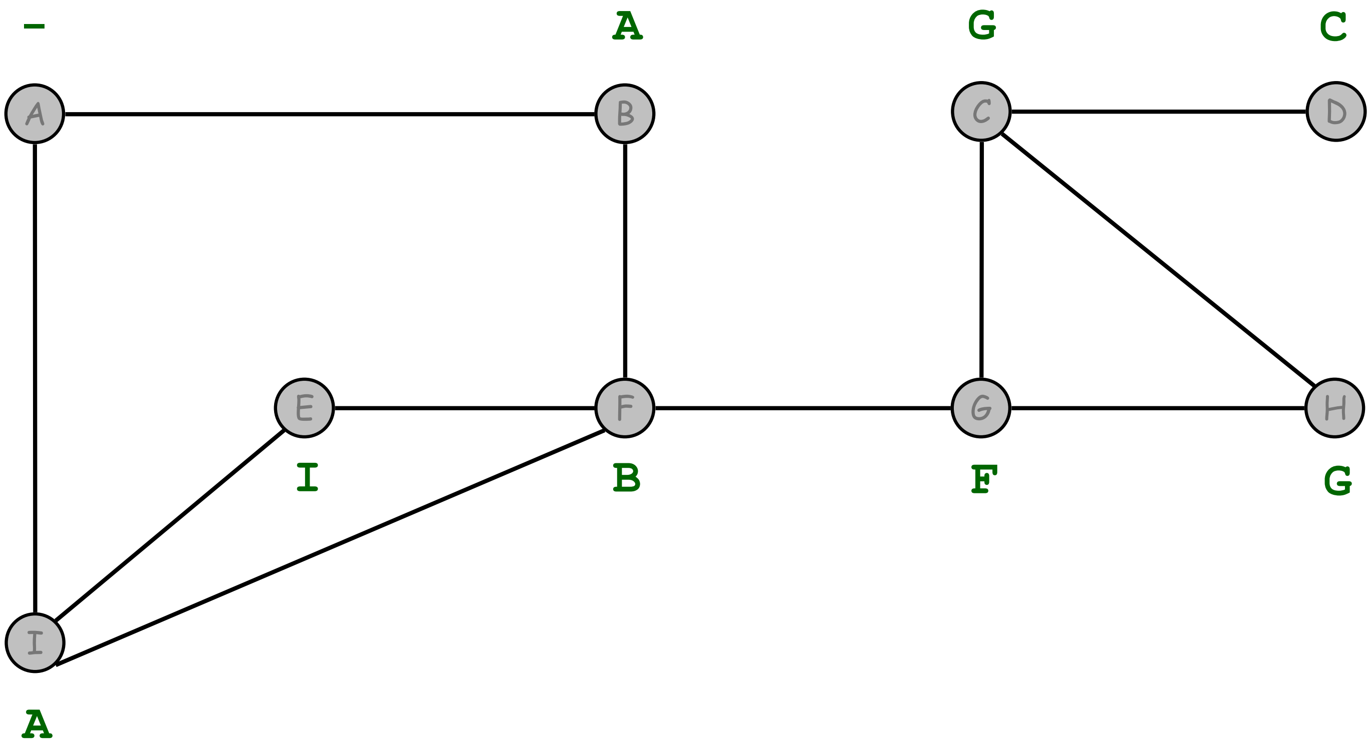


D finished

front

FIFO Queue

Breadth First Search

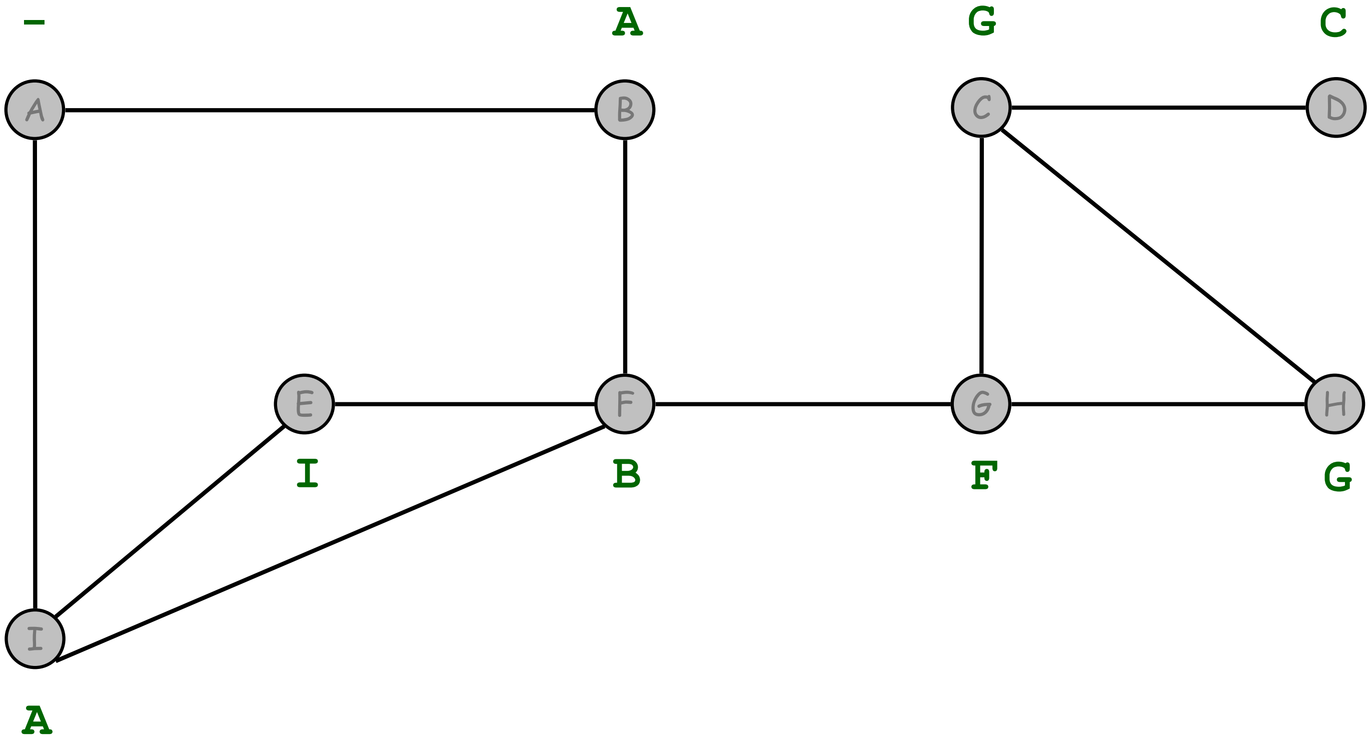


dequeue next vertex

front

FIFO Queue

Breadth First Search



STOP

front

FIFO Queue

Algorithm Performance

Completeness: Is the algorithm guaranteed to find a solution when there is one

Optimality: Does the strategy find the optimal solution

Time complexity: How long does it take to find a solution

Space complexity: How much memory is needed to perform the search

Algorithm Performance

b: the branching factor or maximum number of successors of any node.

d: the depth of the shallowest goal node (i e the number of steps along the path from the root).

m: the maximum length of any path in the state space.

L: depth limit.

Breadth-First Search Performance

Time: O(b^d) [scary]

Space: O(b^d)[Problem store each node]

Complete: Yes (if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes)

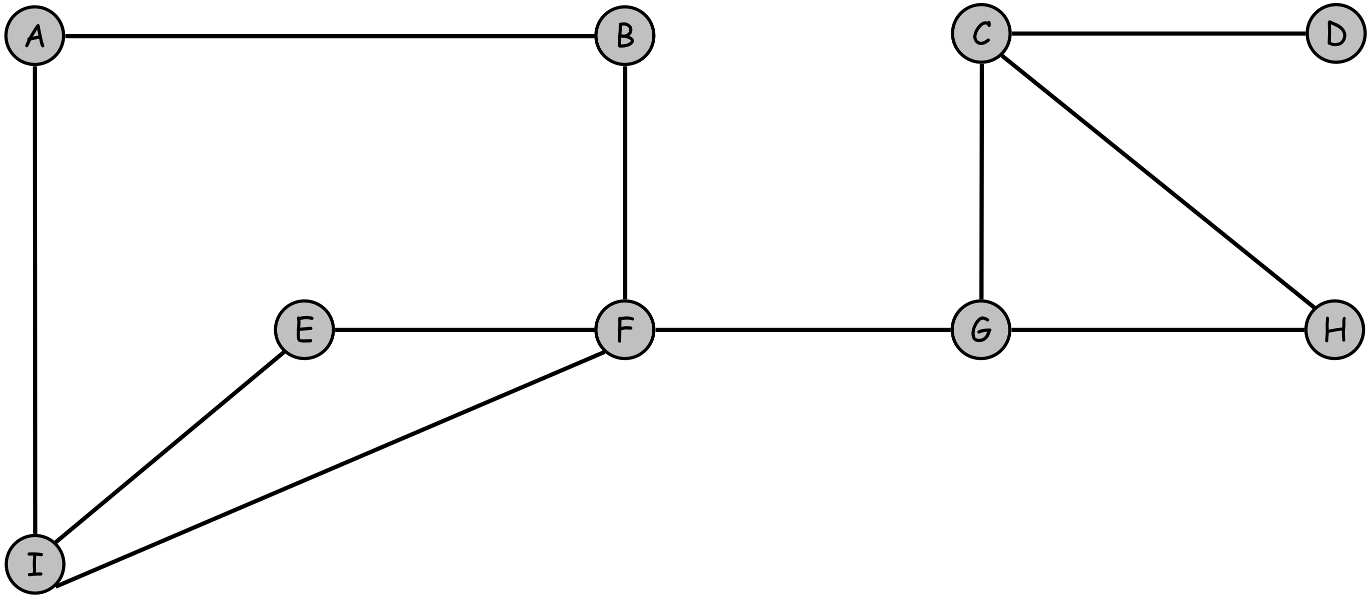
Optimal: Yes (the shallowest goal node is not necessarily the optimal one but, if step costs are all identical??)

