Programming Abstractions

CS106B

Cynthia Bailey

Chris Gregg

pollev.com/cs106b

Today's Topics

Today:

- Graphs
  - Terminology
  - Basics
  - A couple samples of classic Graph Theory problems (Hamiltonian Path and the Good Will Hunting movie problem)

Next time:

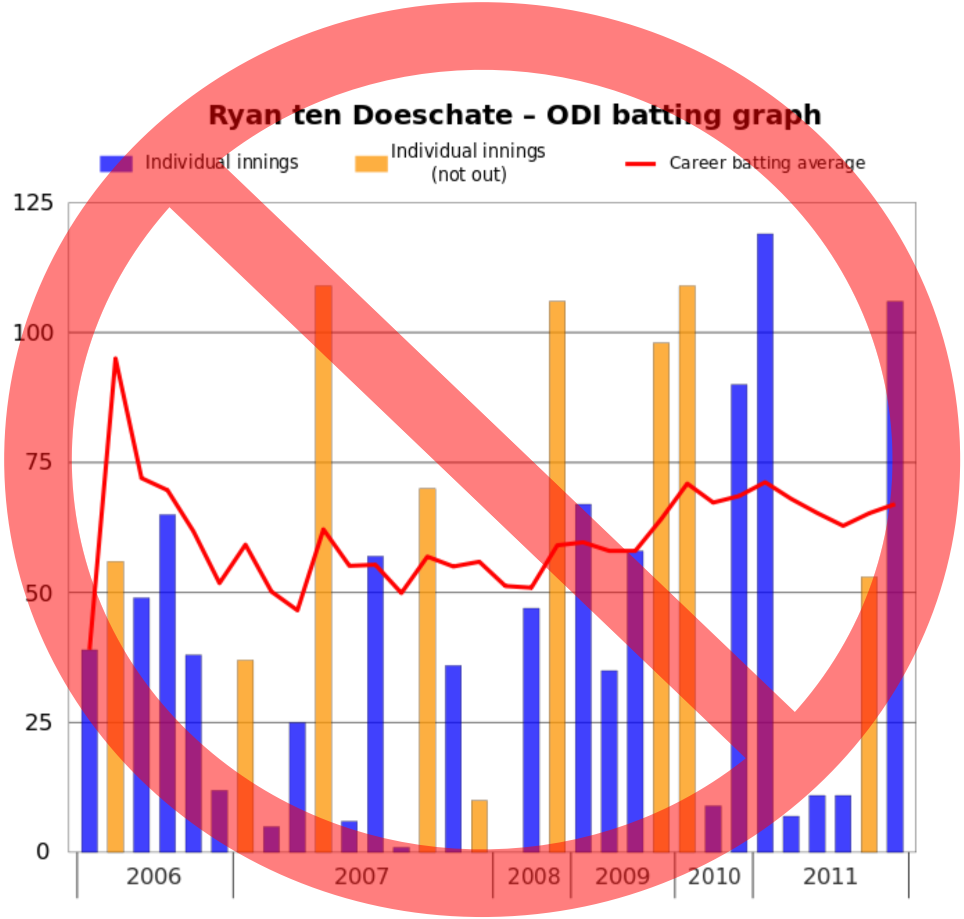
- Another classic graph problem: shortest paths
  - BFS
  - Dijkstra's algorithm

- **For important announcements, be sure to see the weekly announcements post on the Ed Q&A board! https://edstem.org**
- **Also on Ed: live lecture Q&A with Chris & Jonathan**

Graphs

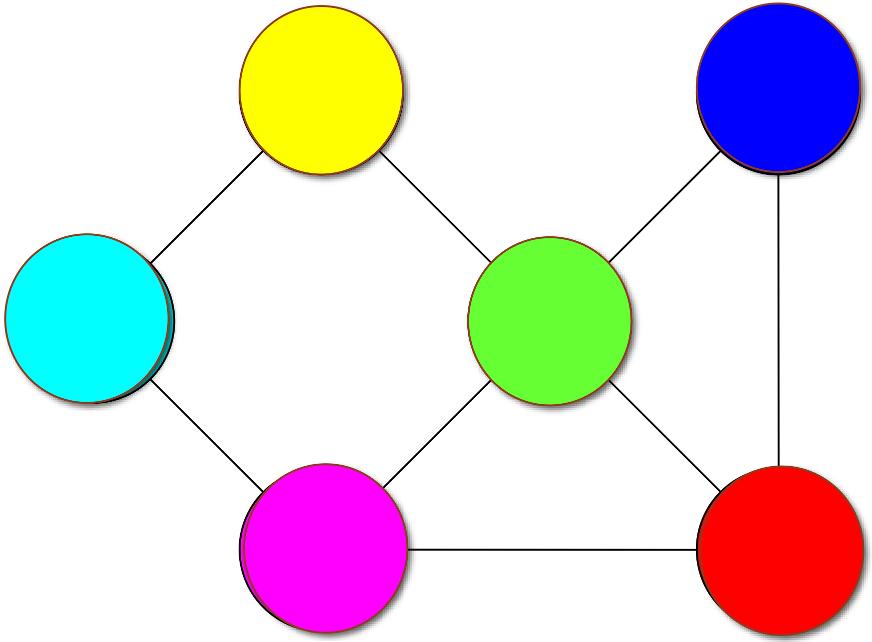
What are graphs? What are they good for?

Graph



Ryan ten Doeschate - ODI batting graph
Individual innings
Individual innings (not out)
Career batting average
125
100
75
50
25
0
2006
2007
2008
2009
2010
2011

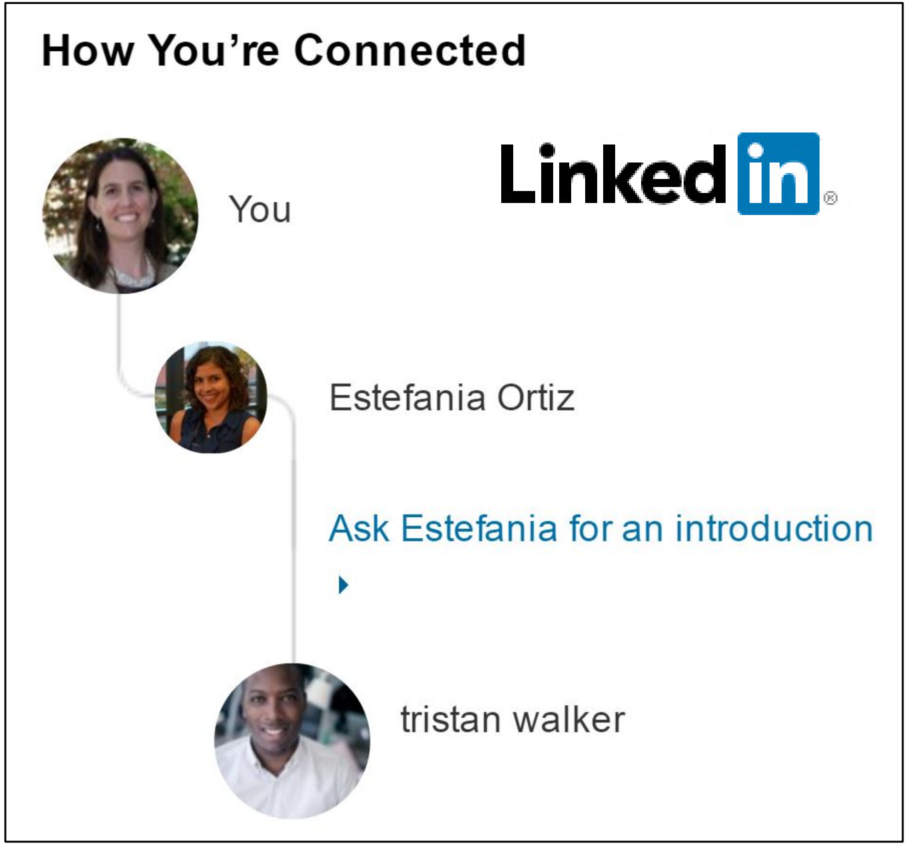
Graphs in Computer Science



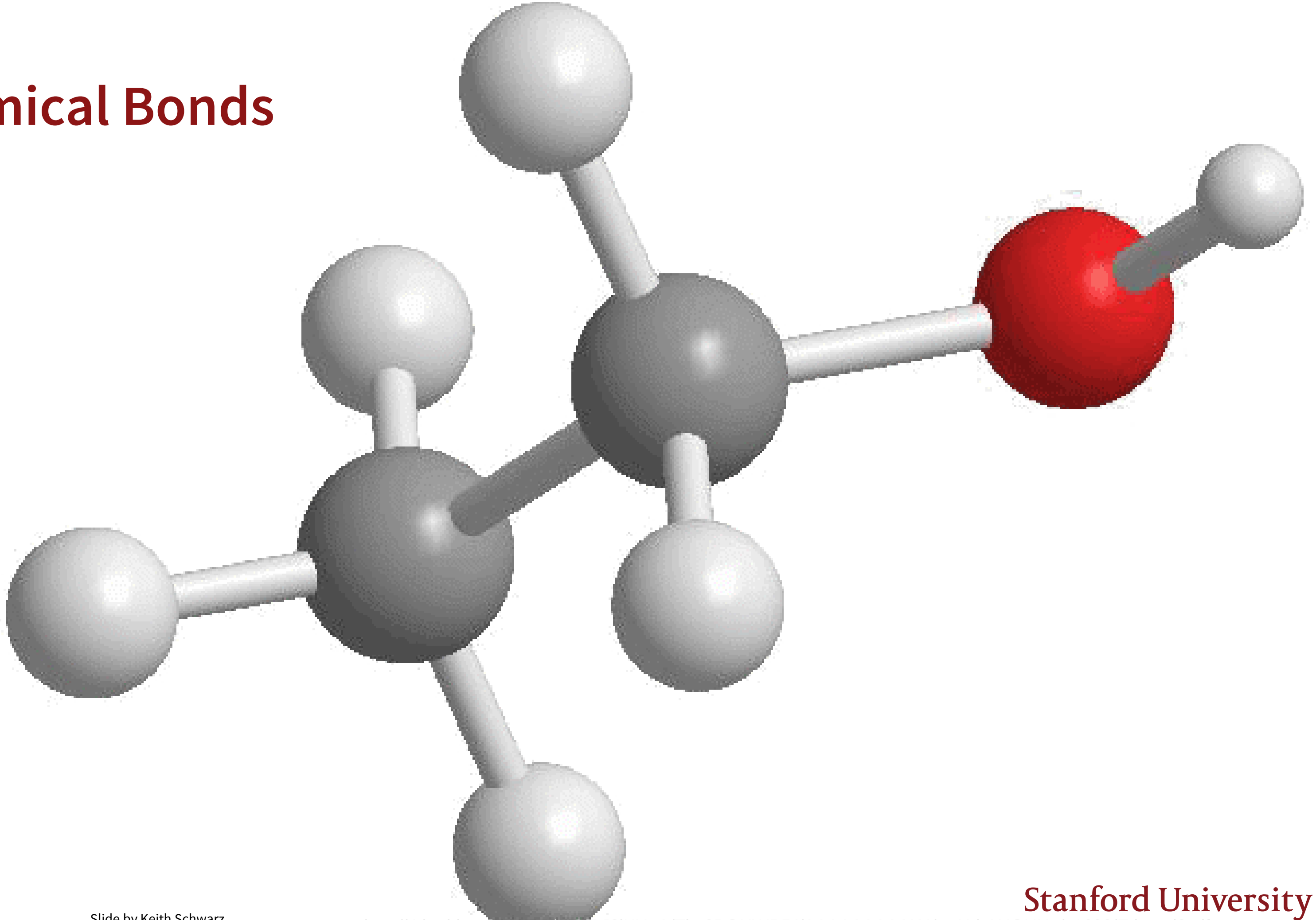
A graph is a mathematical structure for representing relationships

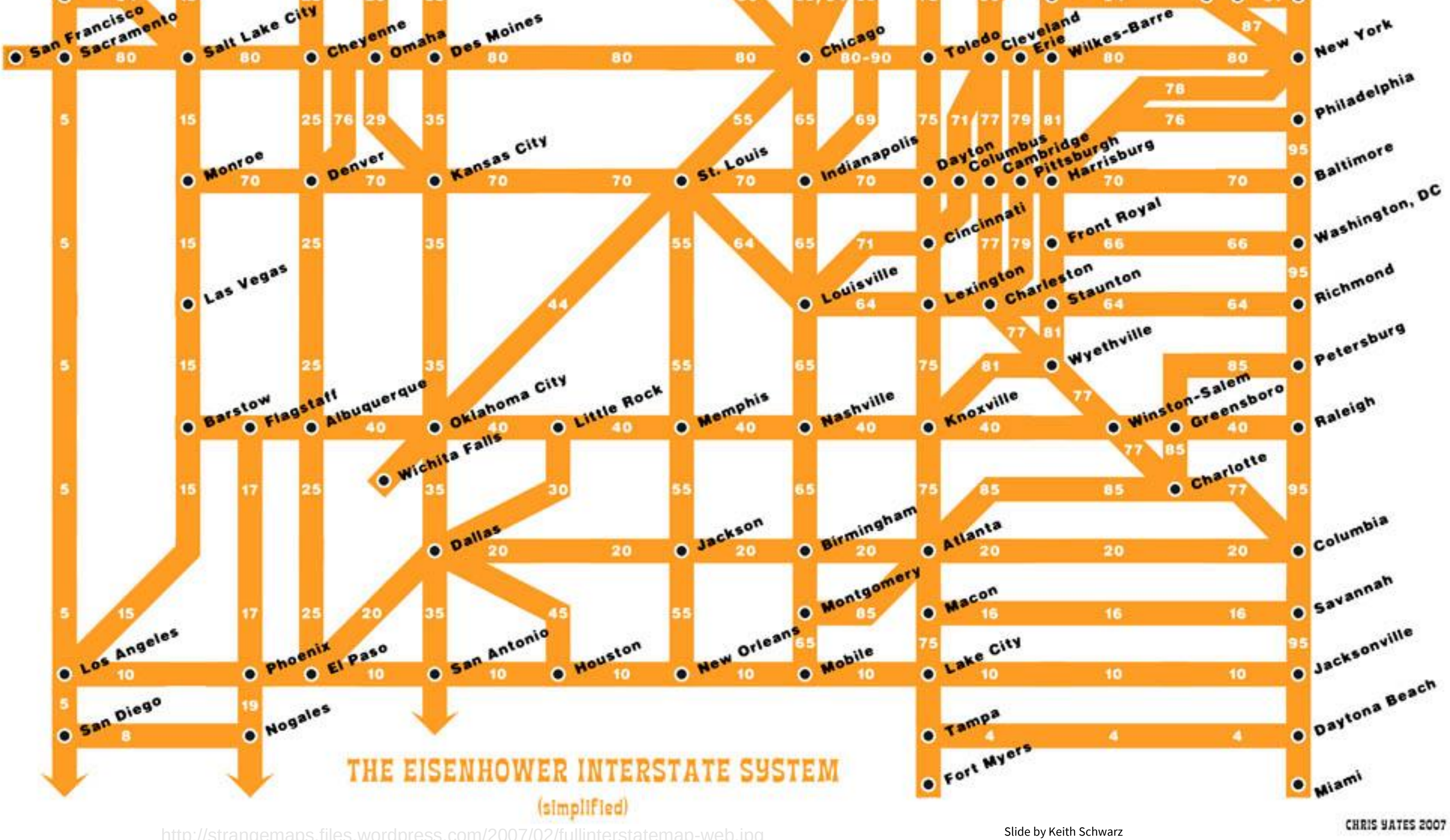
- A set V of **nodes** (or **vertices**)
- A set E of **edges** (or *arcs*) connecting a pair of vertices

A Social Network



Chemical Bonds

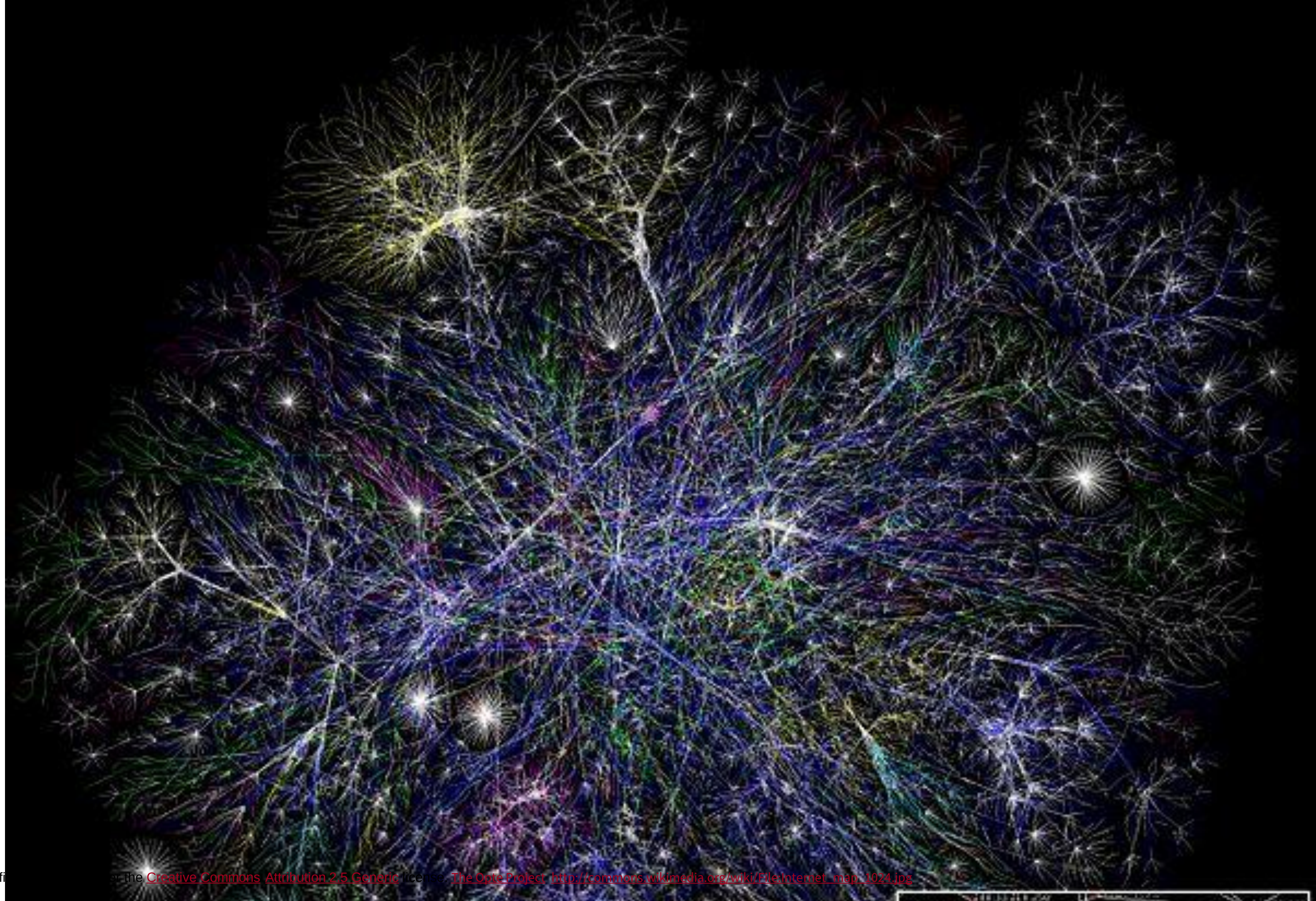




http://strangemaps.files.wordpress.com/2007/02/fullinterstatemap-web.jpg

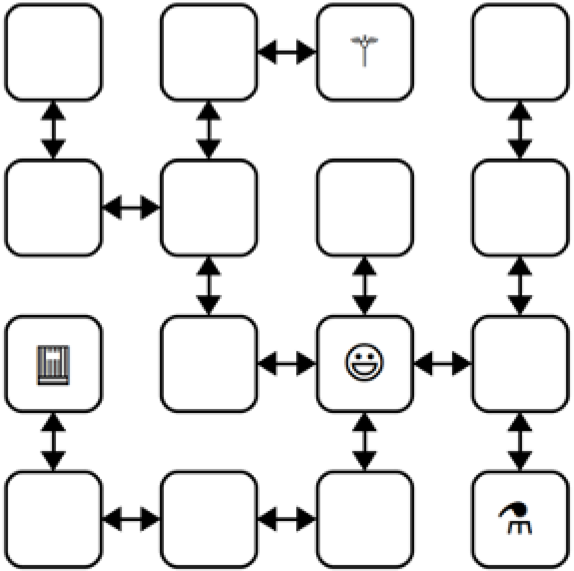
Slide by Keith Schwarz

Internet



You’ve seen Graphs already

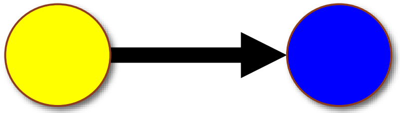
The linked-list Labyrinth assignment is a graph!



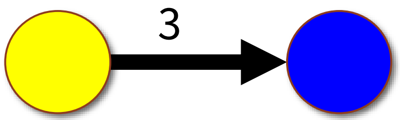
Graph Terminology

Graphs: basic types

- Directedness:
  - A graph may be **directed**—an edge from A to B only allow you to go from A to B, not B to A,
  - or **undirected**—an edge between A and B allows travel in both directions, and/or "direction" doesn't really apply.



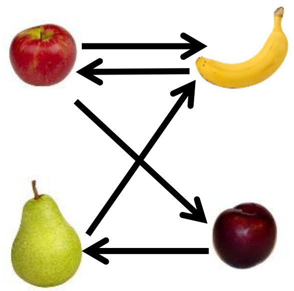
- Weights:
  - A graph may be **weighted**—an edge from A to B has a number representing a length, cost, bandwidth, or strength of that connection)
  - or **unweighted**—all edges are equal.



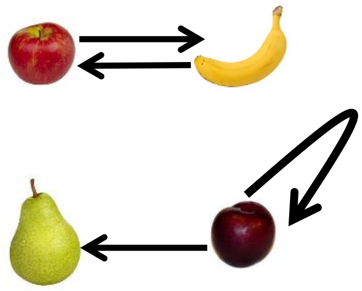
Graphs

All of the following are valid graphs:

A graph could be a single node



An example of a directed graph with 4 nodes



Graphs don't have to be connected (notice this one has two separated parts)

Graph terminology: Paths

path: A path from vertex *a* to *b* is a sequence of edges that can be followed starting from *a* to reach *b*.

neighbor or **adjacent**: Two vertices connected directly by an edge.

reachable: Vertex *a* is *reachable* from *b* if a path exists from *a* to *b*.

connected: A graph is *connected* if every vertex is reachable from every other.

cycle: A path that begins and ends at the same node.

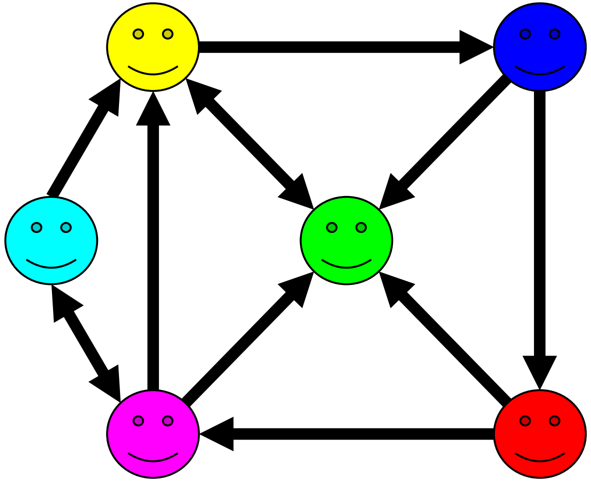
Representing Graphs

WAYS WE COULD IMPLEMENT A GRAPH CLASS

Adjacency Matrix

Representing Graphs: Adjacency matrix

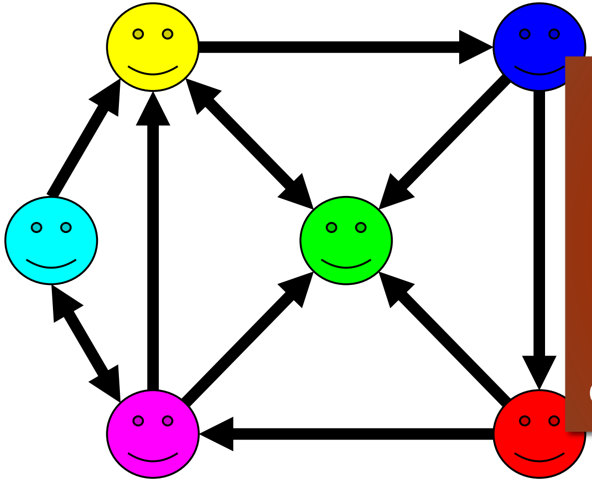
We can represent a graph as a **`Grid<bool>`** (unweighted)



	0	1	1	0	0	0
	0	0	0	1	1	0
	1	1	0	1	0	0
	0	1	0	0	0	0
	0	0	0	1	0	1
	0	0	1	1	0	0

Representing Graphs: Adjacency matrix

We can represent a graph as a **`Grid<bool>`** (unweighted)



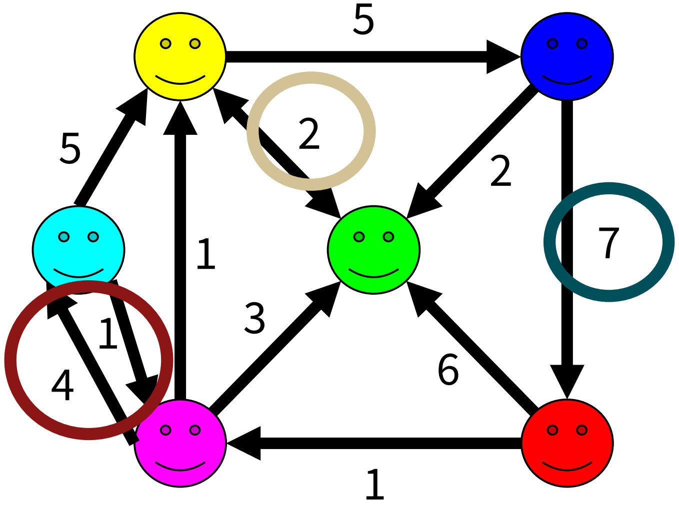
	0	1	1	0	0	0
	0	0	0	1	1	0
	1	1	0	1	0	0
	0	1	0	0	0	0
	0	0	0	1	0	1
	0	0	1	1	0	0

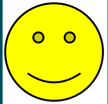
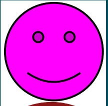
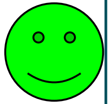
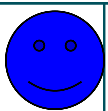
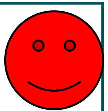
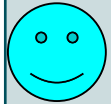
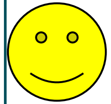
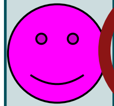
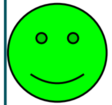
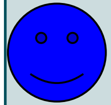
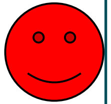
Your Turn: what aspect of the picture does this 0/false correspond to?

Your Turn: which edge in the picture does this 1/true correspond to?

Representing Graphs: Adjacency matrix

We can represent a graph as a **`Grid<int>`** (weighted)

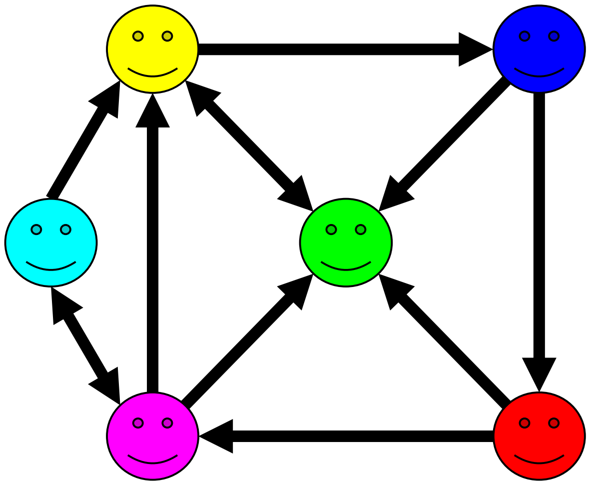


						
	0	5	1	0	0	0
	0	0	0	2	5	0
	4	1	0	3	0	0
	0	2	0	0	0	0
	0	0	0	2	0	7
	0	0	1	6	0	0

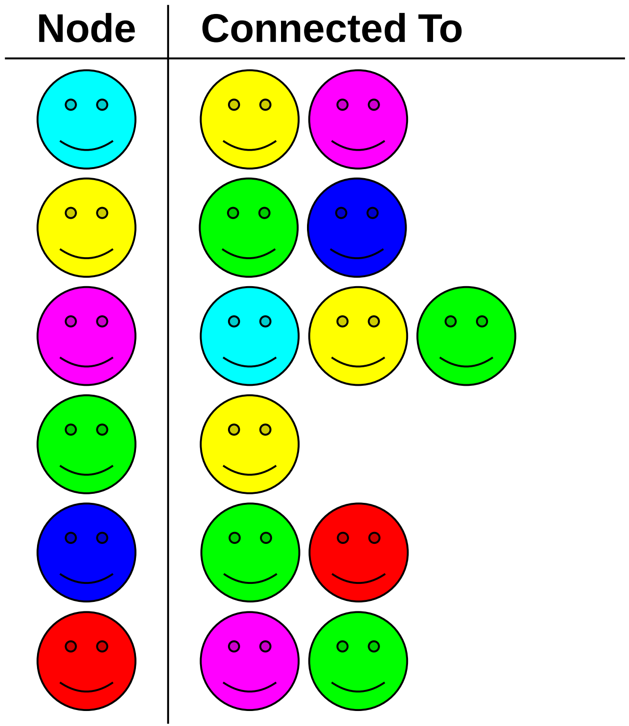
Adjacency List

Representing Graphs: Adjacency list

We can represent a graph as a map from nodes to the set of nodes each node is connected to.

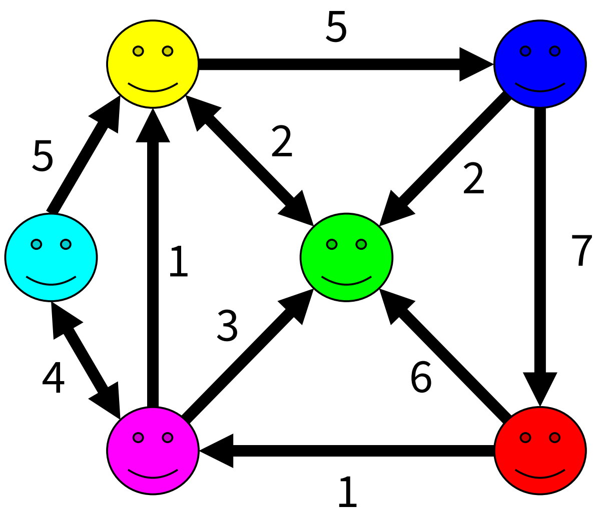


```
Map<Node*, Set<Node*>>
```

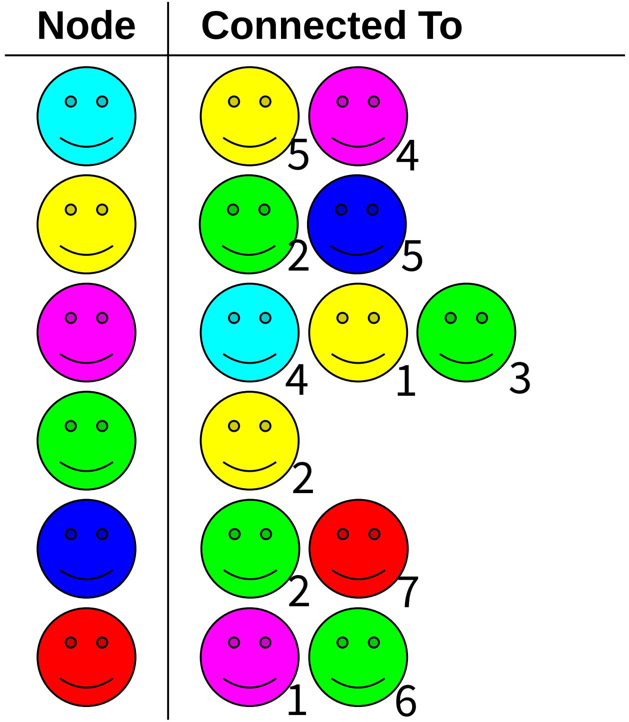


Representing Graphs: Adjacency list

We can represent a graph as a map from nodes to the set of nodes each node is connected to.



`Map<Node*, Set<Edge*>>`



Graph representations

We just saw:

- Adjacency matrix
  - Bool: unweighted
  - Int: weighted
- Adjacency list
  - Without a weight field: unweighted
  - With a weight field: weighted

- These aren't the only options
  - You saw with Labyrinth homework assignment that linked nodes also work!

Your Turn: choosing an implementation

- Which implementation would you choose for the following circumstances:
  - Nodes = Facebook accounts (about 3 billion of them)
  - Edges = the two accounts are “Friends” with each other
- Answer each of the following:
  - Should your graph be weighted or unweighted?
  - Should your graph be directed or undirected?
  - Should you use Adjacency Matrix or Adjacency List? (or something else)
  - And explain (to your neighbors) why ☺

- Now repeat the exercise for Instagram
  - Nodes again represent accounts, and edges capture “following”

Graph representation efficiency considerations

- Adjacency matrix implemented as an NxN `Grid<bool>` takes up how much space in memory?
  - $O(N^2)$
- Approximately what percentage of all N2 entries in the Grid are 0/false?
  - Pretty close to 100%!
- This is a common phenomenon called a "sparse matrix"—there's no specific numeric cutoff to count as one, it just means generally a matrix with few entries that are actually "used" in a useful way
- As an engineer, it helps to know when designing your setup whether a given NxN matrix is expected to be sparse or not, and if so, you can make design decisions to reduce waste!

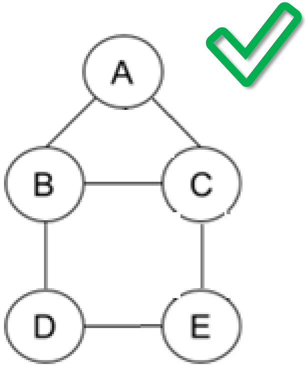
Quick Sampling of Classic Graph Theory Problems

Hamiltonian Cycle

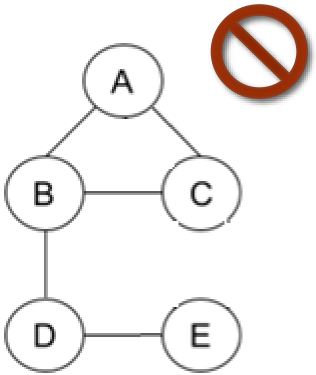
SADLY, HAS NOTHING TO DO WITH THE MUSICAL HAMILTON. NAMED AFTER SIR WILLIAM ROWAN HAMILTON.

Hamiltonian Cycle

- A **Hamiltonian Cycle** is a path that starts and ends at the same node, and visits every node exactly once *(except that start/end node, which is of course visited twice)*.



There are several different ways to do a Hamiltonian Cycle in this graph. (ex: A, B, D, E, C, A)



It's not possible to do a Hamiltonian Cycle in this graph.