Depth-first search

```
 DFS-iterative (G, s):                              //Where G is graph and s is
source vertex
      let S be stack
      S.push( s )           //Inserting s in stack
      mark s as visited.
      while ( S is not empty):
          //Pop a vertex from stack to visit next
          v  =  S.pop( )
         //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                     S.push( w )
                    mark w as visited
```

Depth-first search

Output									



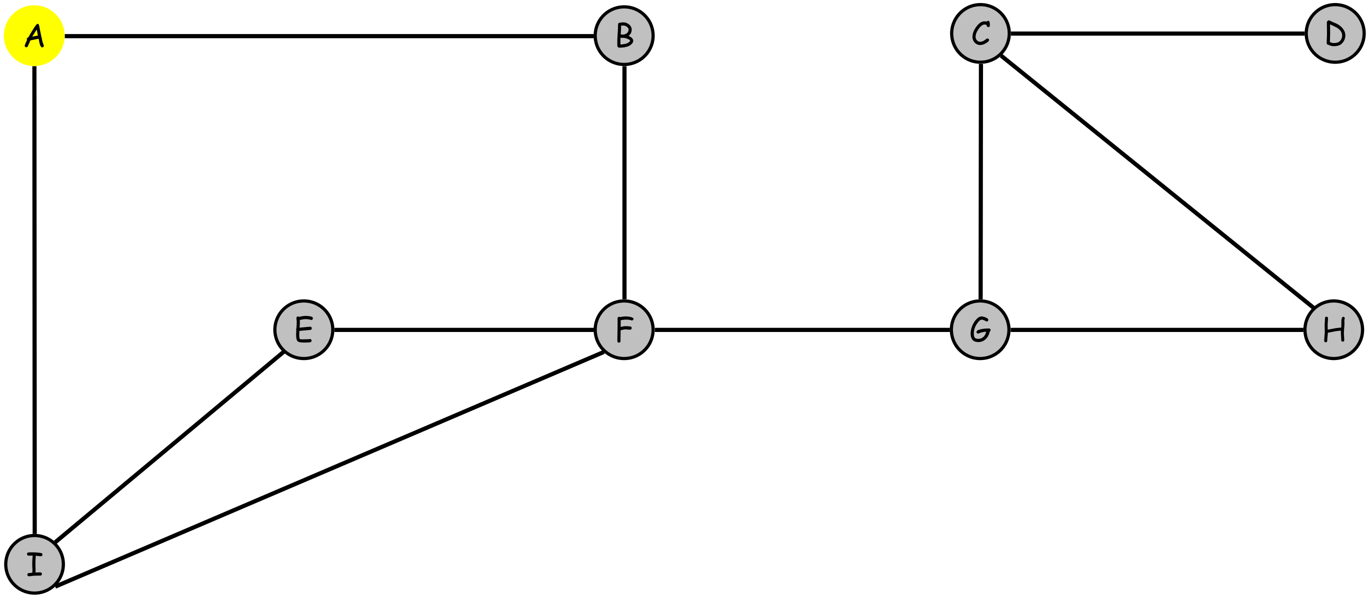
A	B	C	D	E	F	G	H	I

A						

LIFO Stack

Depth-first search

Output	A								

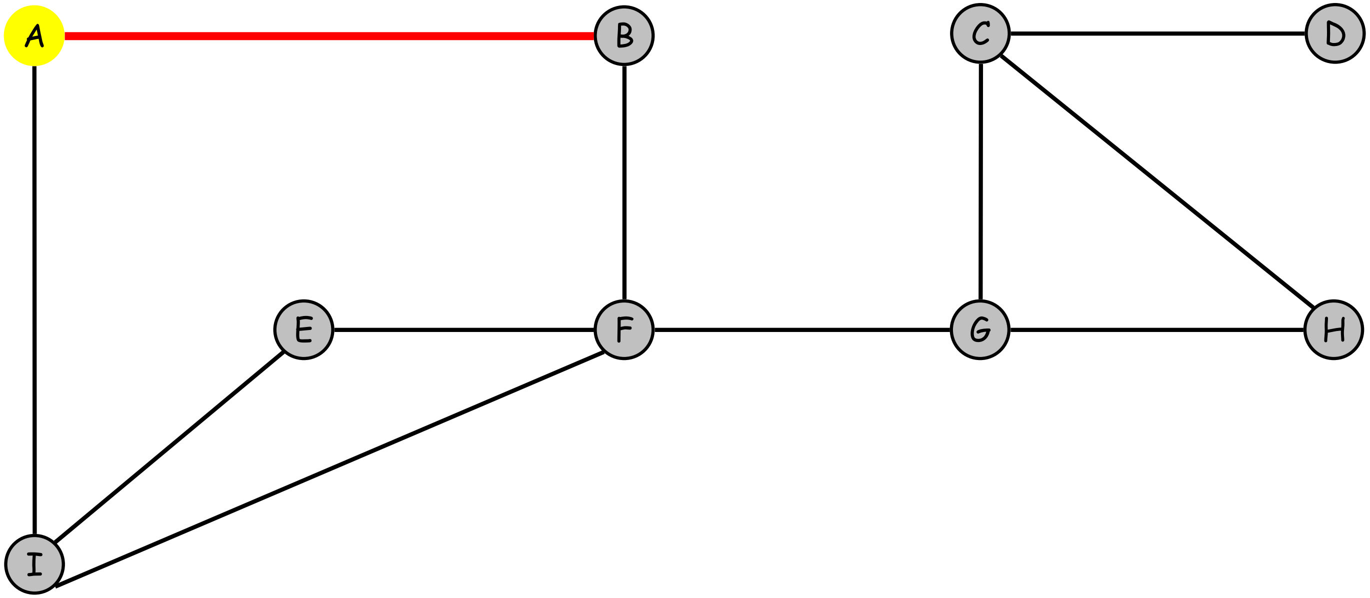


A	B	C	D	E	F	G	H	I
								Y

LIFO Stack

Depth-first search