Hamiltonian Cycle

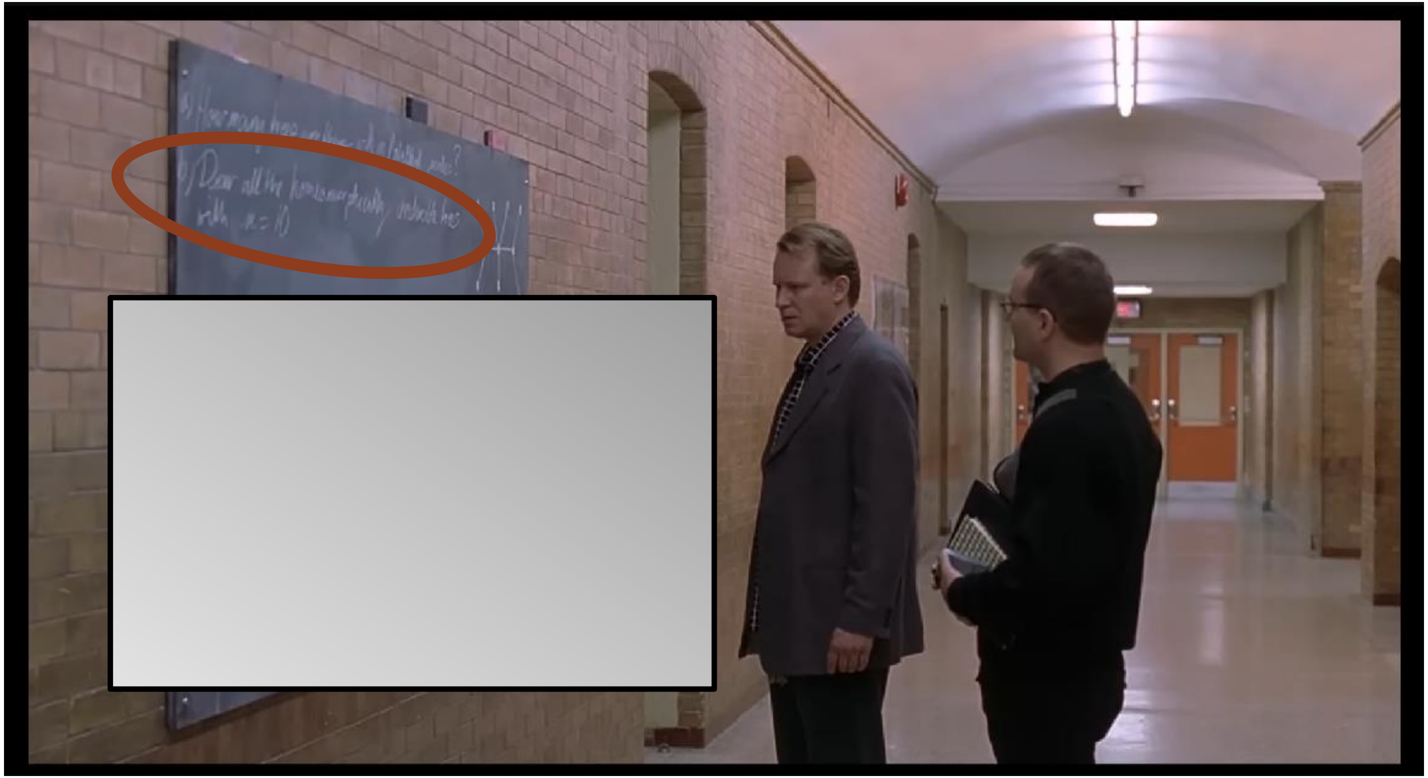
- A **Hamiltonian Cycle** is a path that starts and ends at the same node, and visits every node exactly once *(except that start/end node, which is of course visited twice)*.

- The concept of a Hamiltonian Cycle is pretty simple!
- And yet, there is no known algorithm for detecting if a graph contains a Hamiltonian Cycle that is faster than $O(2^N)$ ☹ ☹
  - Using CS106B skills, it's not very hard to write a recursive depth-first search / backtracking function to find a Hamiltonian Path in a graph. But the function unfortunately will be very slow.

Problem from “Good Will Hunting” Movie

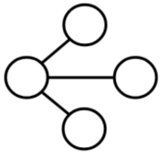
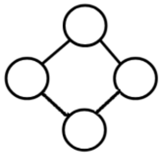
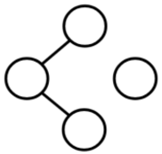
OR, HOW HOLLYWOOD THINKS YOU PROVE YOU ARE A MATH SAVANT

“Draw all the homeomorphically irreducible trees with n=10.”

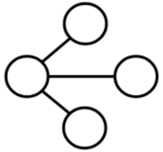
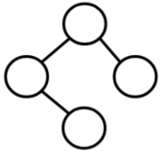
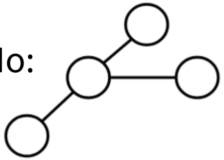


"Draw all the homeomorphically irreducible trees with n=10."

- **"n = 10"** means it has **10 nodes**
- **"trees"** means an **undirected graph** with no "loops" (no way to go from a node and get back to itself without reusing edges), and there aren't any parts of the graph that are totally disconnected from the rest

Yes:
(for n=4)

No:

No:


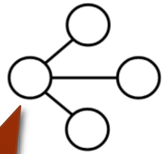
- **"homeomorphically irreducible"** means that for this problem, nodes that lie between exactly 2 other nodes are useless in terms of branching structure—they in effect just act as a blip on a longer edge—and are therefore banned. Also we ignore superficial changes in the drawing.

Yes:
(for n=4)

No:

No:

Legal, but same underlying structure as the first, so it doesn't count as a new one

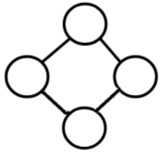
“Draw all the homeomorphically irreducible trees with n=10.”

- **“n = 10”** means it has **10 nodes**
- **“trees”** means an **undirected graph** with no “loops” (no wa… a node and get back to itself without reusing edges), and … any parts of the graph that are totally disconnected from th…

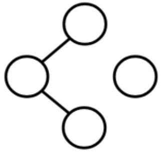
Yes: (for n=4)



No:



No:

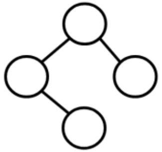


How many can you find?

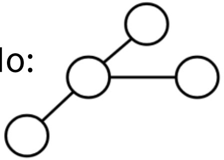
- **“…omeo…rphically irreducible”** means that for this problem, nodes … actly 2 other nodes are useless in terms of tree … effect just act as a blip on a longer edge—and are … lso we ignore superficial changes in the drawing.

Extra challenge: can we predict the number of such trees for arbitrary n?

No:



No:



Legal, but same underlying structure as the first, so it doesn’t count as a new one

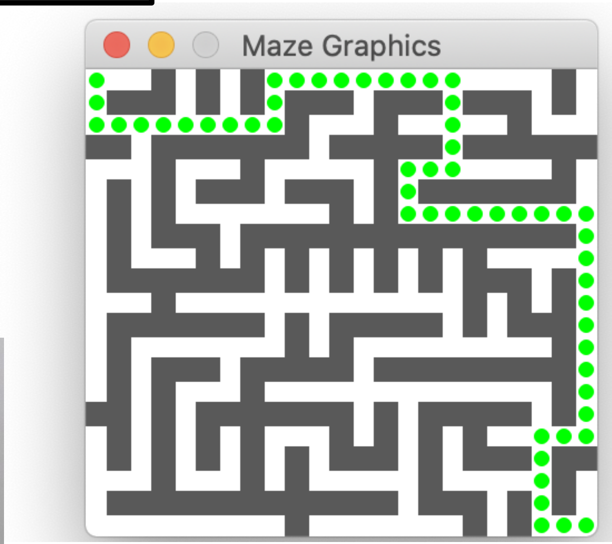
Stanford University

Breadth-First Search

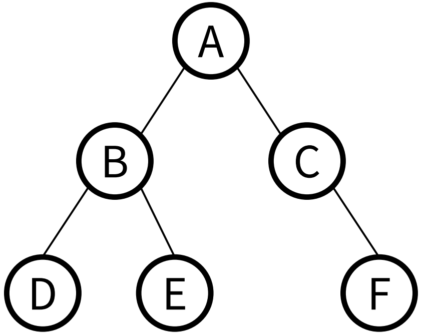
WE'VE SEEN BFS BEFORE THIS QUARTER!

BFS in this class so far

Assignments



Slime Mold



Trees

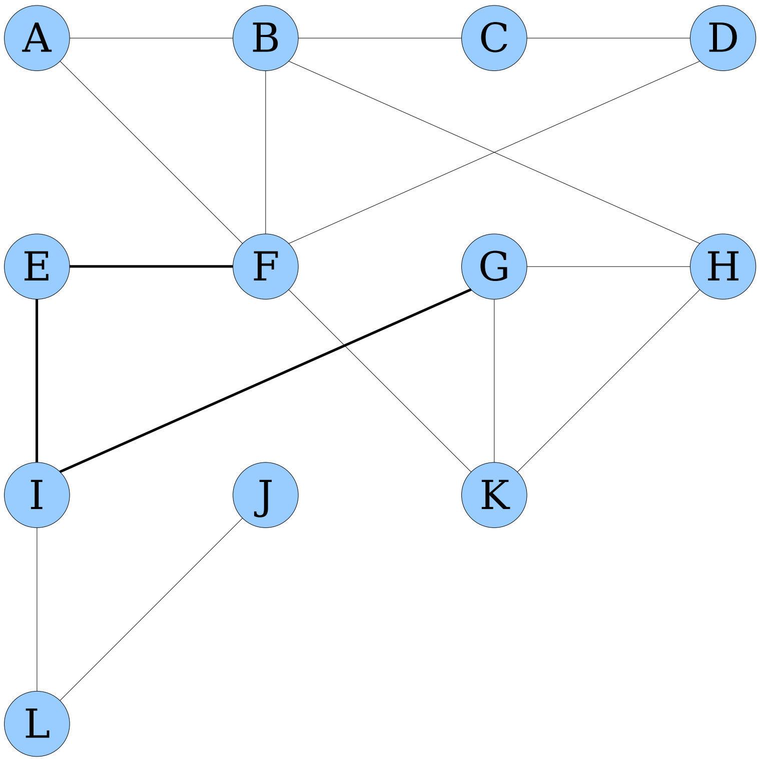
Generic BFS algorithm pseudocode

1. Make an empty queue to store places we want to visit in the future
2. Enqueue the starting location
3. While the queue is not empty (and/or until you reach a desired destination):
   - Dequeue a location
   - Mark that location as visited
   - Enqueue all the neighbors of that location

Breadth-First Search in a Graph

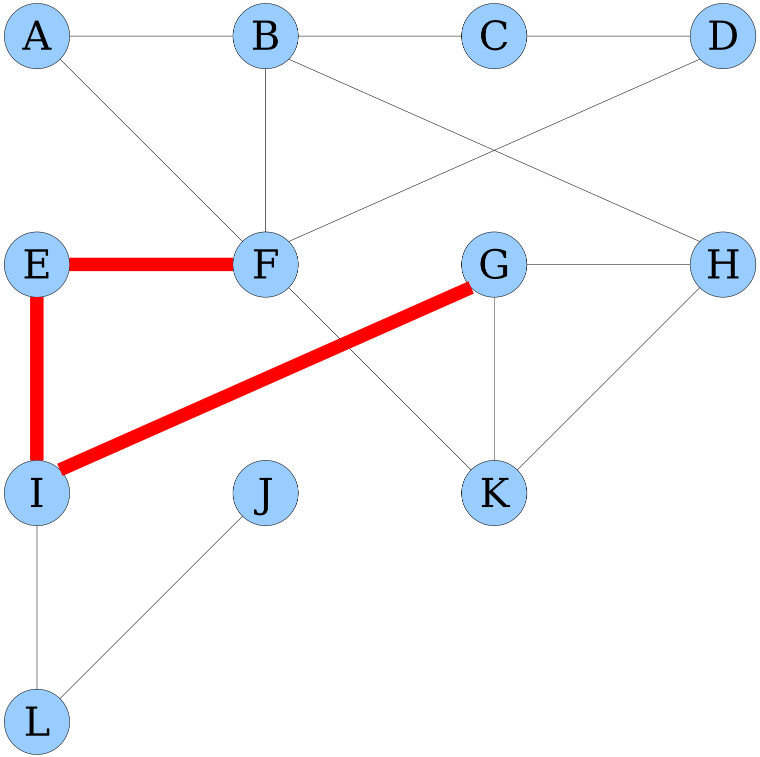
GRAPH ALGORITHMS

Breadth-First Search



BFS is useful for finding the <u>shortest path</u> between two nodes (in an unweighted, or equally-weighted graph).

Breadth-First Search

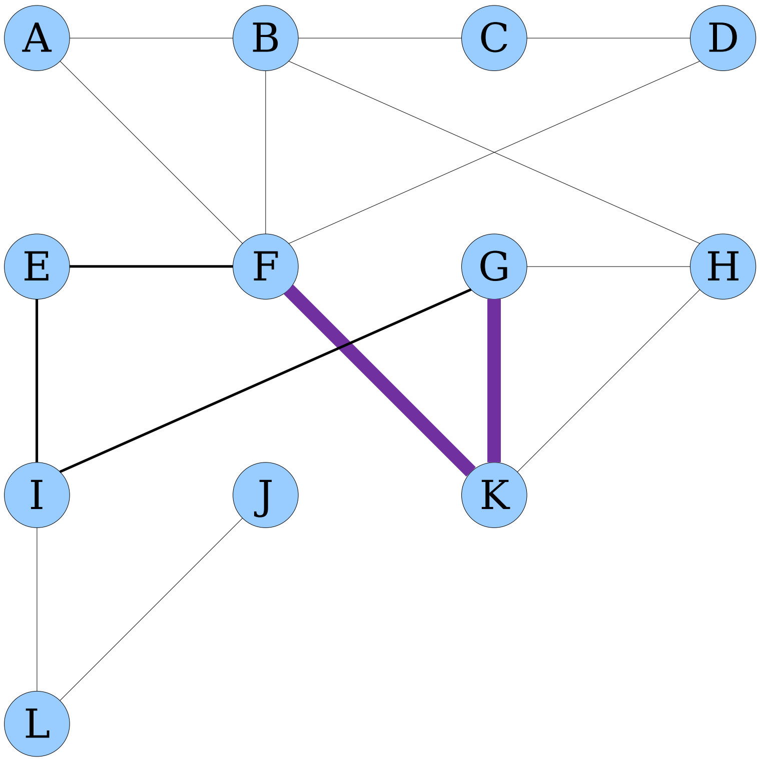


BFS is useful for finding the <u>shortest path</u> between two nodes (in an unweighted, or equally-weighted graph).

Example: What is the shortest way to go from F to G?

One way (not the shortest):
F → E → I → G **3 edges**

Breadth-First Search

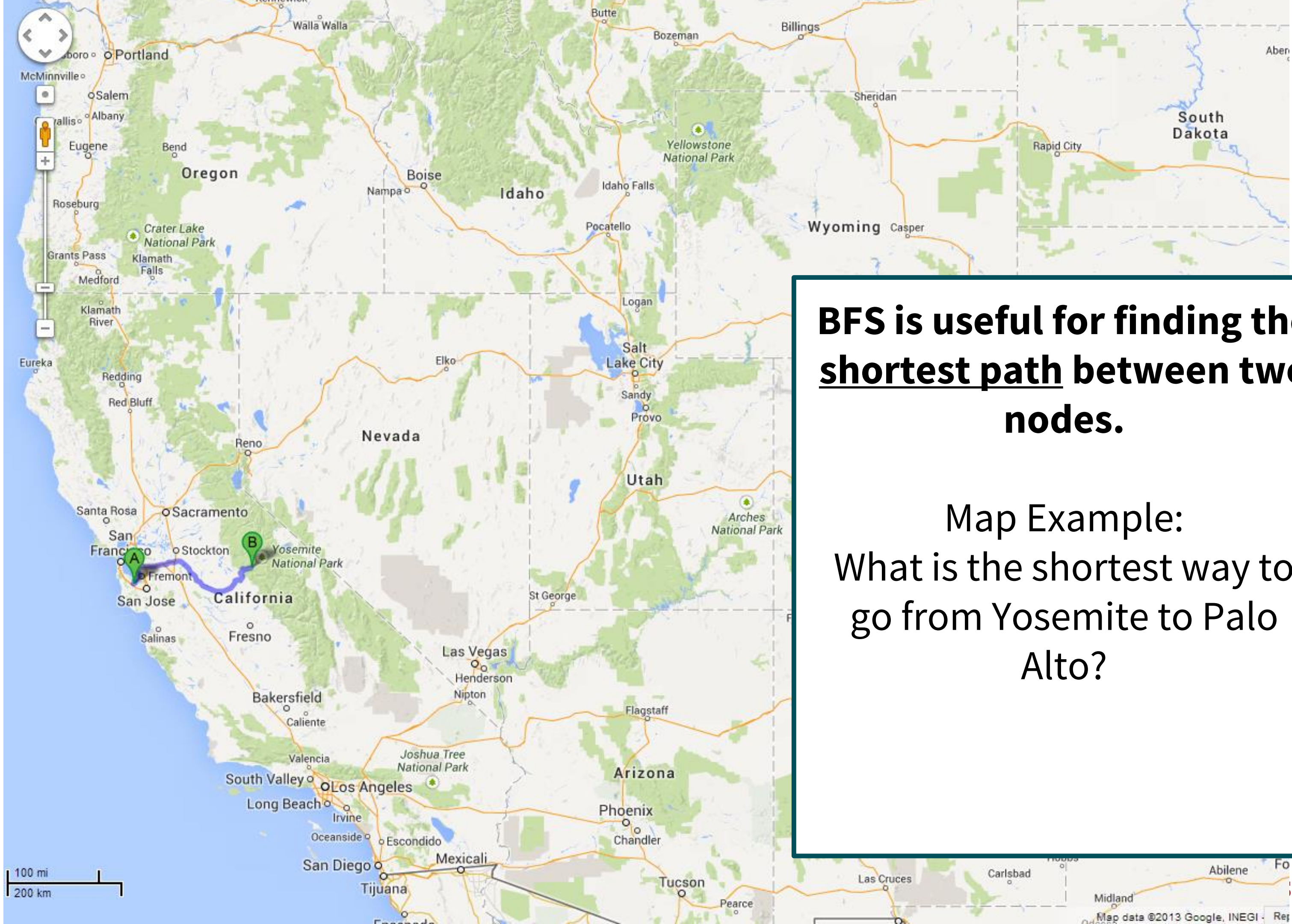


BFS is useful for finding the shortest path between two nodes (in an unweighted, or equally-weighted graph).

Example: What is the shortest way to go from F to G?

One way (not the shortest):
F → E → I → G **3 edges**

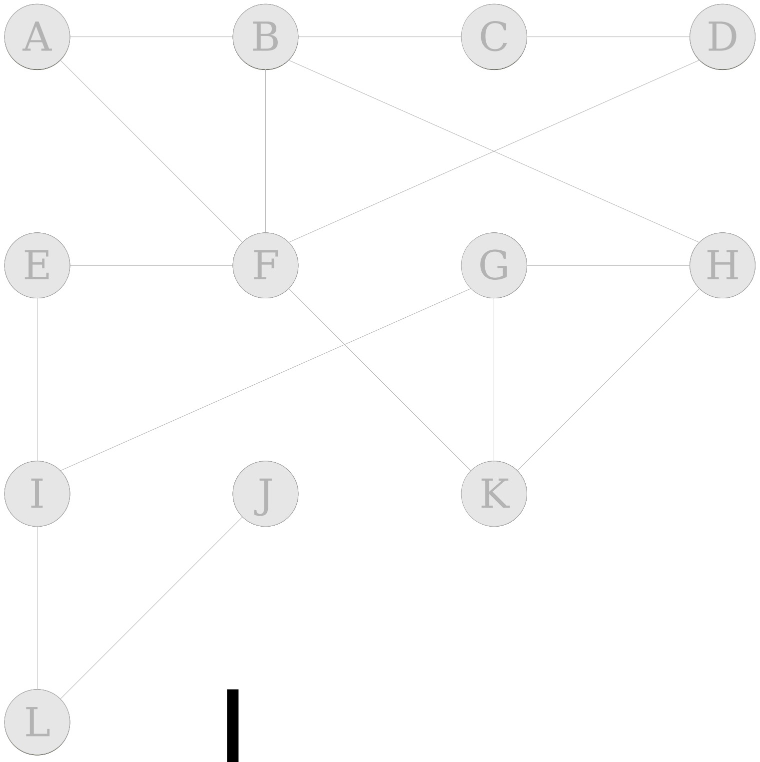
Shortest way:
F → K → G **2 edges**



BFS is useful for finding the <u>shortest path</u> between two nodes.

Map Example:
What is the shortest way to go from Yosemite to Palo Alto?

A BFS algorithm for graphs with a special property…

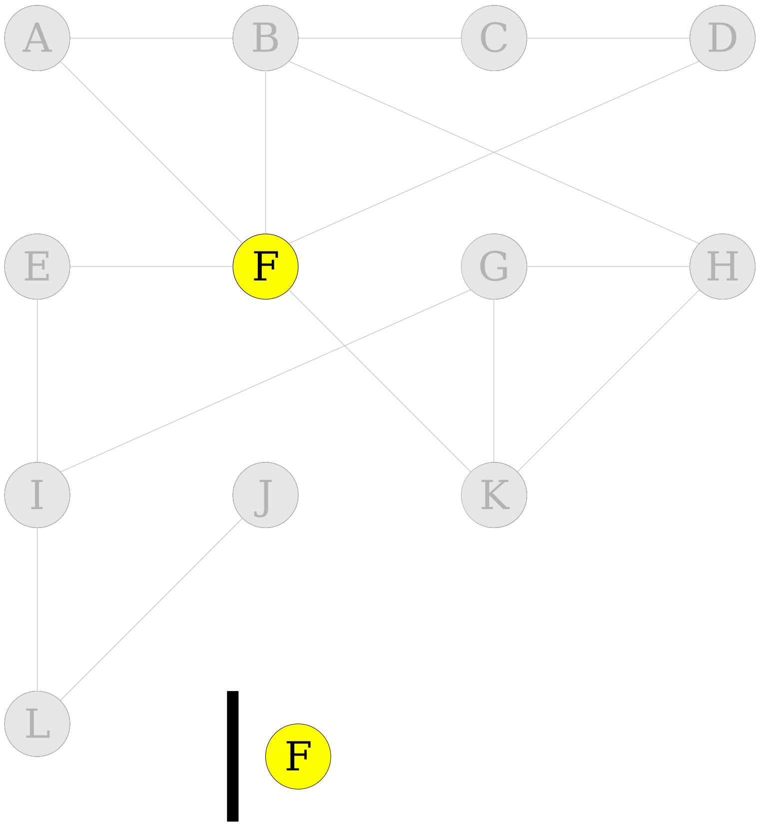


TO START:

(1) Color all nodes GREY to mean UNVISITED

(2) Queue is empty

A BFS algorithm for graphs with a special property…



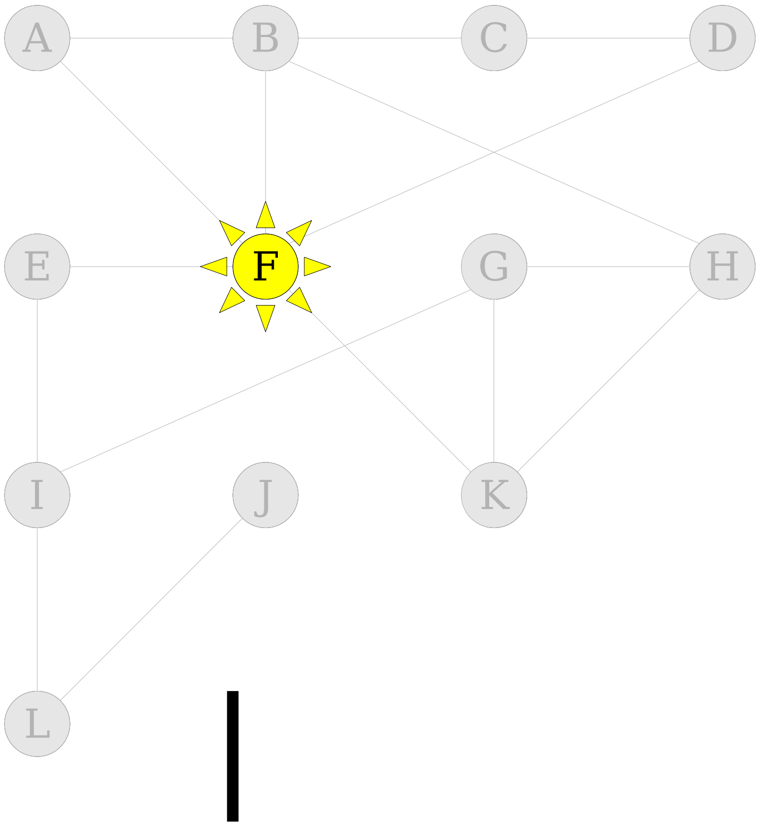
TO START:

(1) Color all nodes GREY to mean UNVISITED

(2) Queue is empty

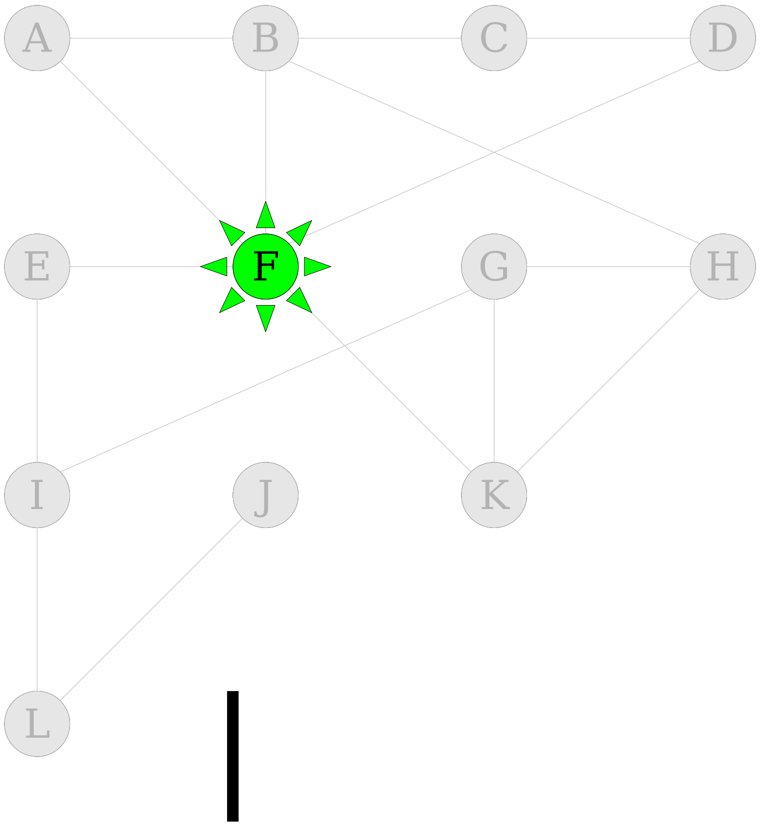
(3) Enqueue the desired **start** node, change its color to mark it VISITED

A BFS algorithm for graphs with a special property…

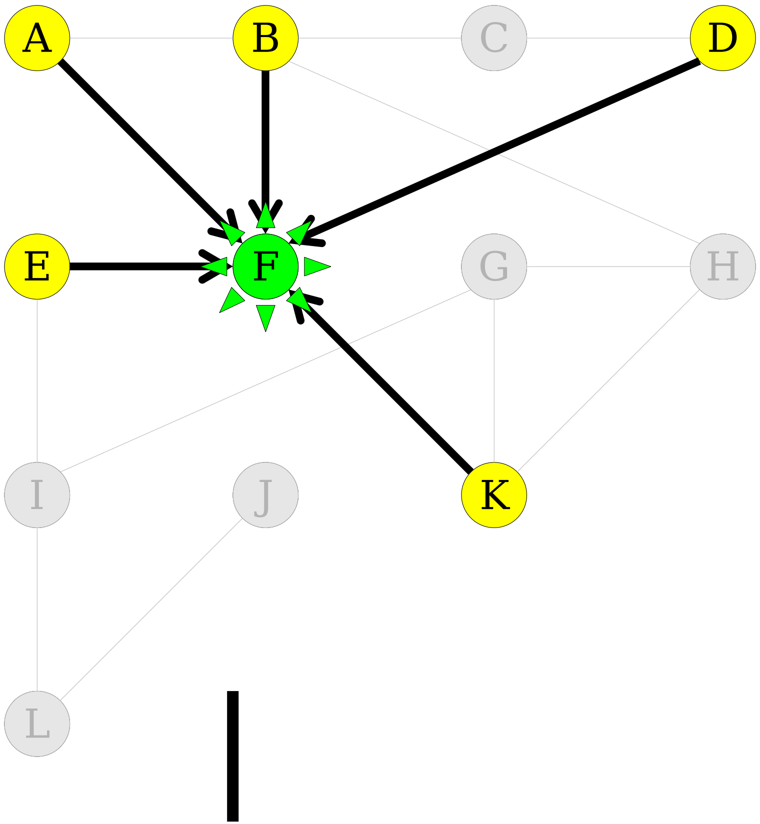


LOOP PROCEDURE:
(1) Dequeue a node
(2) Set current node's UNVISITED neighbors' parent pointers to current node, then enqueue them (and mark them visited when we enqueue)