Output	A								



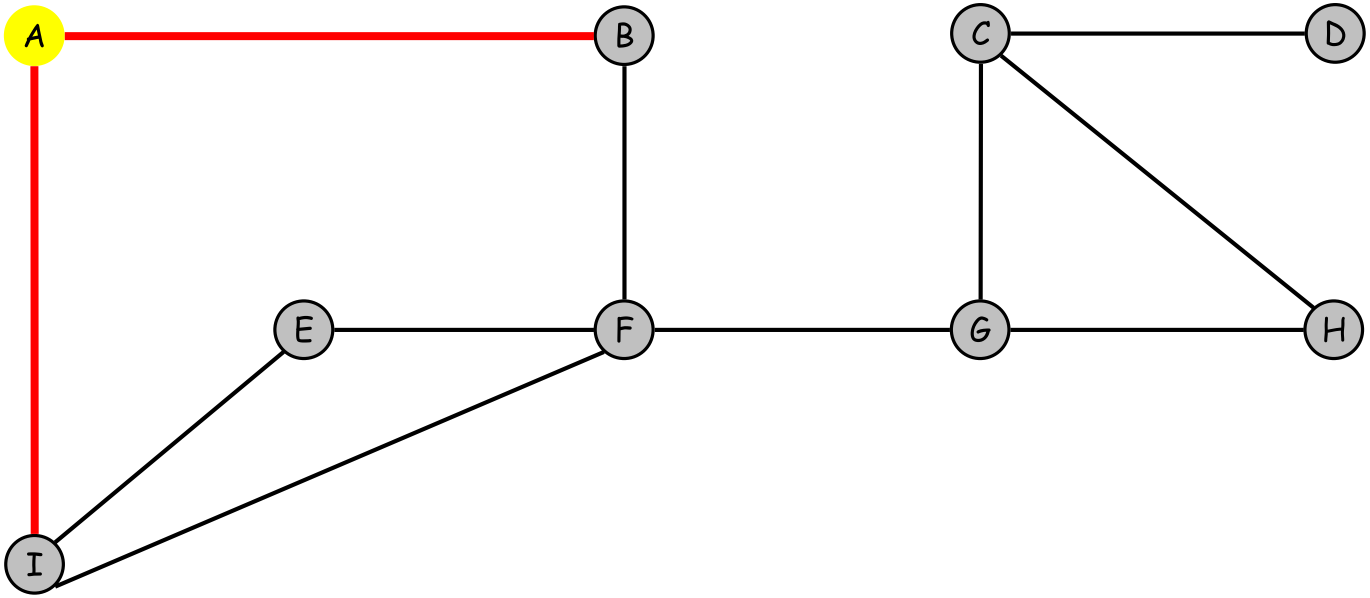
A	B	C	D	E	F	G	H	I
	y							y

B						

LIFO Stack

Depth-first search

Output	A								



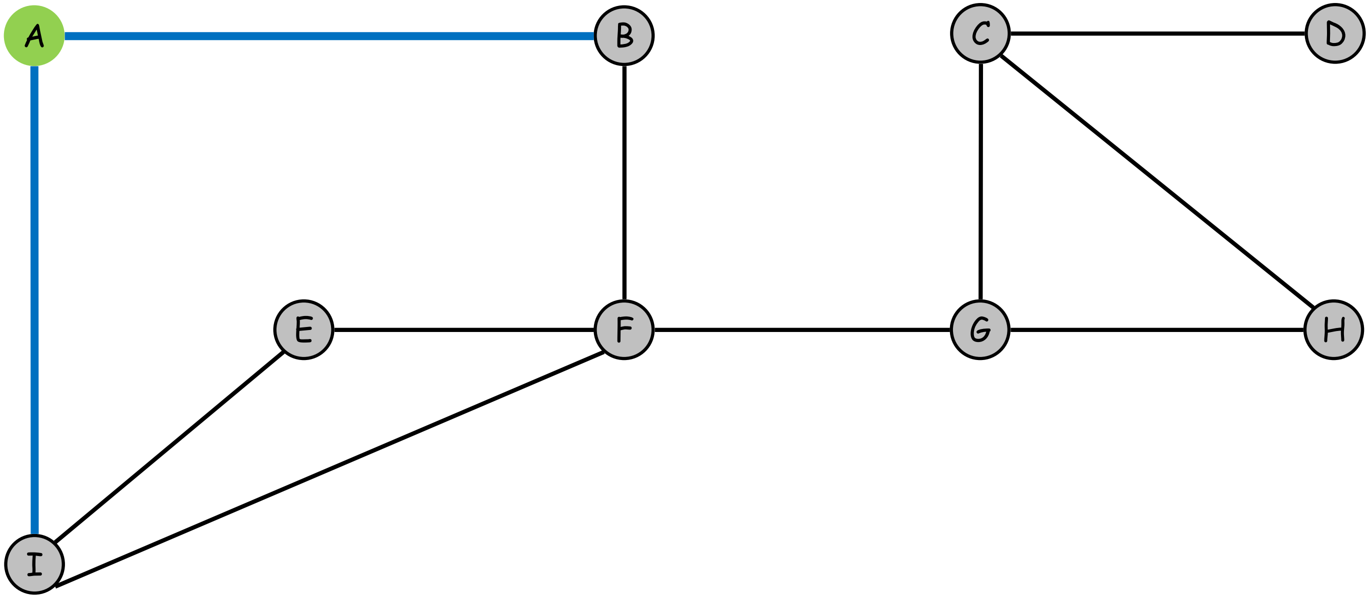
A	B	C	D	E	F	G	H	I
y	y							y

I	B					

LIFO Stack

Depth-first search

Output	A								



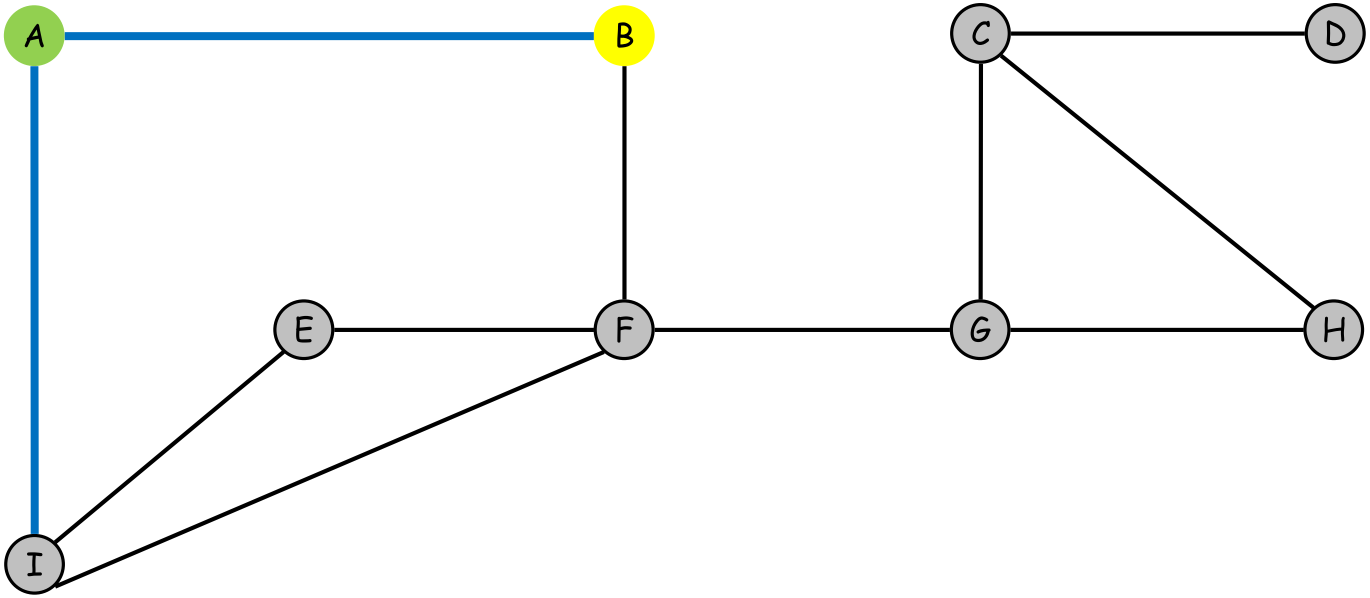
A	B	C	D	E	F	G	H	I
y	y							y

I	B					

LIFO Stack

Depth-first search

Output	A	B						



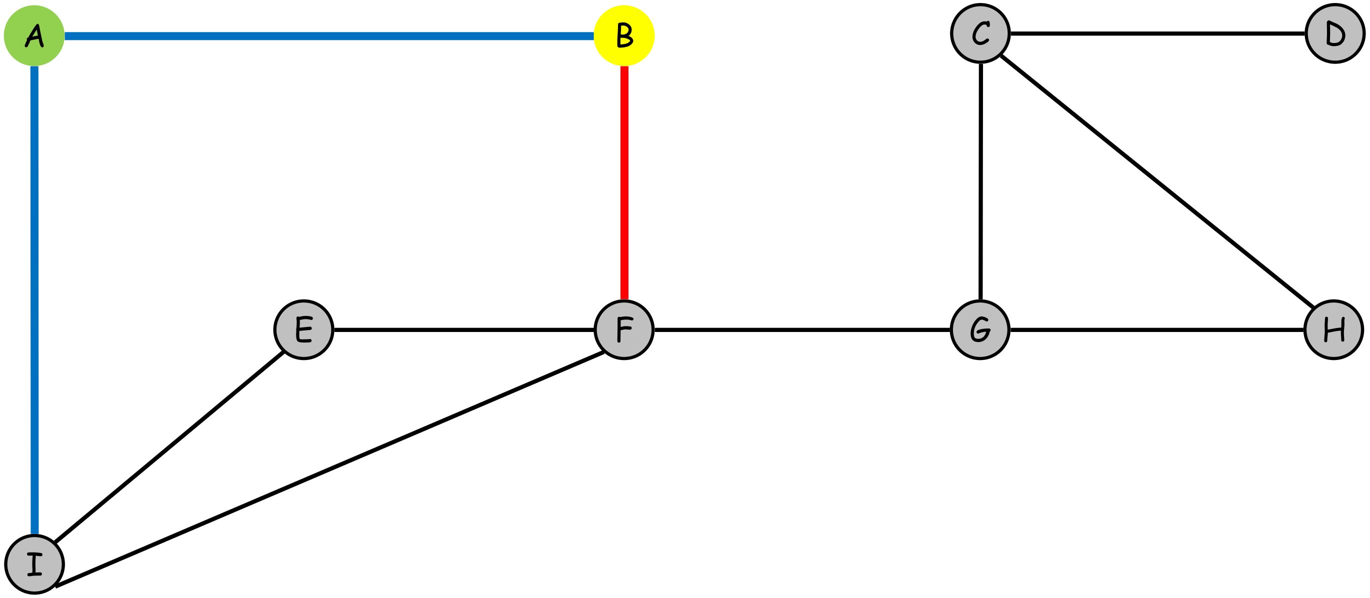
A	B	C	D	E	F	G	H	I
y	y							y

I						

LIFO Stack

Depth-first search

Output	A	B							



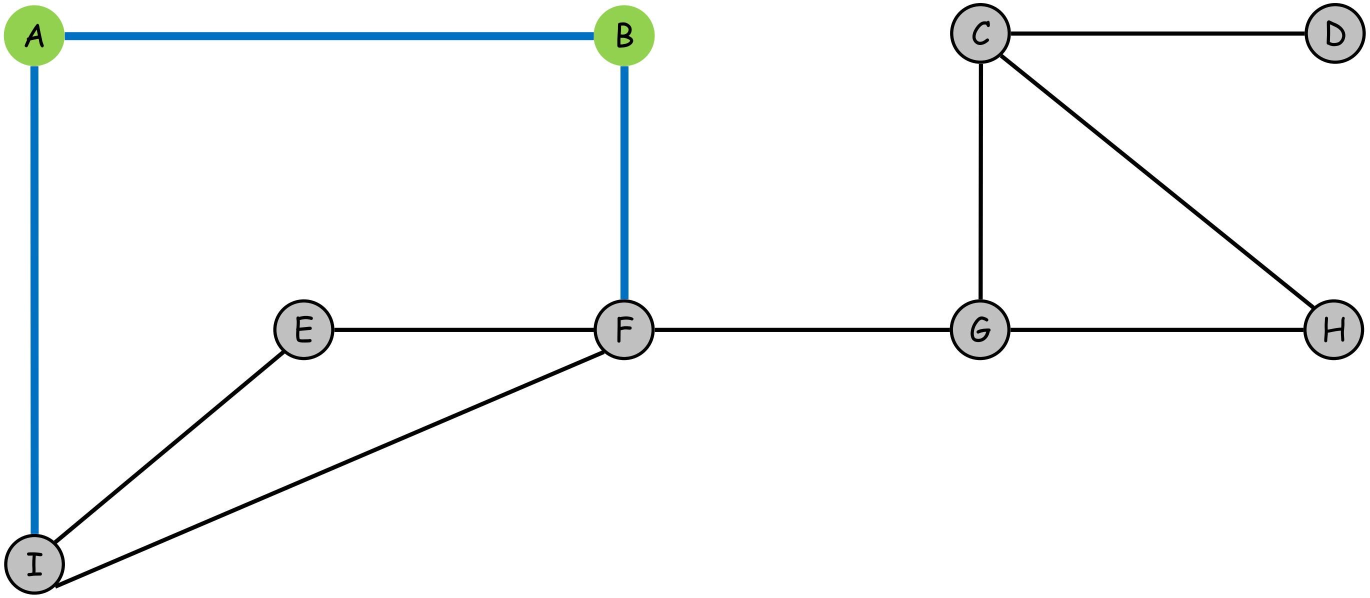
A	B	C	D	E	F	G	H	I
y	y				y			y

I	F					

LIFO Stack

Depth-first search

Output	A	B							



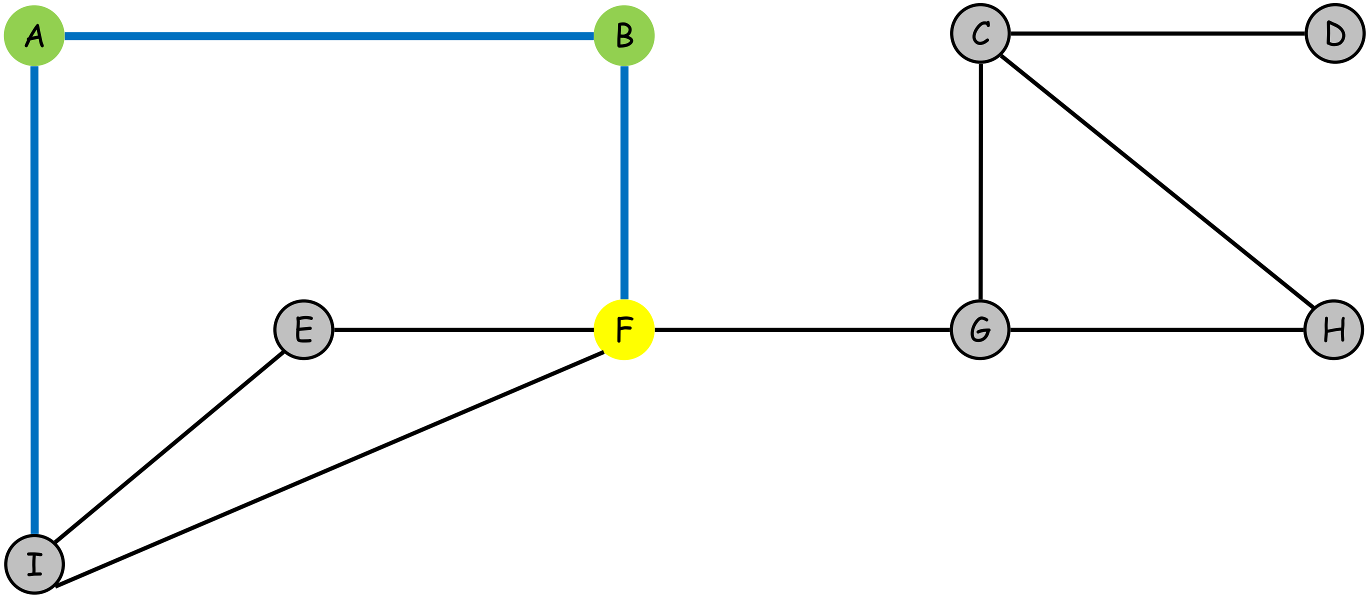