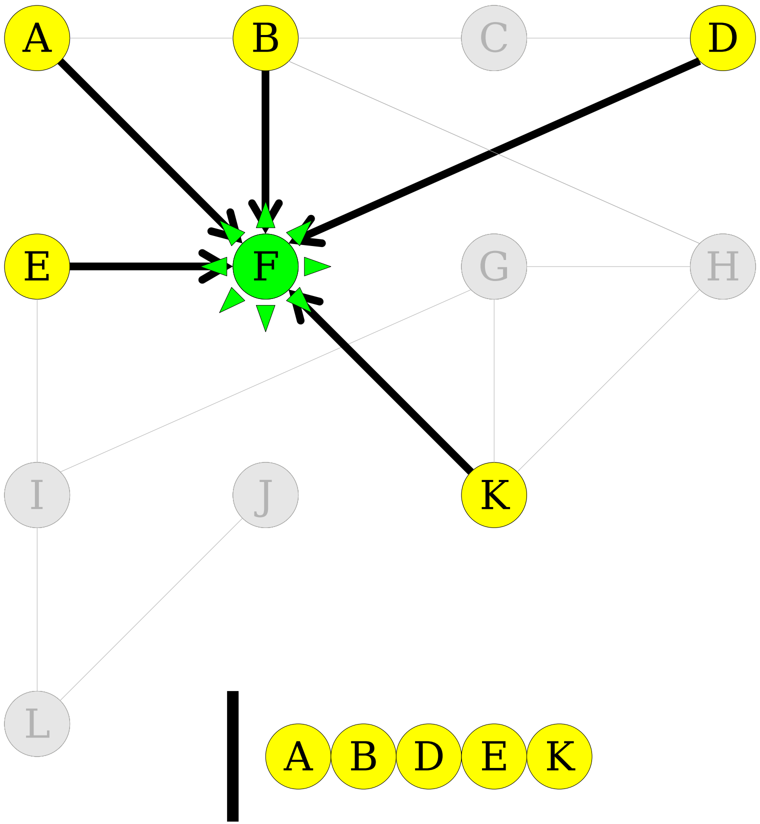
Breadth-First Search



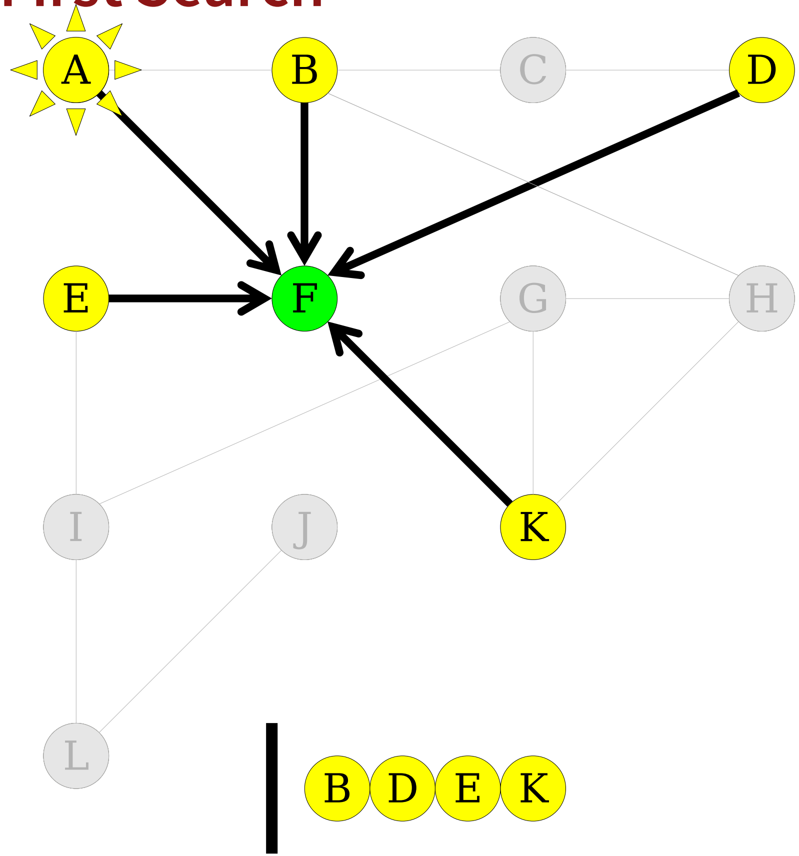
Breadth-First Search



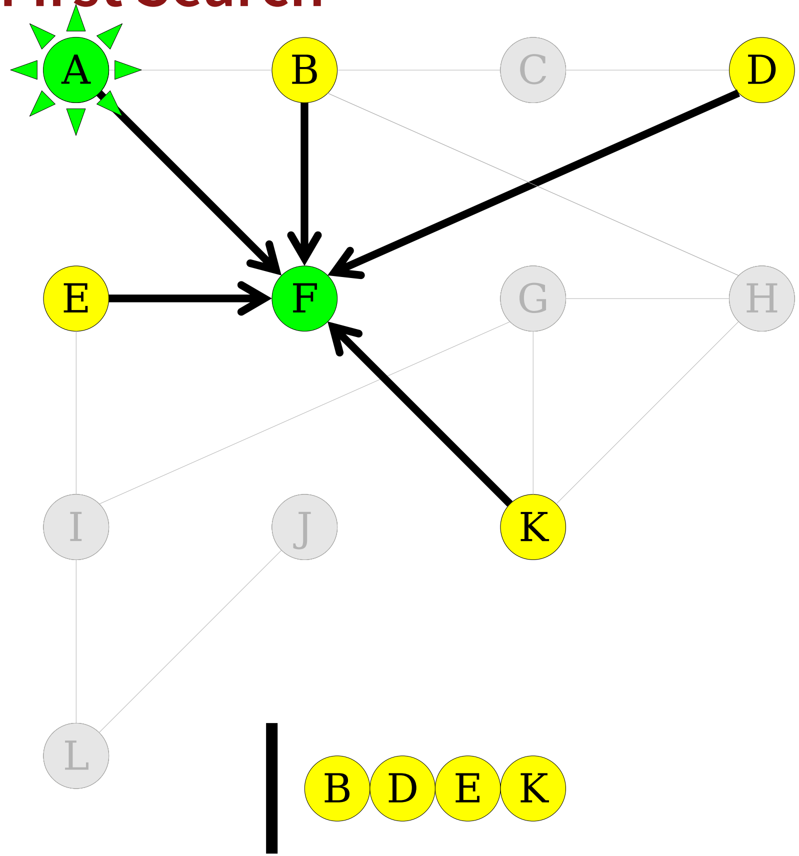
Breadth-First Search



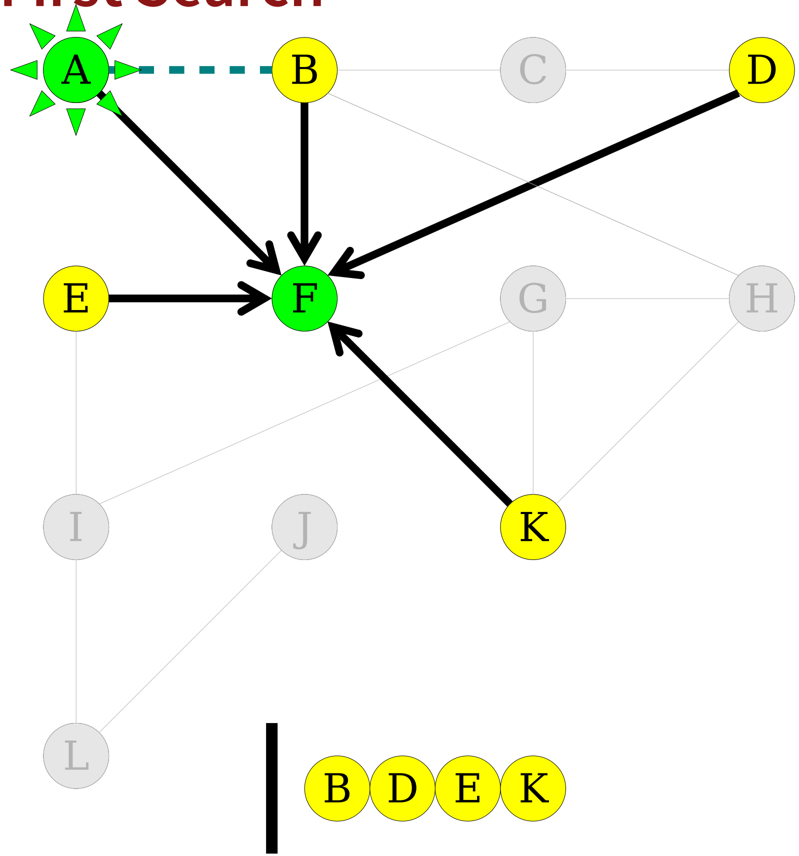
Breadth-First Search



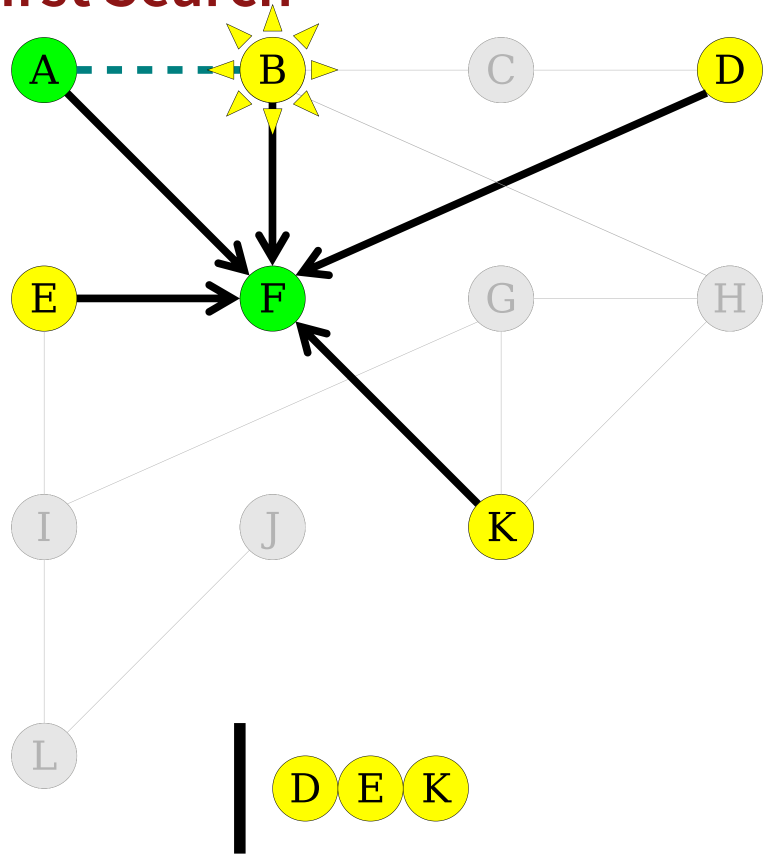
Breadth-First Search



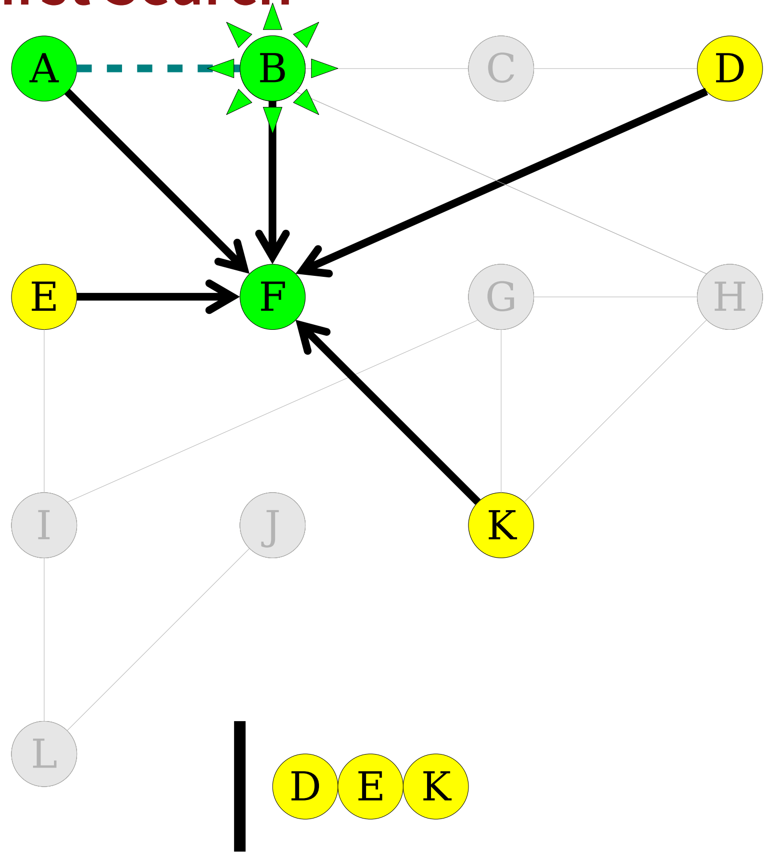
Breadth-First Search



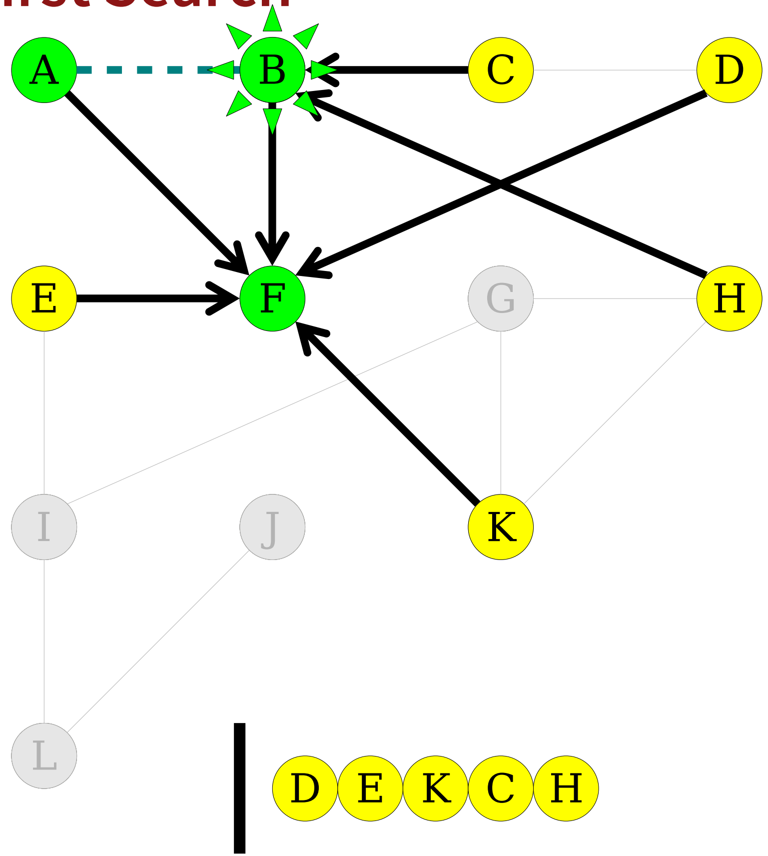
Breadth-First Search



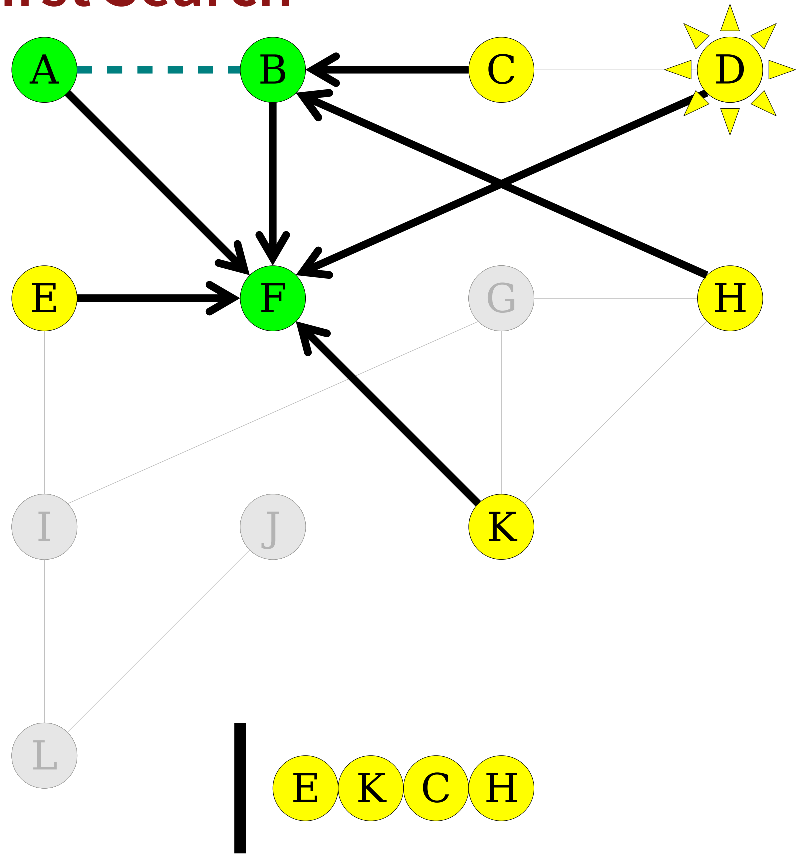
Breadth-First Search



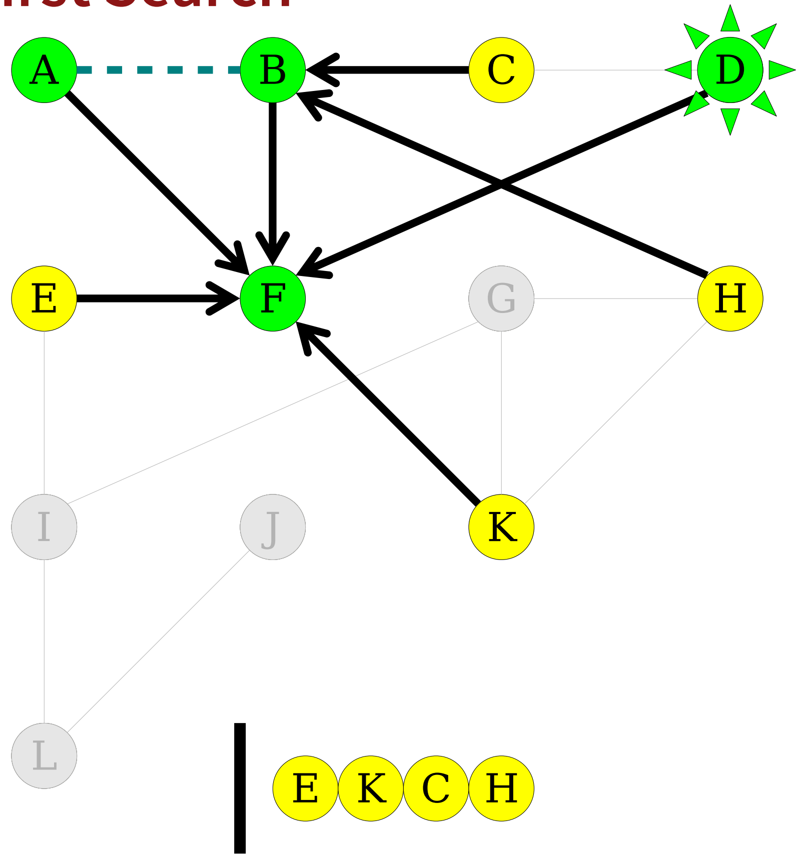
Breadth-First Search



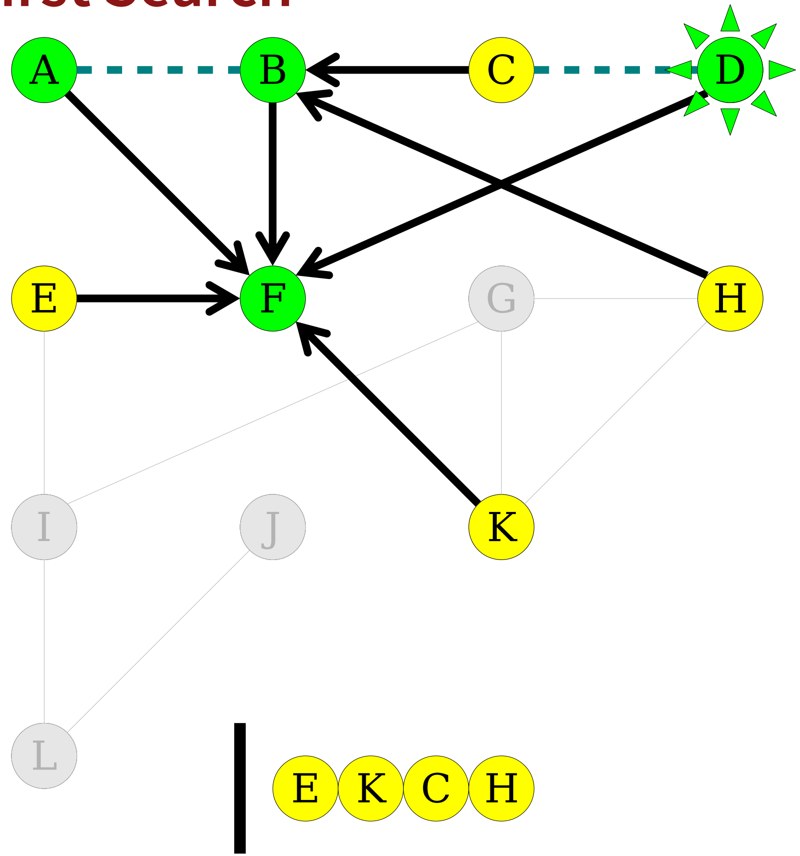
Breadth-First Search



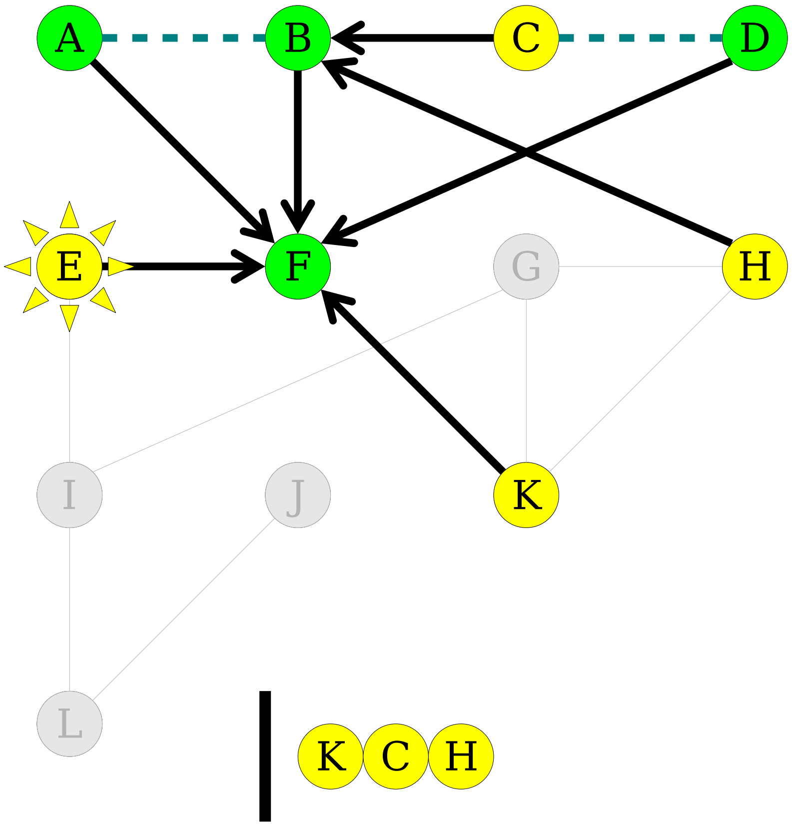
Breadth-First Search



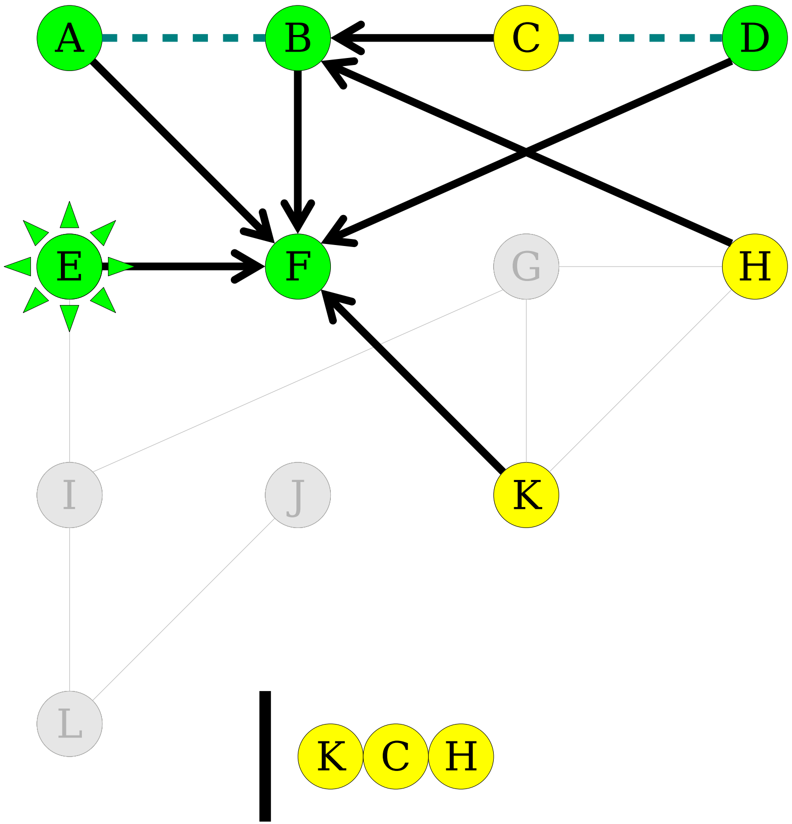
Breadth-First Search



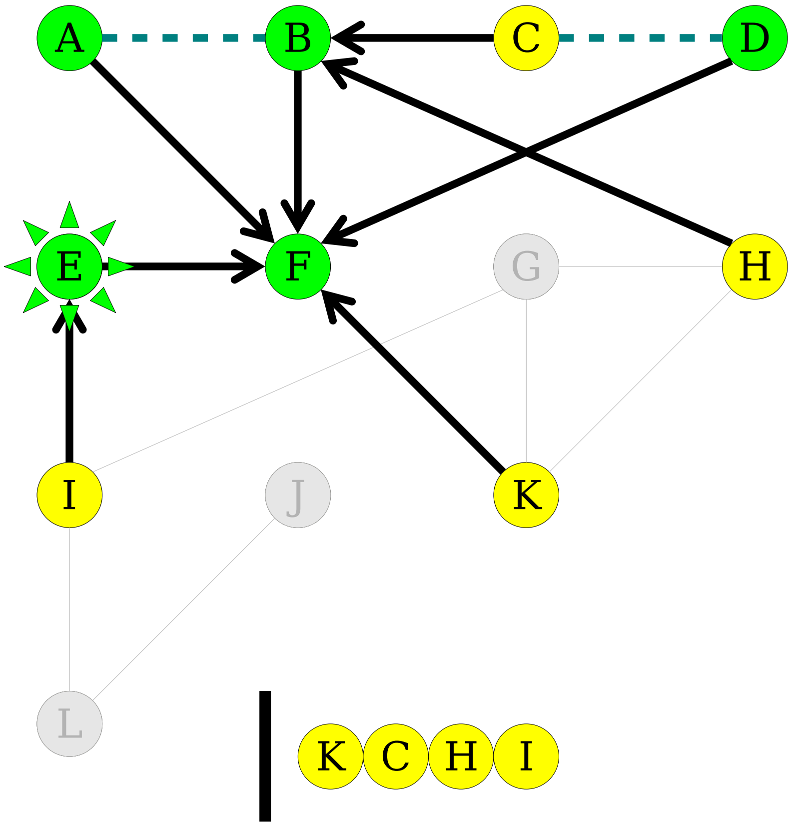
Breadth-First Search



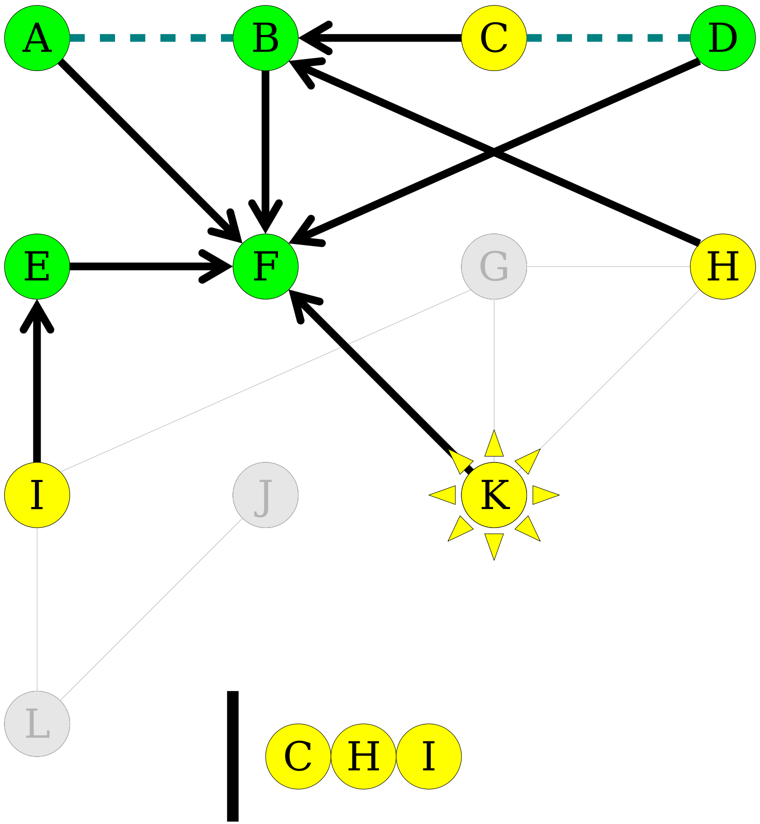