A	B	C	D	E	F	G	H	I
y	y				y			y

I	F					

LIFO Stack

Depth-first search

Output	A	B	F						



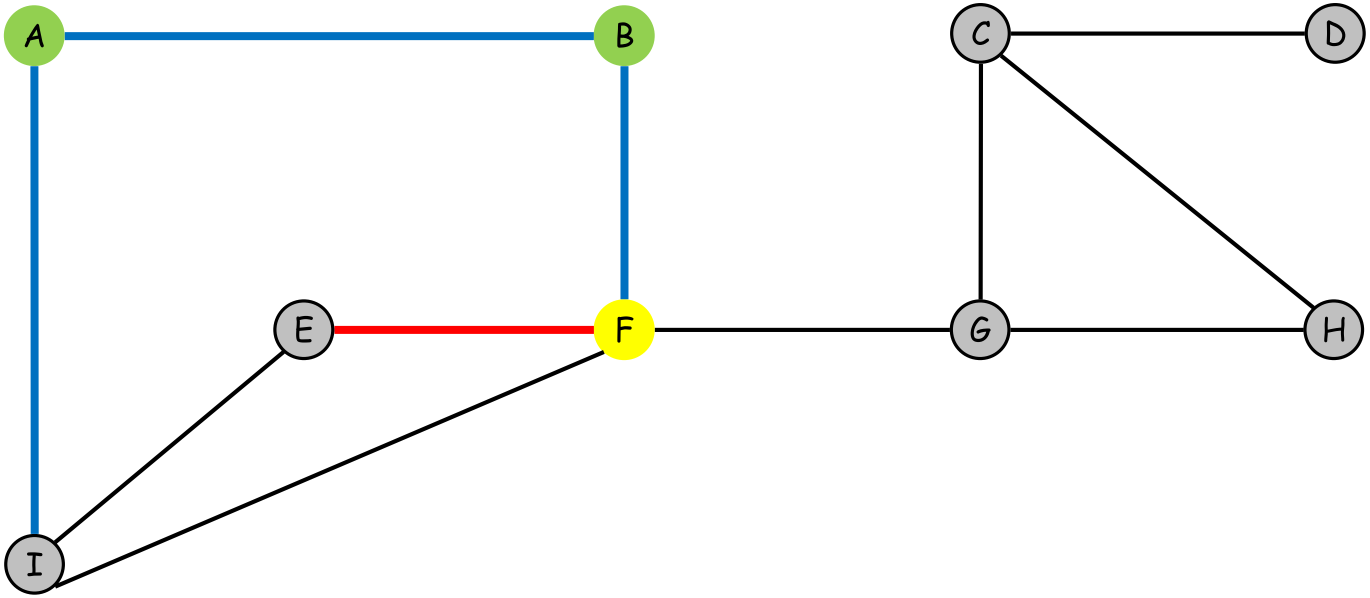
A	B	C	D	E	F	G	H	I
y	y				y			y

I						

LIFO Stack

Depth-first search

Output	A	B	F						



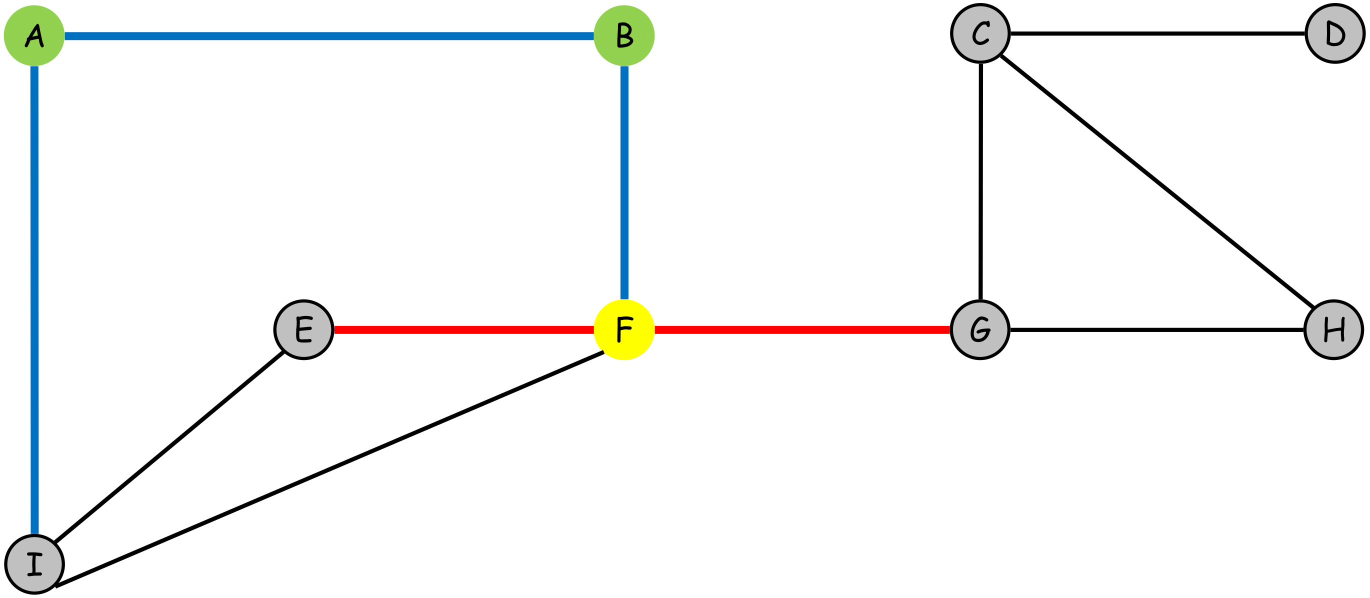
A	B	C	D	E	F	G	H	I
y	y			y	y			y

I	E					

LIFO Stack

Depth-first search

Output	A	B	F						



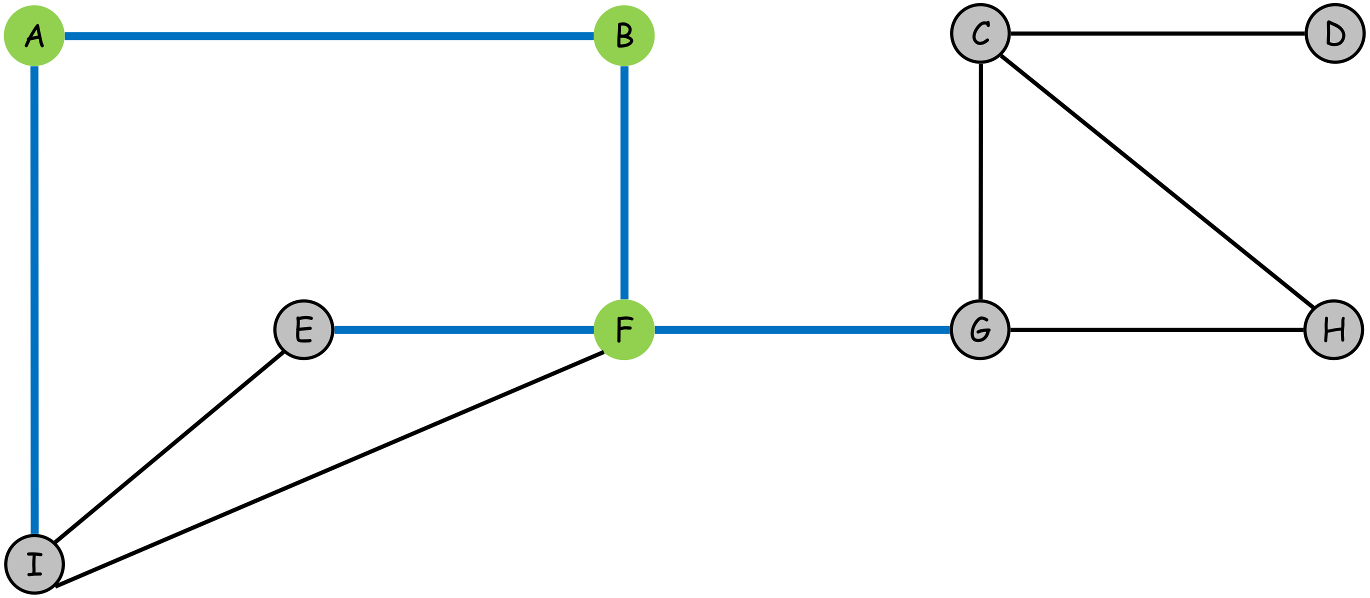
A	B	C	D	E	F	G	H	I
y	y			y	y	y		y

I	E	G				

LIFO Stack

Depth-first search

Output	**A**	**B**	**F**						



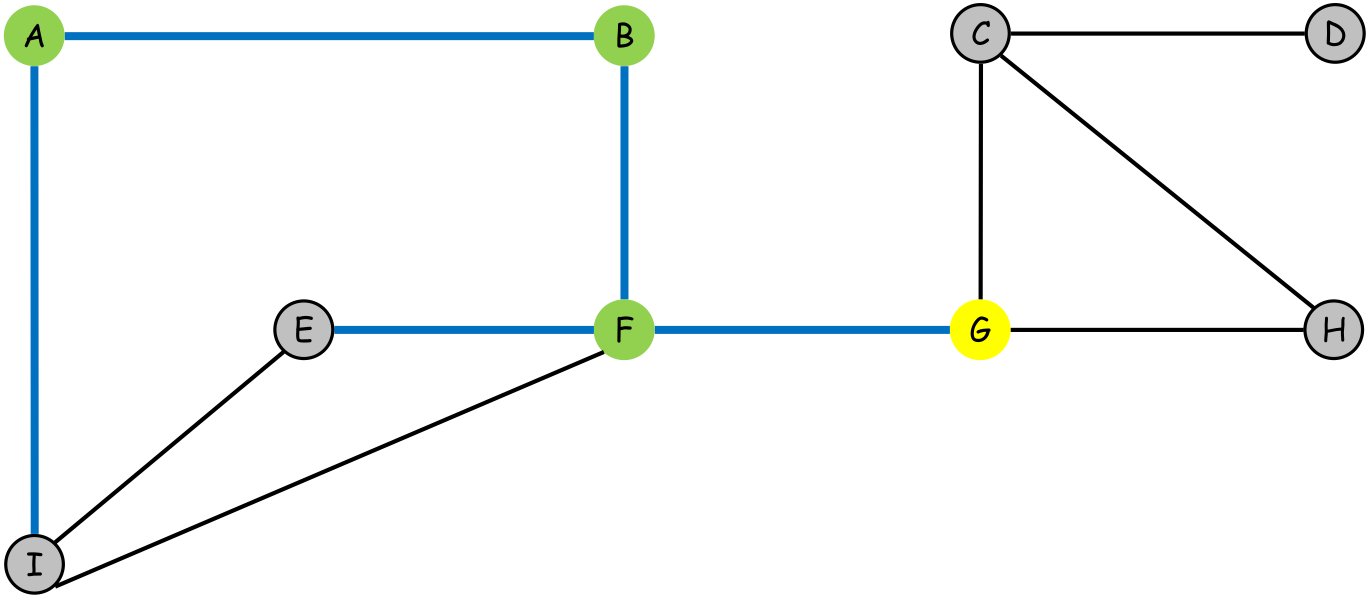
A	B	C	D	E	F	G	H	I
y	y			y	y	y		y

I	E	G				

LIFO Stack

Depth-first search

Output	A	B	F	G					



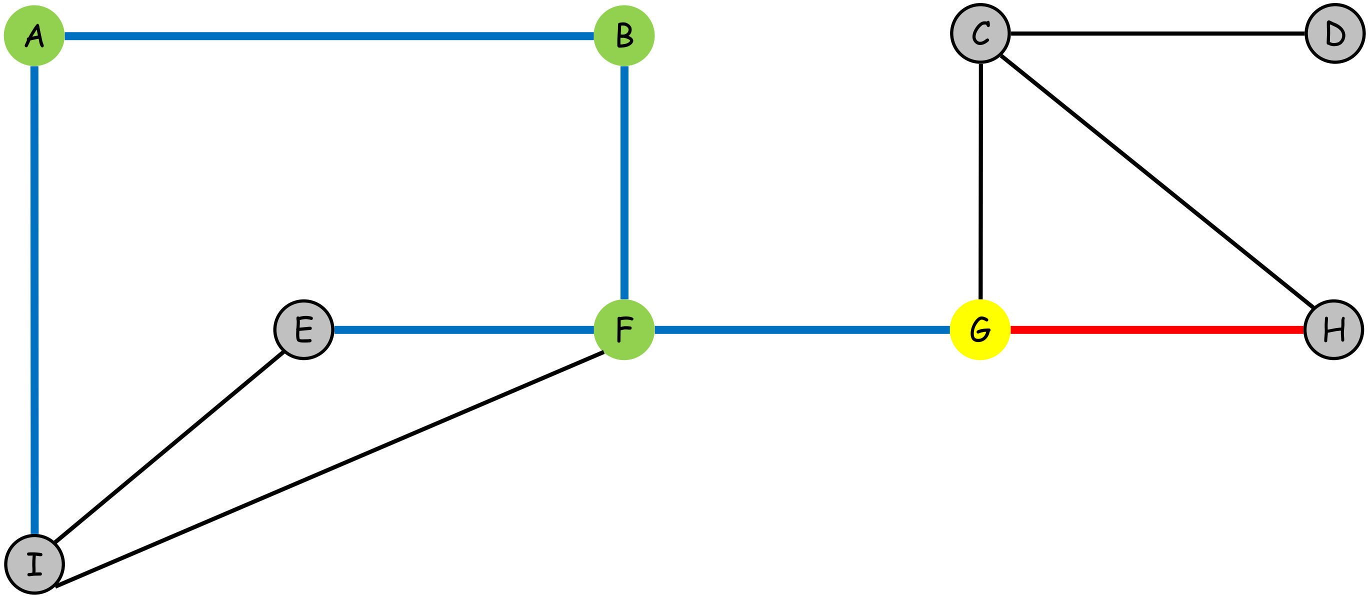
A	B	C	D	E	F	G	H	I
y	y			y	y	y		y

I	E					

LIFO Stack

Depth-first search

Output	A	B	F	G					



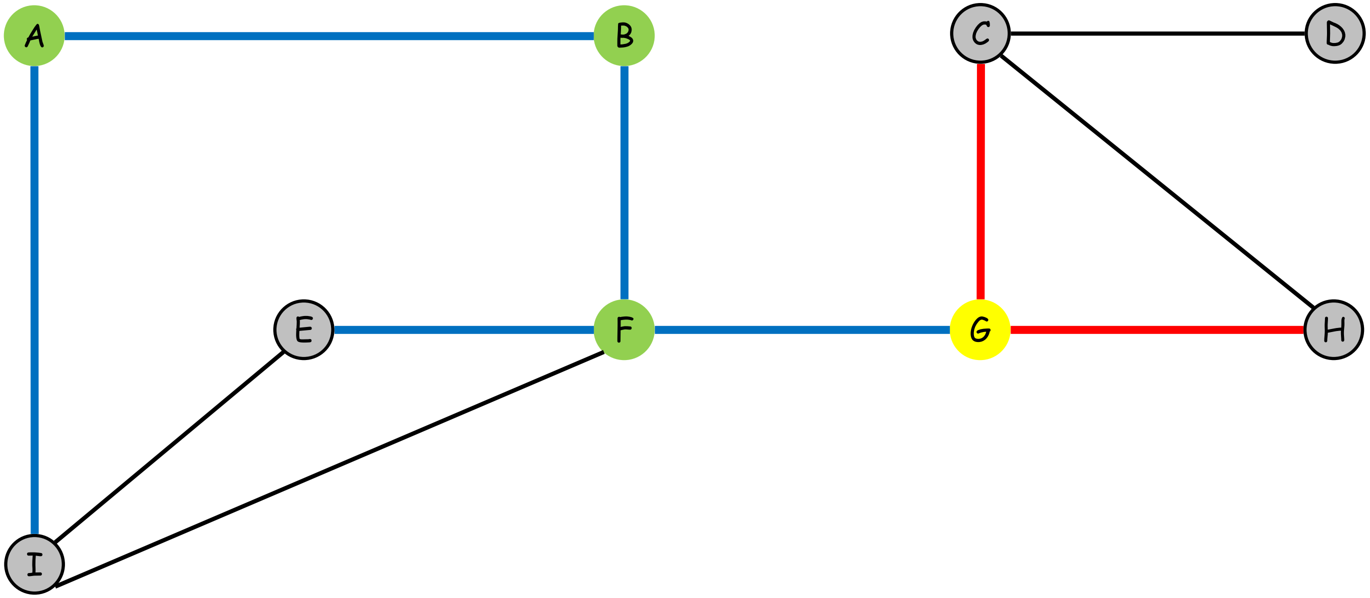
A	B	C	D	E	F	G	H	I
y	y			y	y	y	y	y

I	E	H				

LIFO Stack

Depth-first search

Output	A	B	F	G					



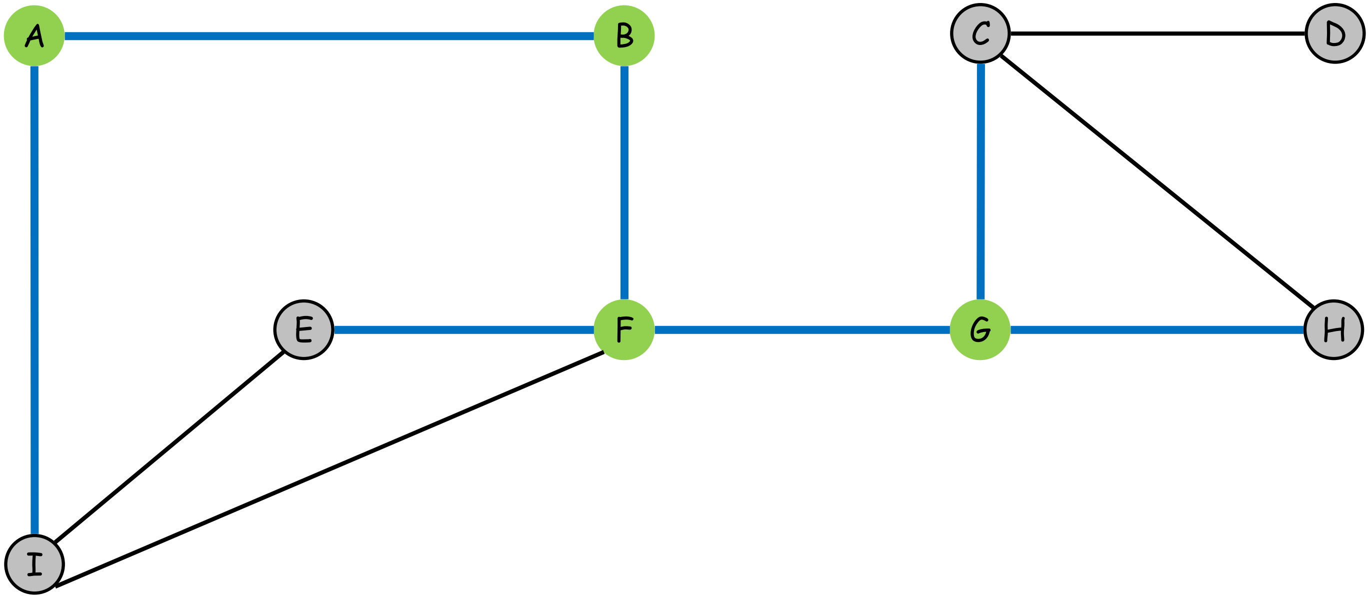
A	B	C	D	E	F	G	H	I
y	y	y		y	y	y	y	y

I	E	H	C			

LIFO Stack

Depth-first search

Output	A	B	F	G					



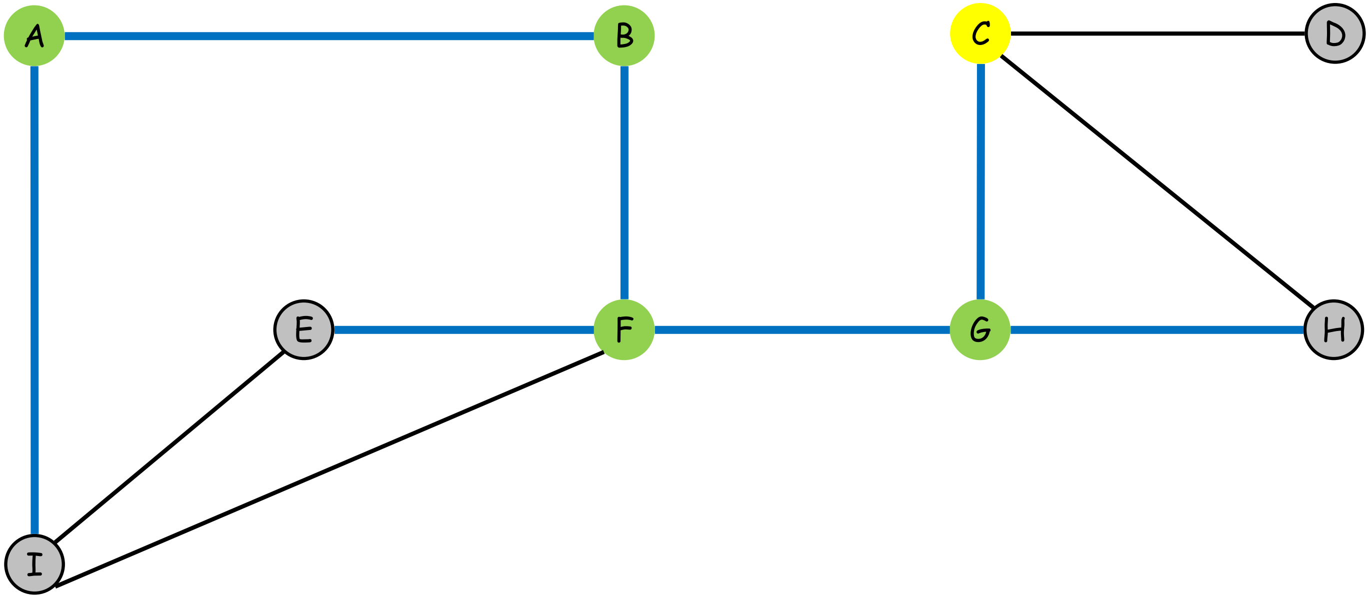
A	B	C	D	E	F	G	H	I
y	y	y		y	y	y	y	y

I	E	H	C			

LIFO Stack

Depth-first search

Output	A	B	F	G	C				



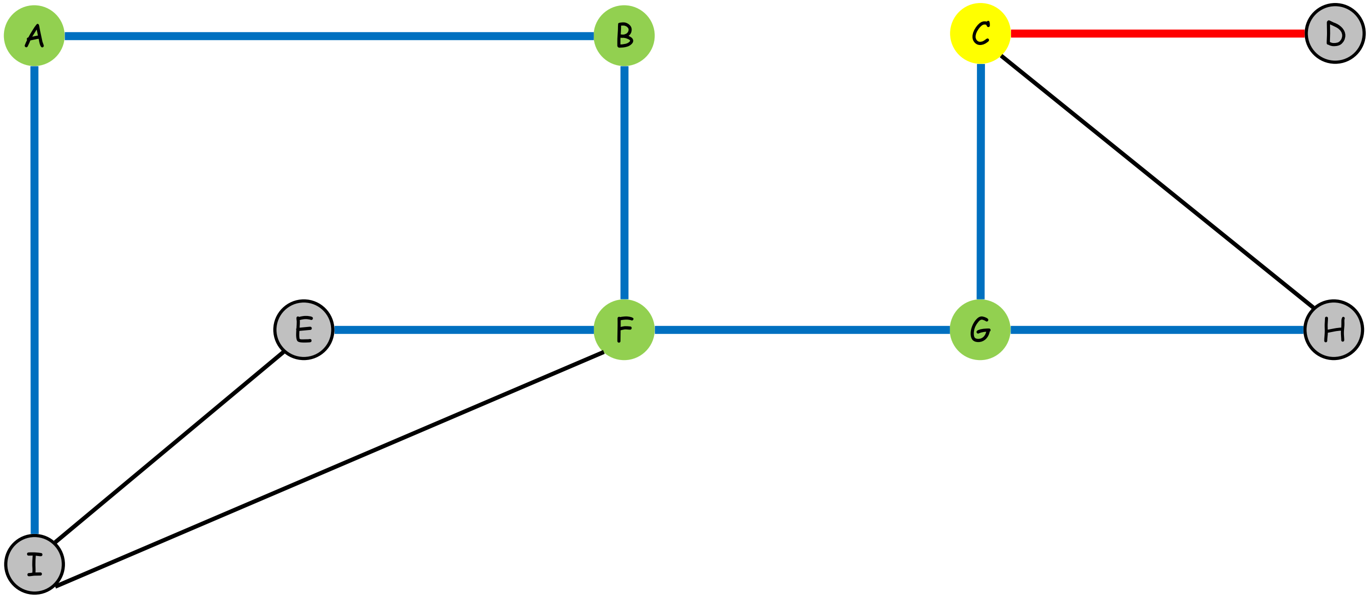
A	B	C	D	E	F	G	H	I
y	y	y		y	y	y	y	y

I	E	H				

LIFO Stack

Depth-first search

Output	A	B	F	G	C				



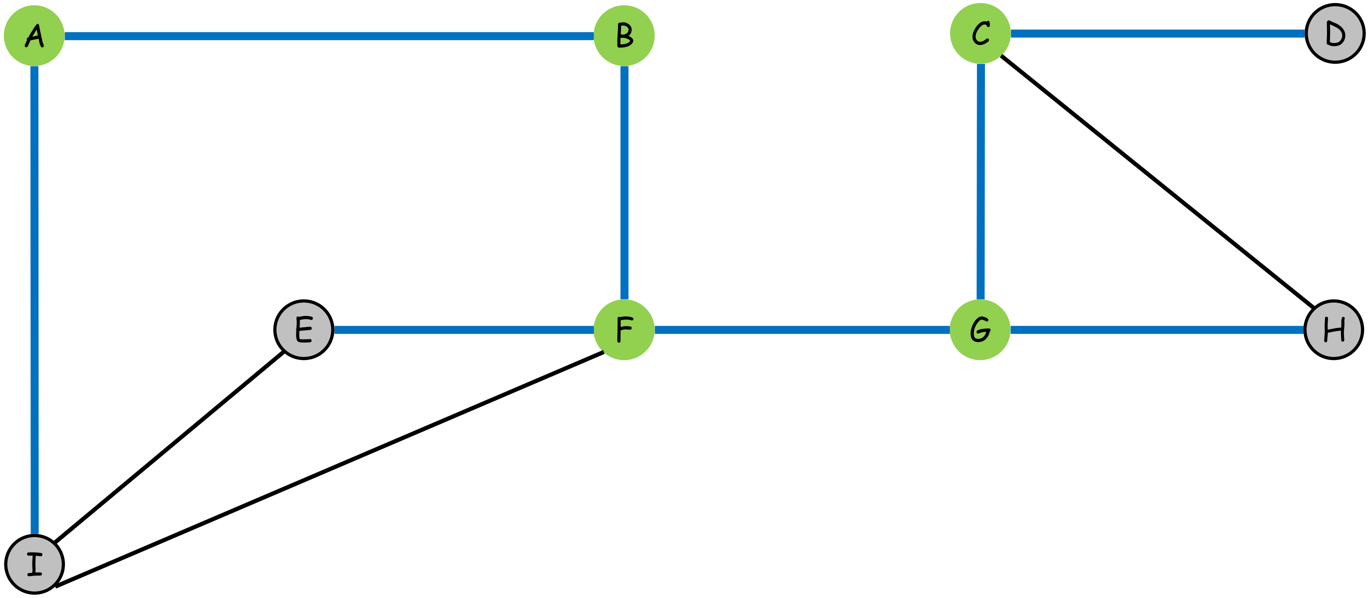
A	B	C	D	E	F	G	H	I