Breadth-First Search



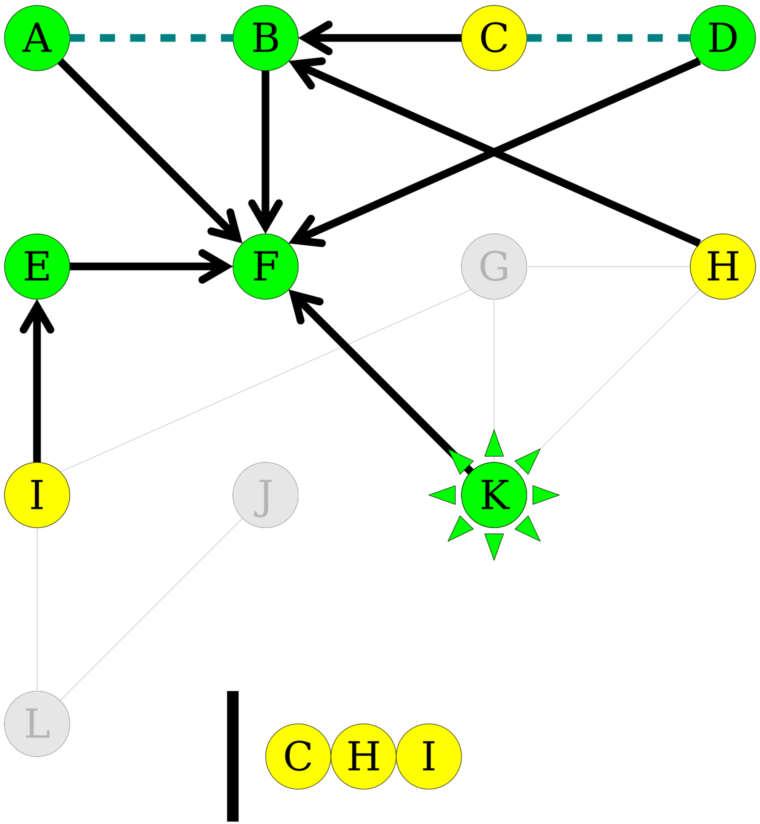
Breadth-First Search



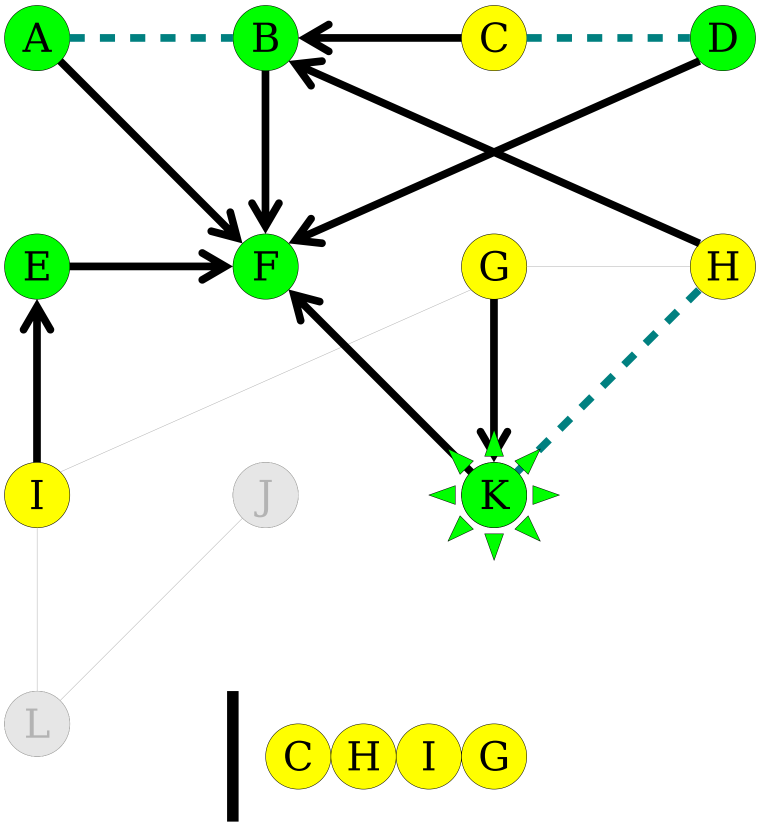
Breadth-First Search



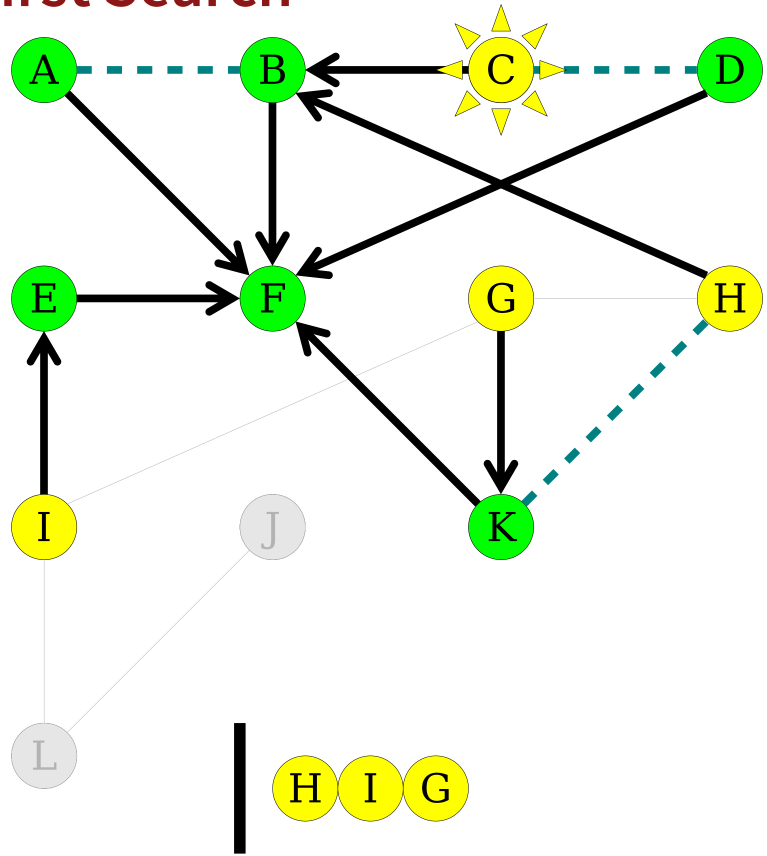
Breadth-First Search



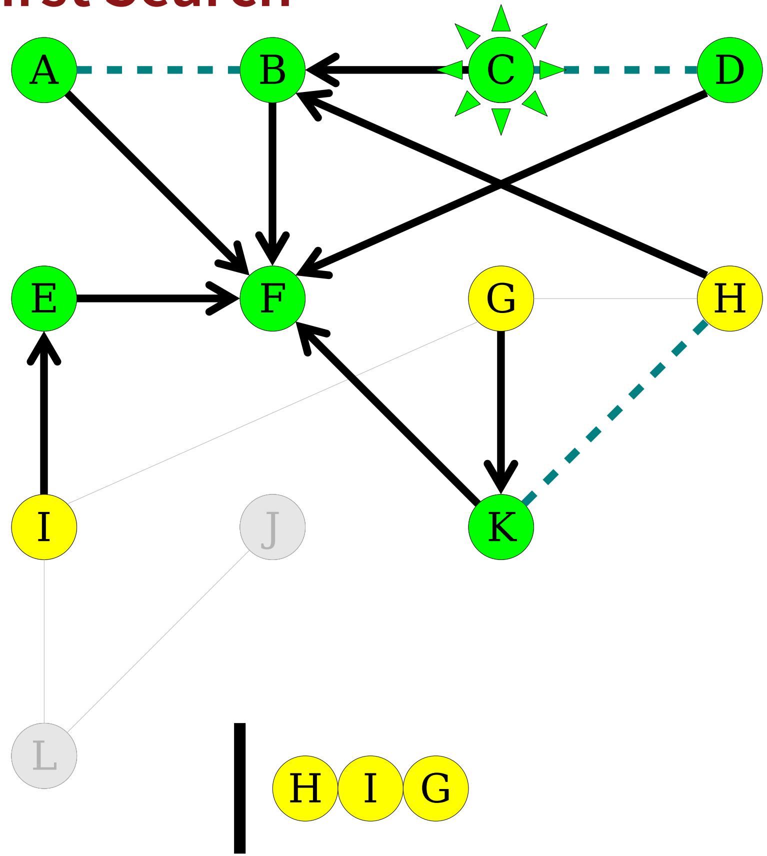
Breadth-First Search



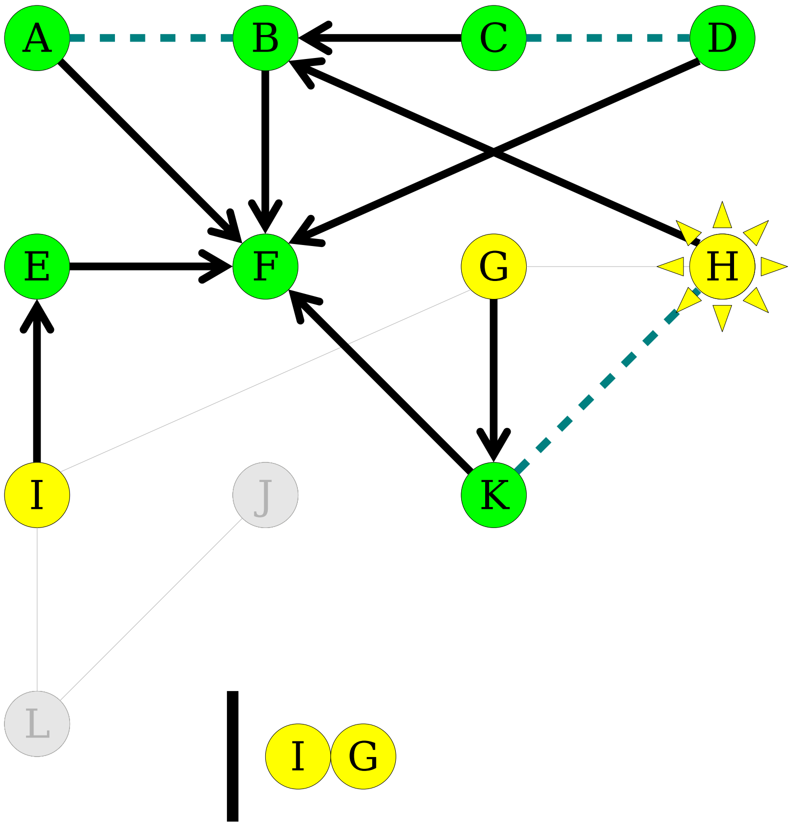
Breadth-First Search



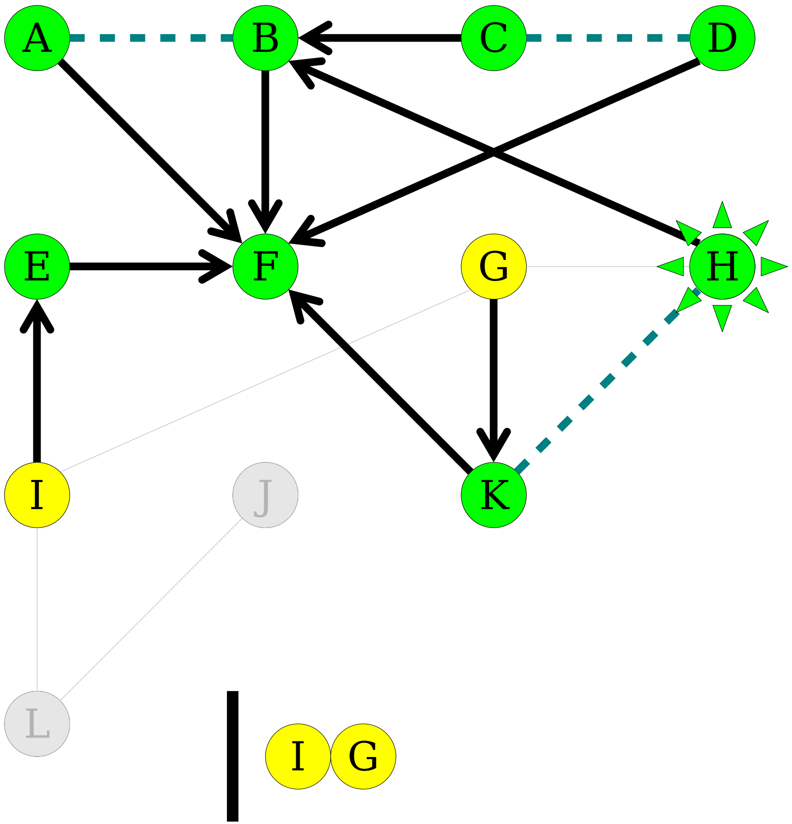
Breadth-First Search



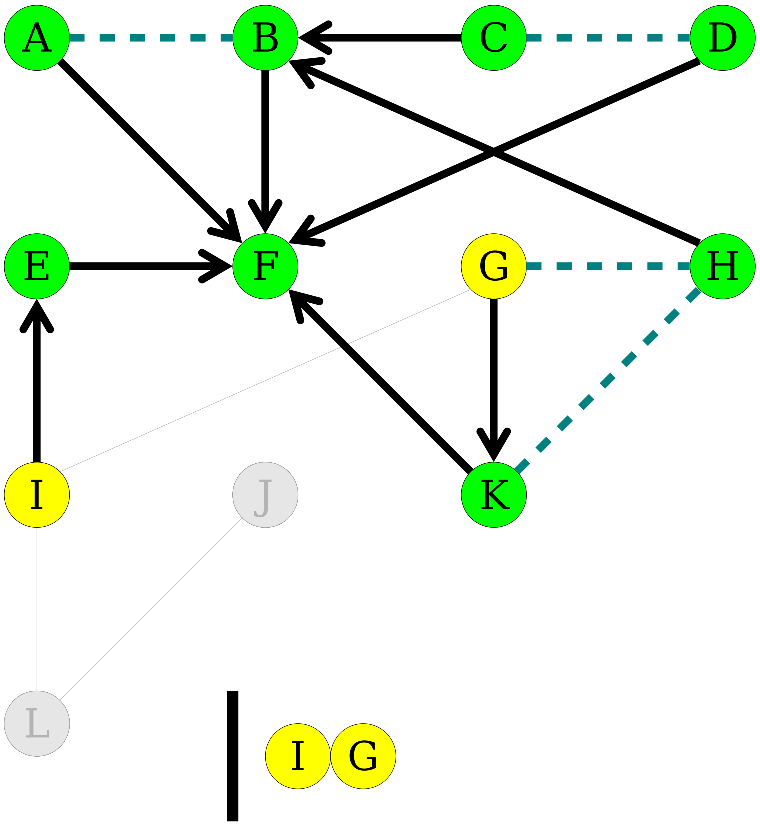
Breadth-First Search



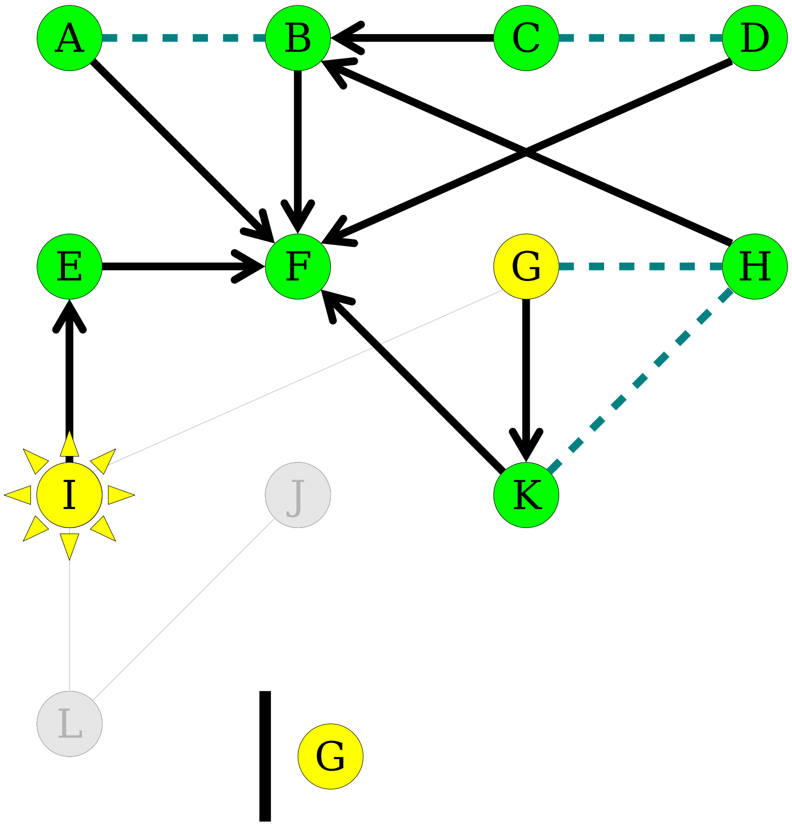
Breadth-First Search



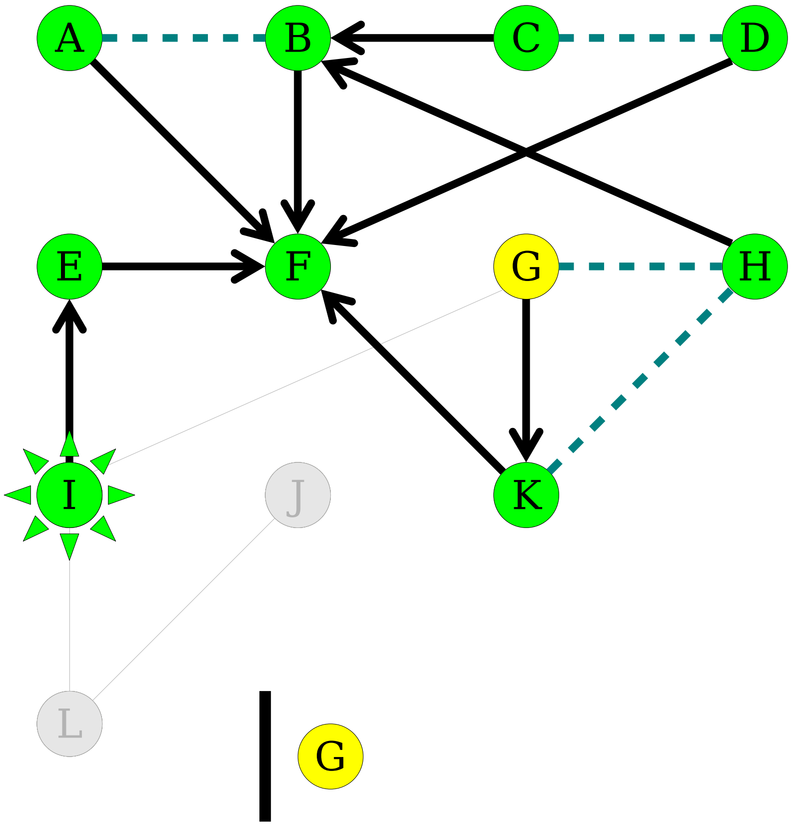
Breadth-First Search



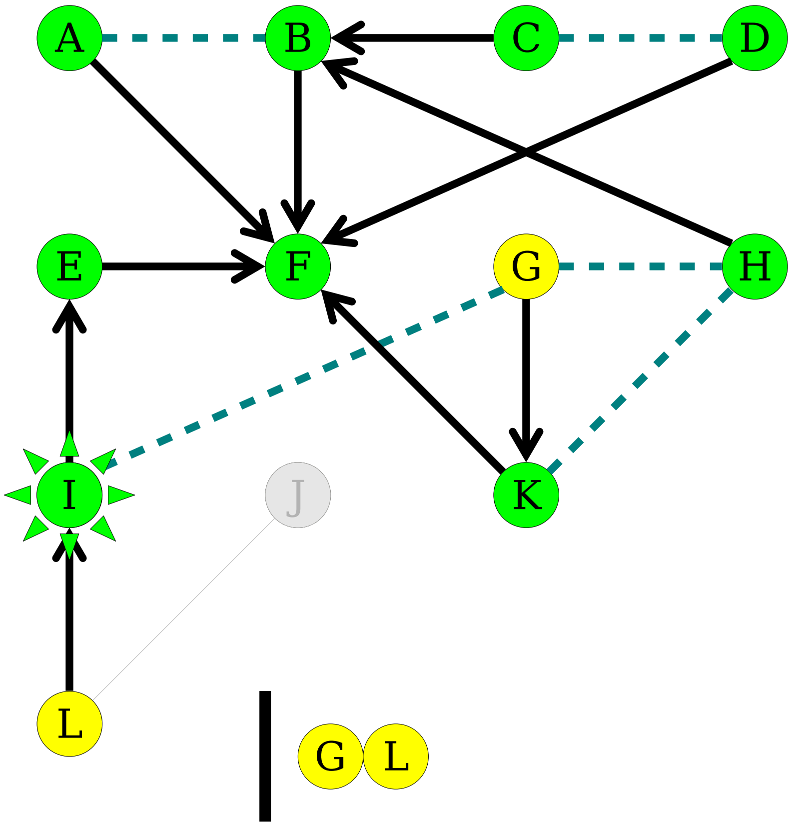
Breadth-First Search



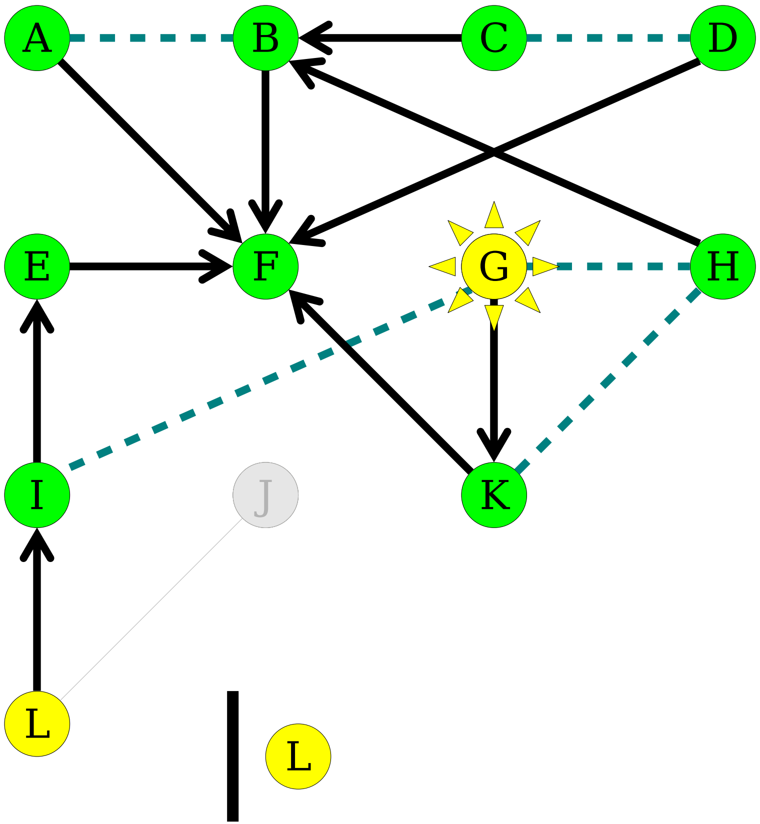
Breadth-First Search



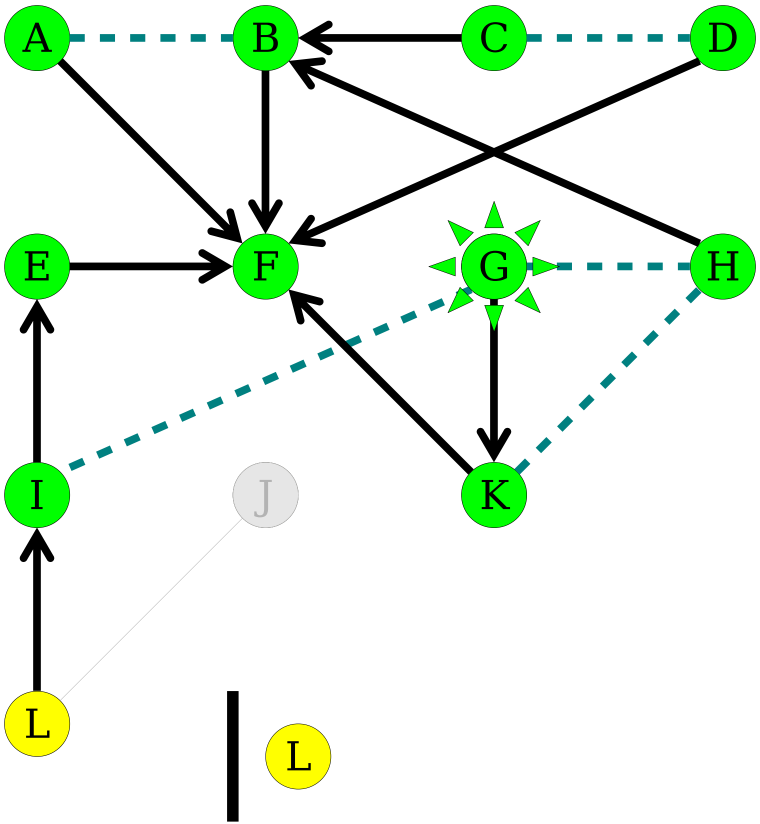
Breadth-First Search



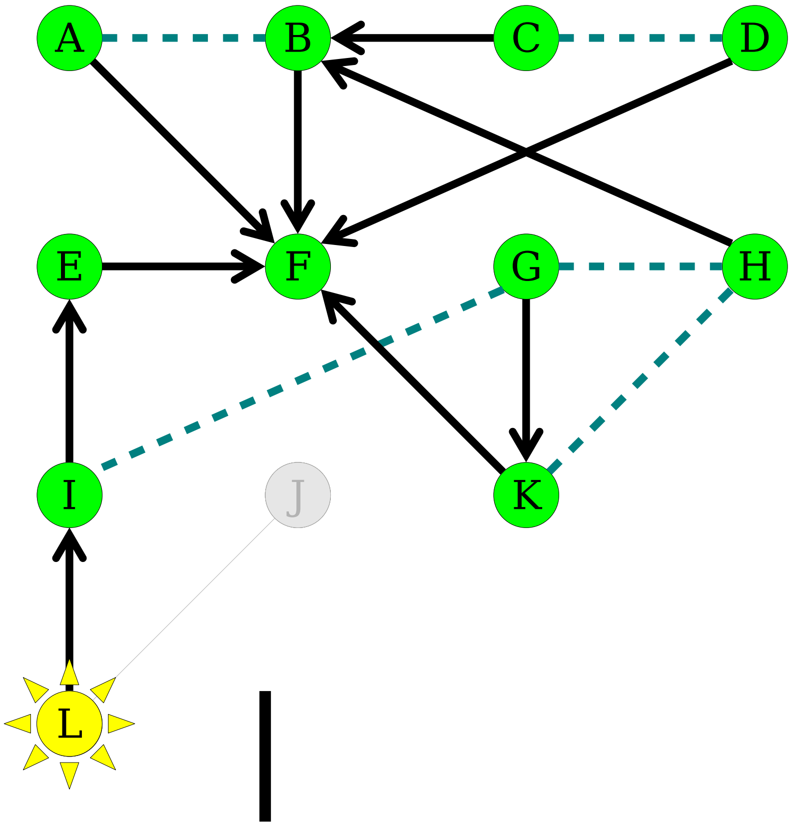
Breadth-First Search