y	y	y	y	y	y	y	y	y

I	E	H	D			

LIFO Stack

Depth-first search

Output	A	B	F	G	C			



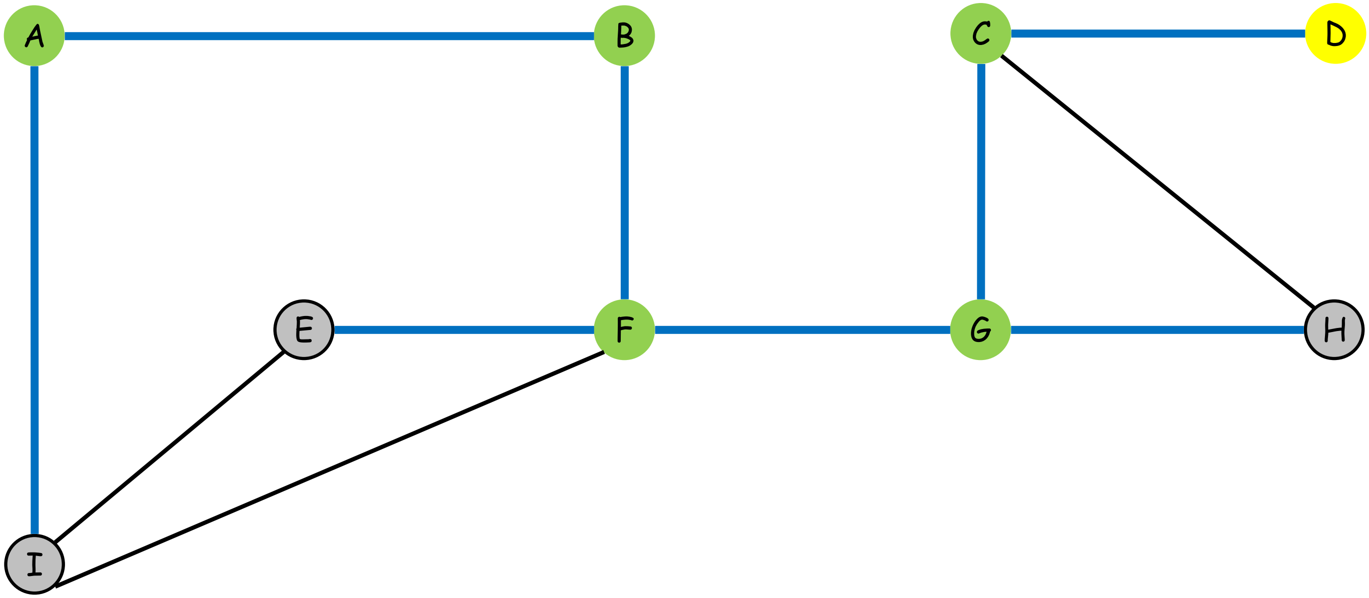
A	B	C	D	E	F	G	H	I
y	y	y	y	y	y	y	y	y

I	E	H	D			

LIFO Stack

Depth-first search

Output	A	B	F	G	C	D			



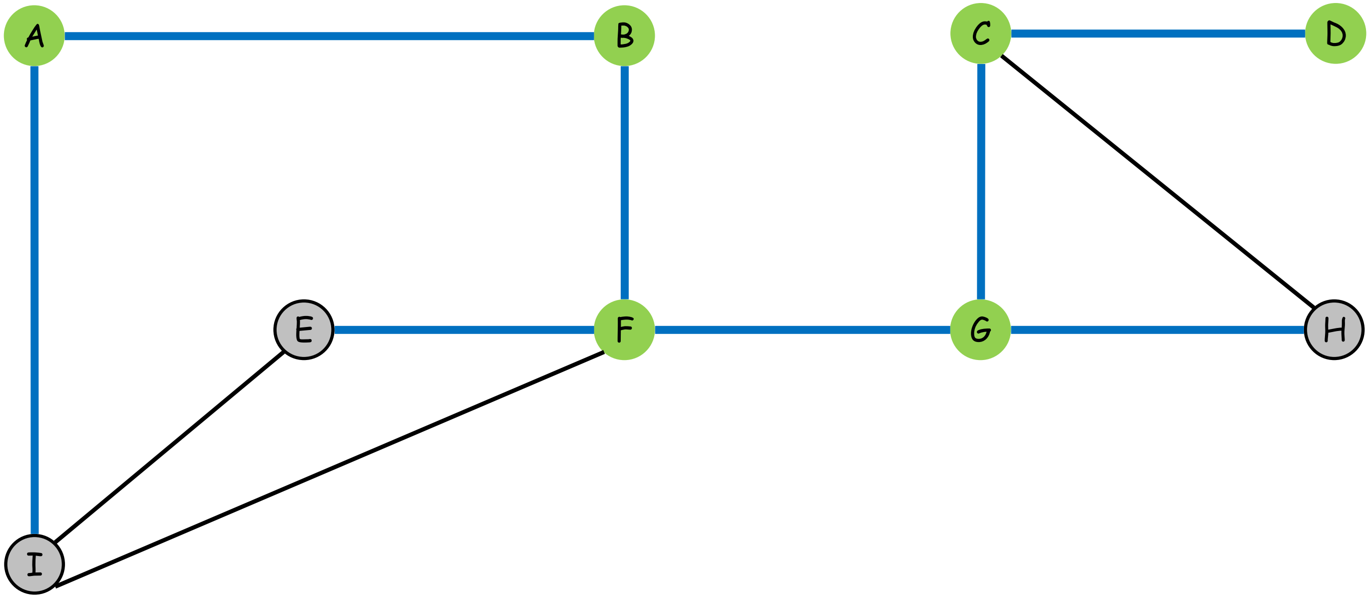
A	B	C	D	E	F	G	H	I
y	y	y	y	y	y	y	y	y

I	E	H				

LIFO Stack

Depth-first search

Output	A	B	F	G	C	D			



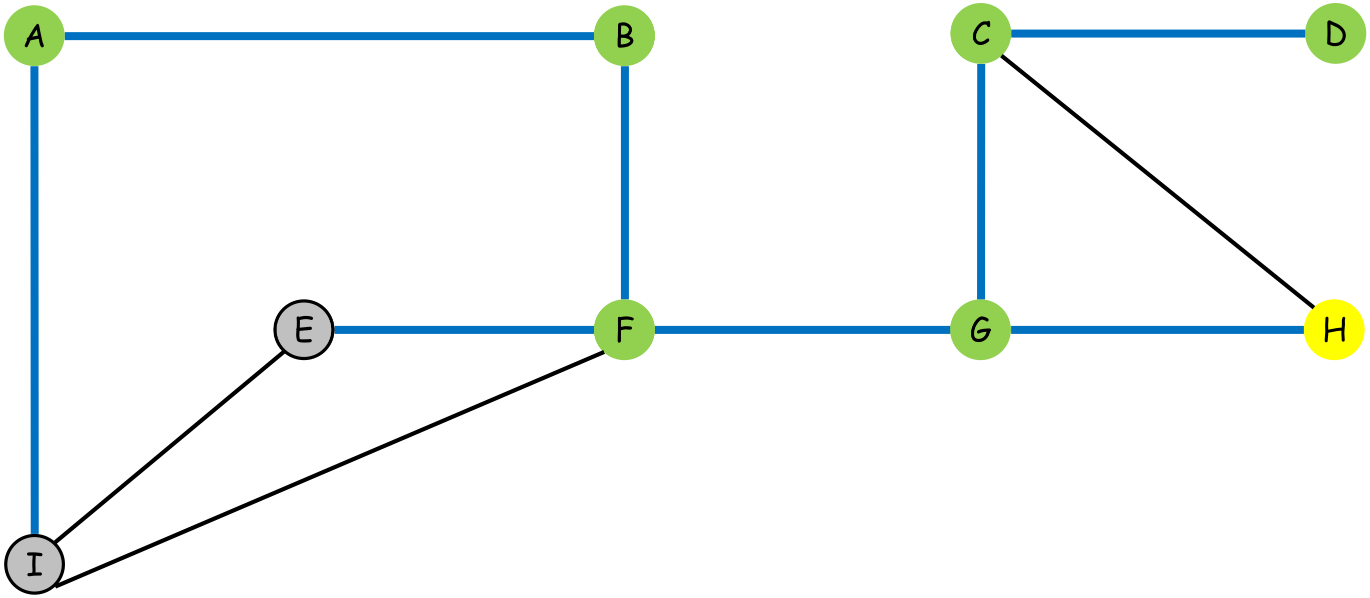
A	B	C	D	E	F	G	H	I
y	y	y	y	y	y	y	y	y

I	E	H				

LIFO Stack

Depth-first search

Output	A	B	F	G	C	D	H		



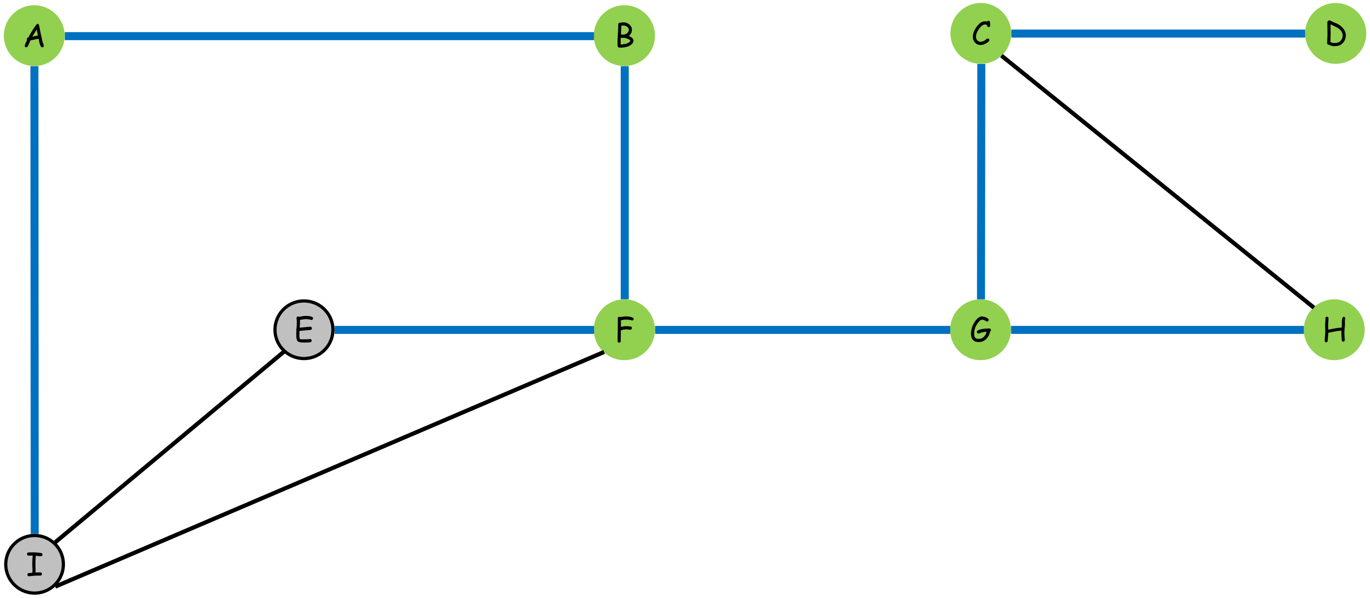
A	B	C	D	E	F	G	H	I
y	y	y	y	y	y	y	y	y

I	E					

LIFO Stack

Depth-first search

Output	A	B	F	G	C	D	H		



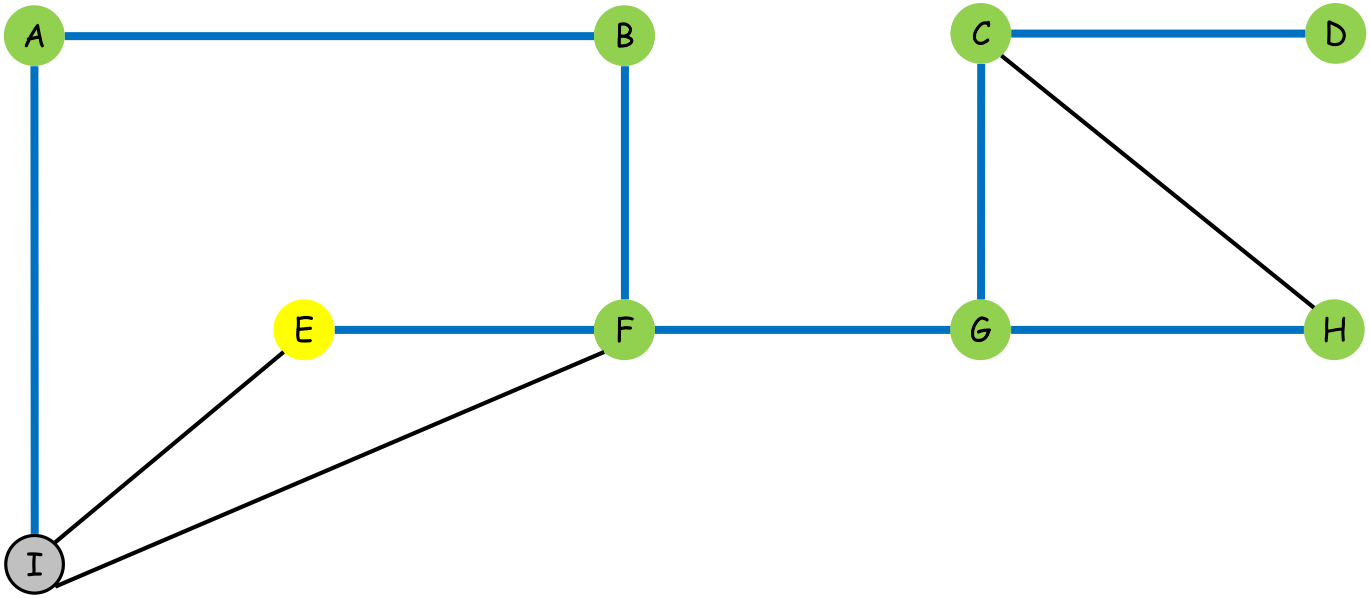
A	B	C	D	E	F	G	H	I