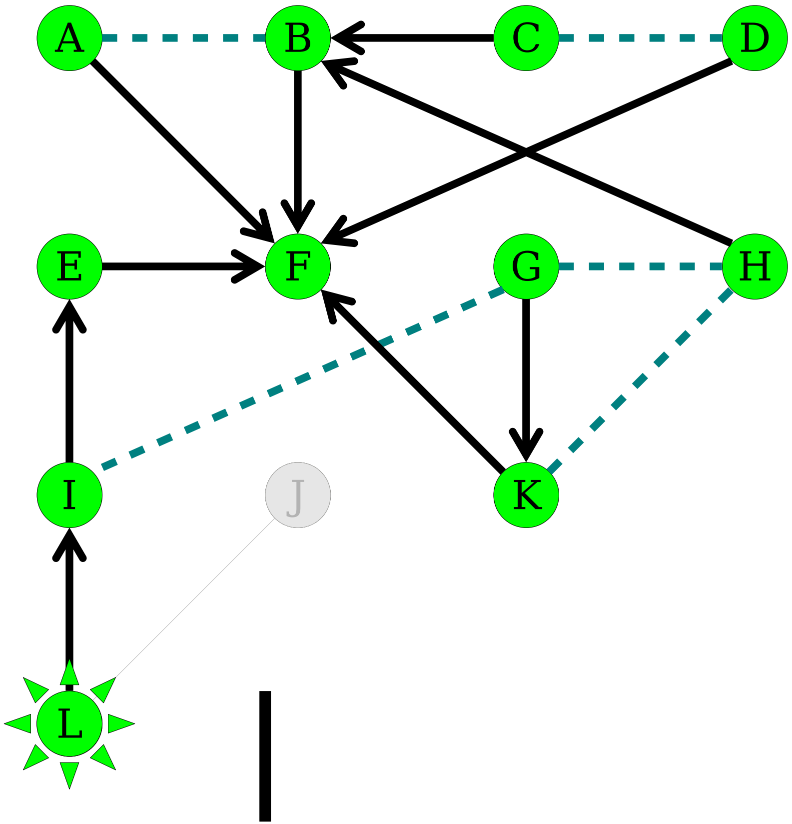
Breadth-First Search



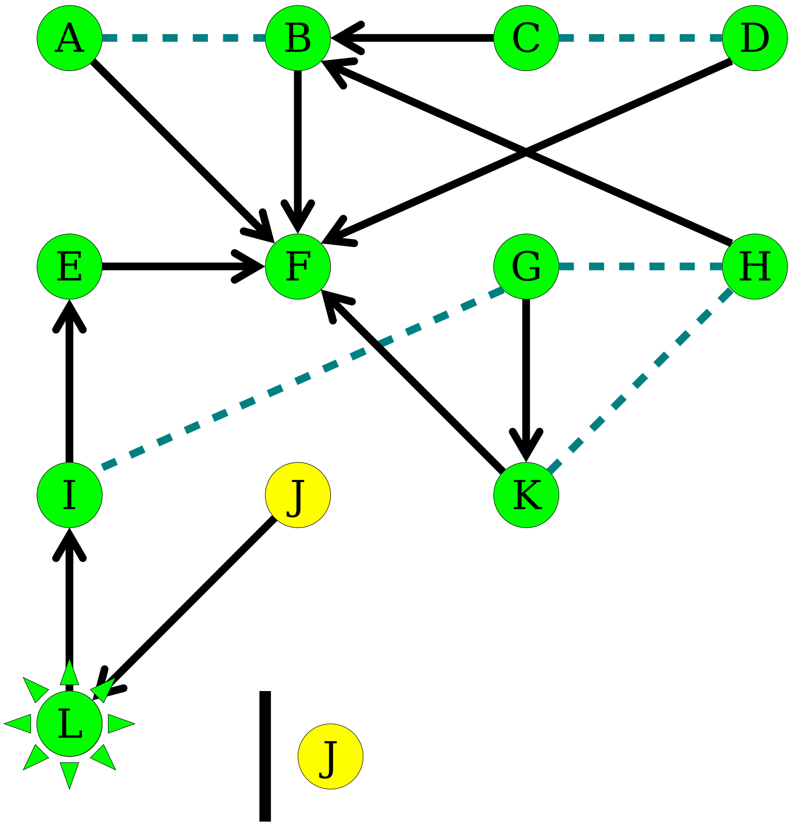
Breadth-First Search



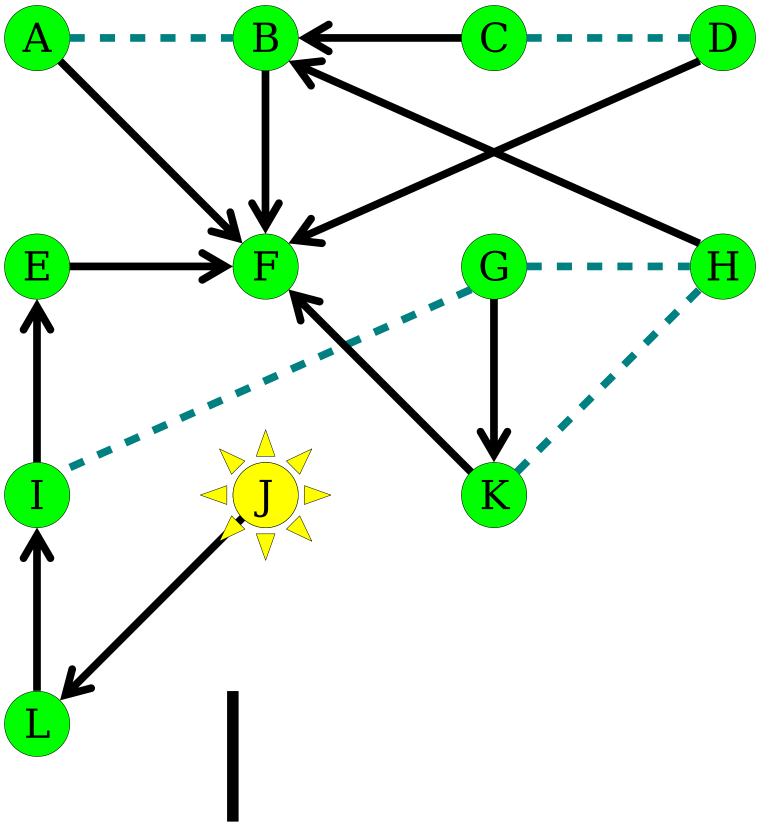
Breadth-First Search



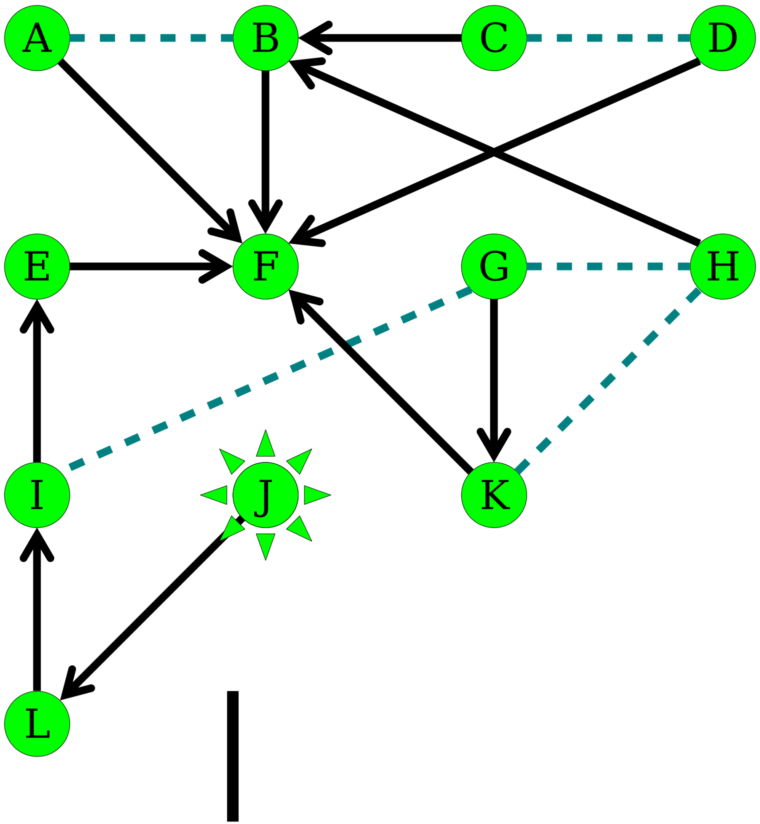
Breadth-First Search



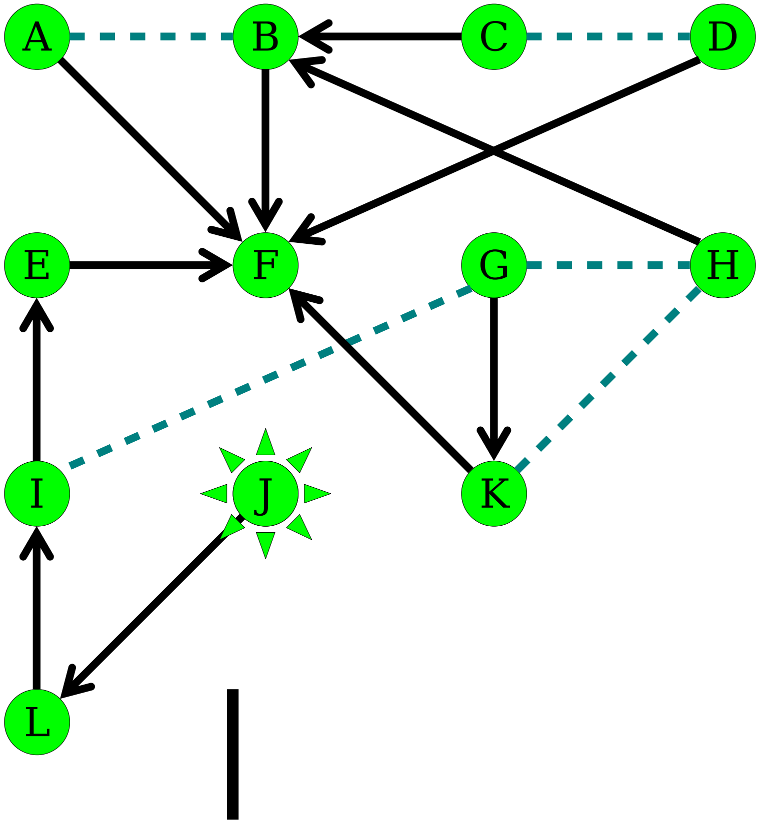
Breadth-First Search



Breadth-First Search



Breadth-First Search



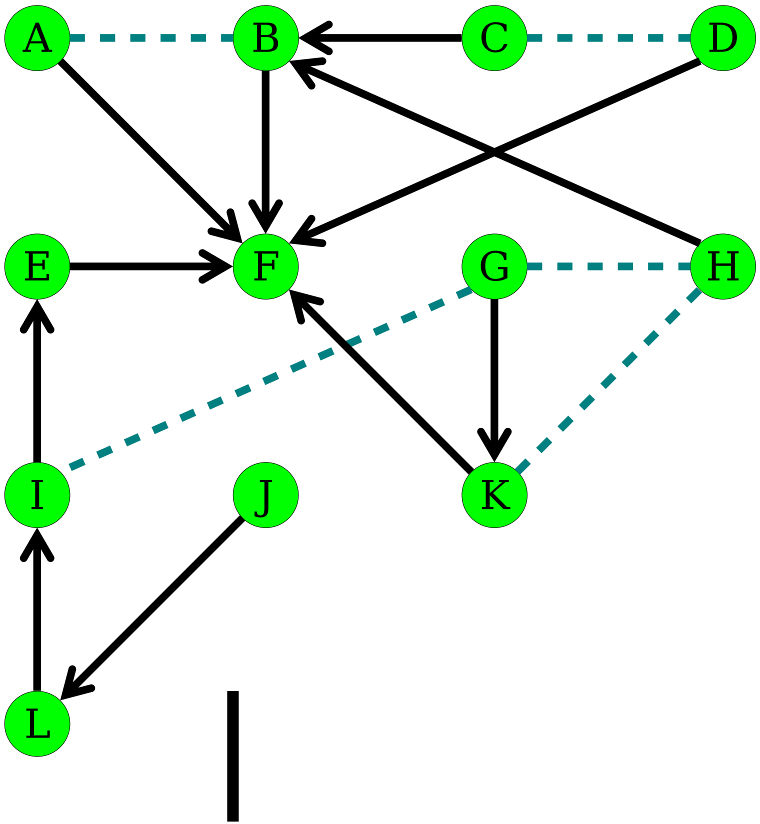
Done!

Now we know that to go from Yoesmite (F) to Palo Alto (J), we should go:

F->E->I->L->J
(4 edges)

(note we follow the parent pointers backwards)

Breadth-First Search



THINGS TO NOTICE:

(1) We used a queue

(2) What's left is a kind of subset of the edges, in the form of 'parent' pointers

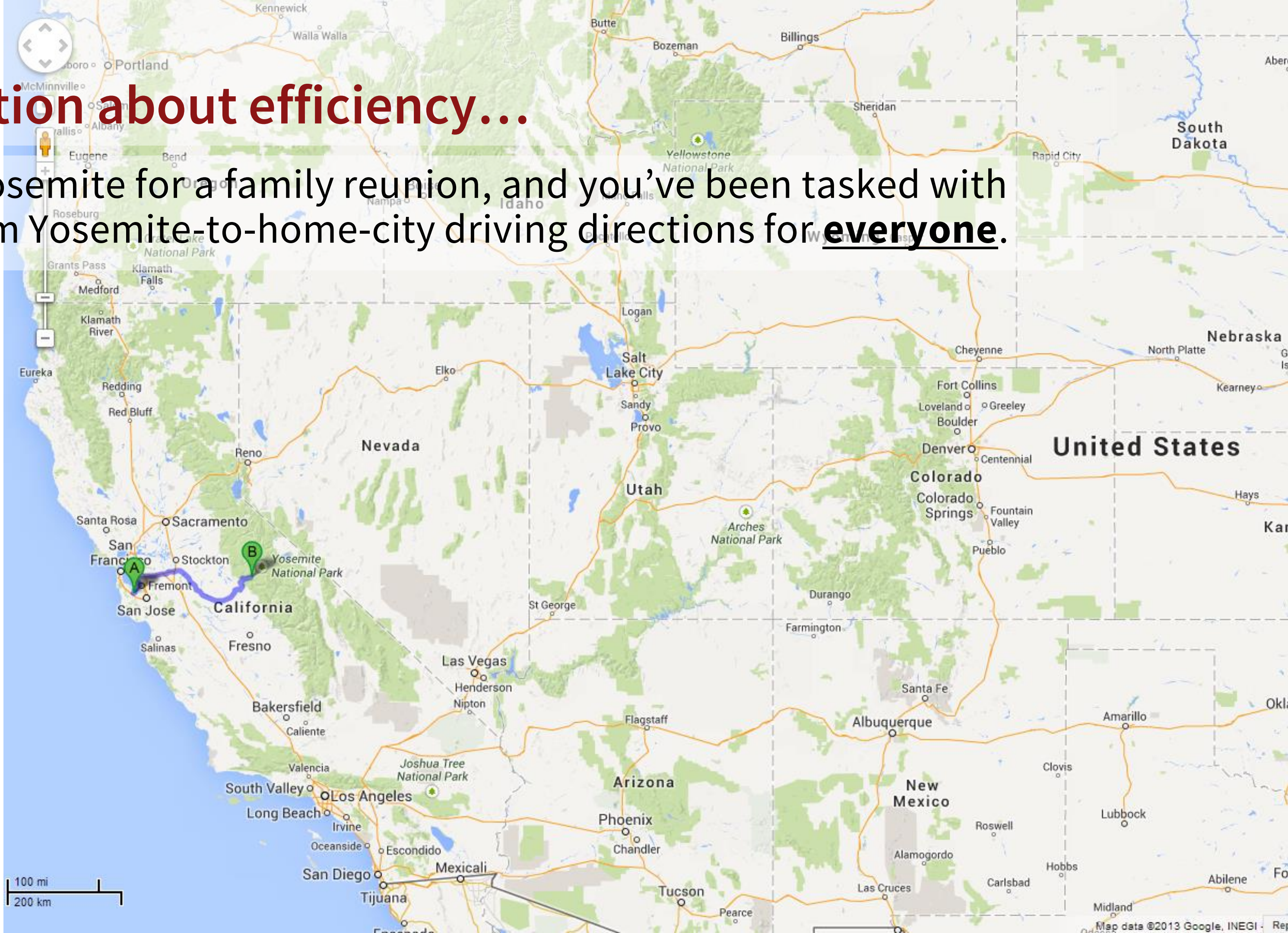
(3) If you follow the parent pointers from the desired end point, you will get back to the start point, and it will be the shortest way to do that