y	y	y	y	y	y	y	y	y

I	E					

LIFO Stack

Depth-first search

Output	A	B	F	G	C	D	H	E	



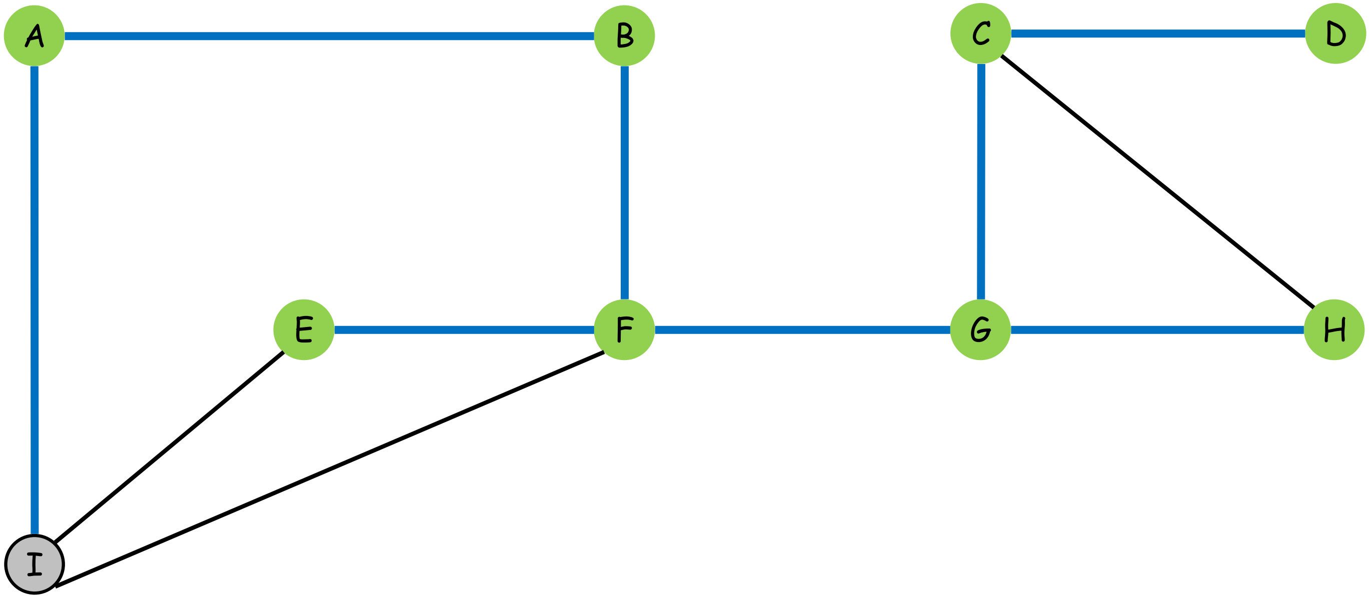
A	B	C	D	E	F	G	H	I
y	y	y	y	y	y	y	y	y

I						

LIFO Stack

Depth-first search

Output	A	B	F	G	C	D	H	E	



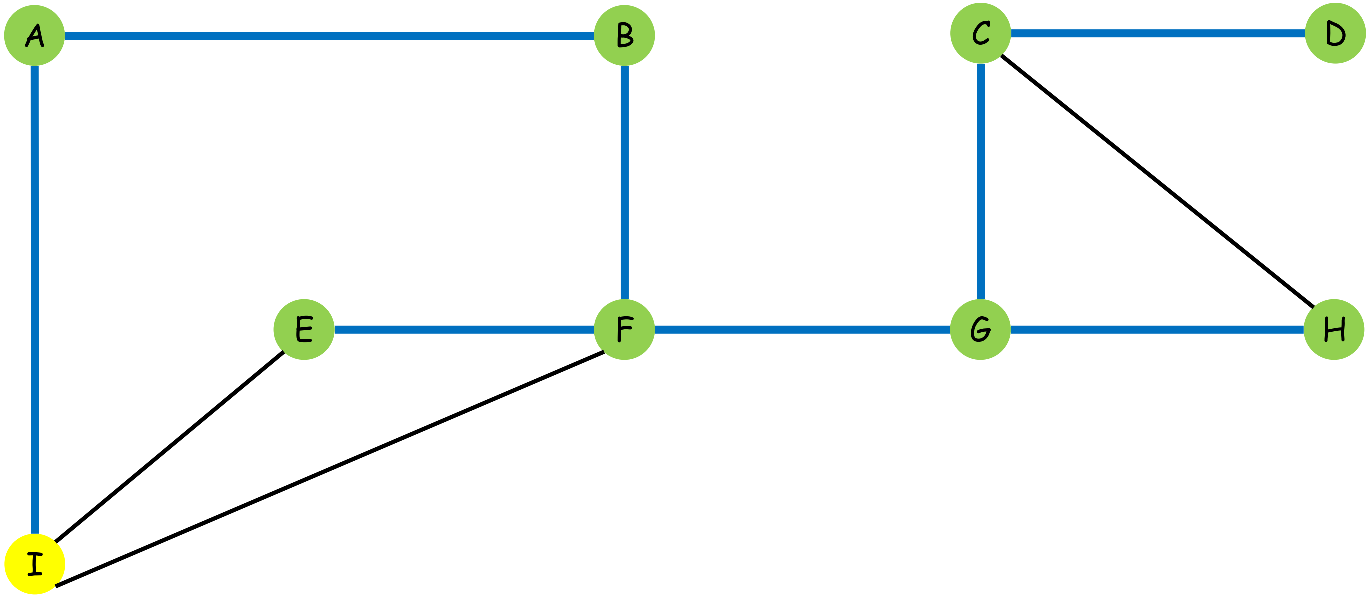
A	B	C	D	E	F	G	H	I
y	y	y	y	y	y	y	y	y

I						

LIFO Stack

Depth-first search

Output	A	B	F	G	C	D	H	E	I

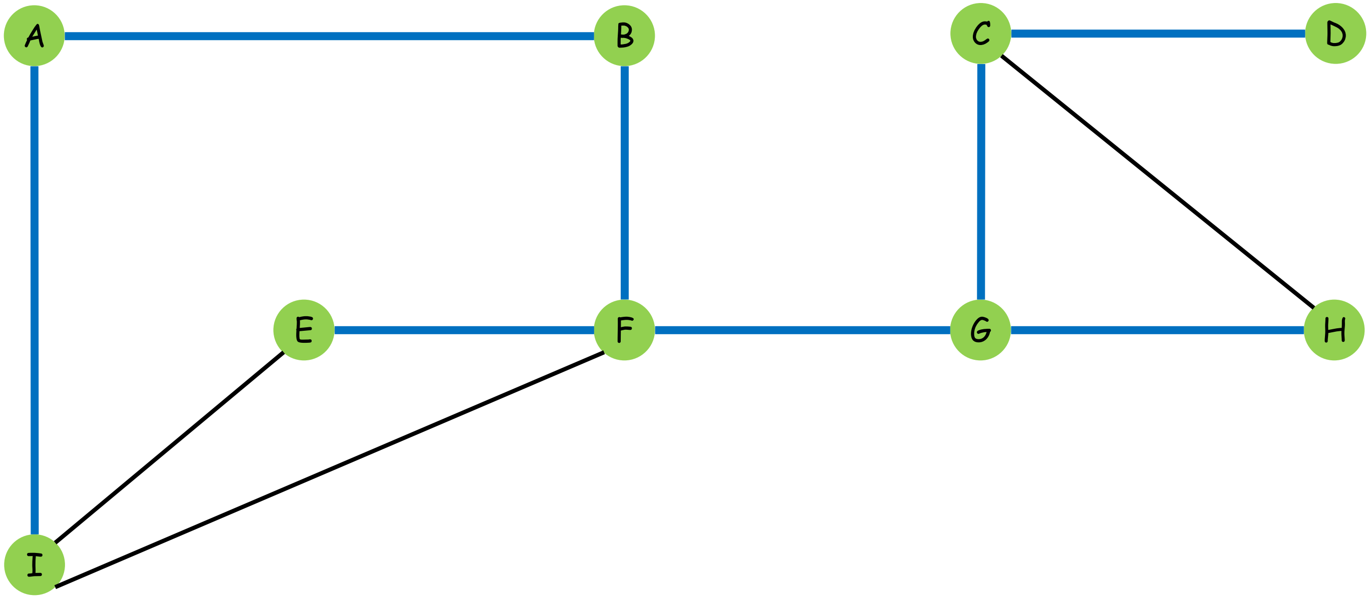


A	B	C	D	E	F	G	H	I
y	y	y	y	y	y	y	y	y

LIFO Stack

Depth-first search

Output	A	B	F	G	C	D	H	E	I



A	B	C	D	E	F	G	H	I
y	y	y	y	y	y	y	y	y

LIFO Stack

Algorithm Performance

Completeness: Is the algorithm guaranteed to find a solution when there is one

Optimality: Does the strategy find the optimal solution

Time complexity: How long does it take to find a solution

Space complexity: How much memory is needed to perform the search

Algorithm Performance

b: the branching factor or maximum number of successors of any node.

d: the depth of the shallowest goal node (i e the number of steps along the path from the root).

m: the maximum length of any path in the state space.

L: depth limit.

Depth-First Search Performance

Time: O(b^m)

- (Note that m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.)

Space: O(bm)

- (depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.)

Complete: No

- (Because of Cycle and In infinite state spaces, fail if an infinite non-goal path is encountered.)

Optimal: No