Quick question about efficiency…

Let’s say that you have an extended family with somebody living in every major city in the western U.S.

Quick question about efficiency…

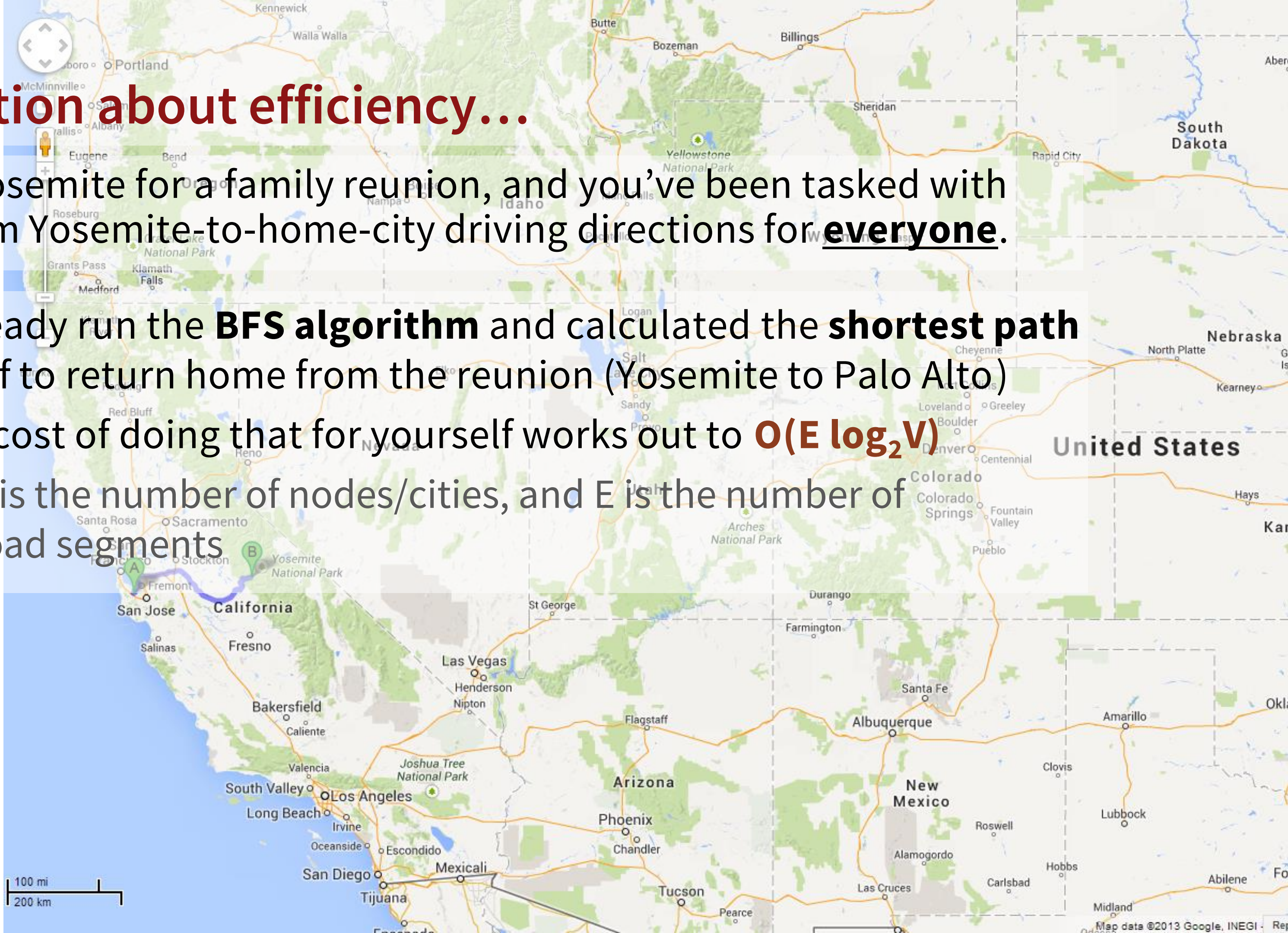
You're all in Yosemite for a family reunion, and you've been tasked with making custom Yosemite-to-home-city driving directions for **everyone**.



Quick question about efficiency…

You're all in Yosemite for a family reunion, and you've been tasked with making custom Yosemite-to-home-city driving directions for **everyone**.

- You've already run the **BFS algorithm** and calculated the **shortest path** for yourself to return home from the reunion (Yosemite to Palo Alto)
- The Big-O cost of doing that for yourself works out to **$O(E \log_2 V)$**
  - Where V is the number of nodes/cities, and E is the number of edges/road segments



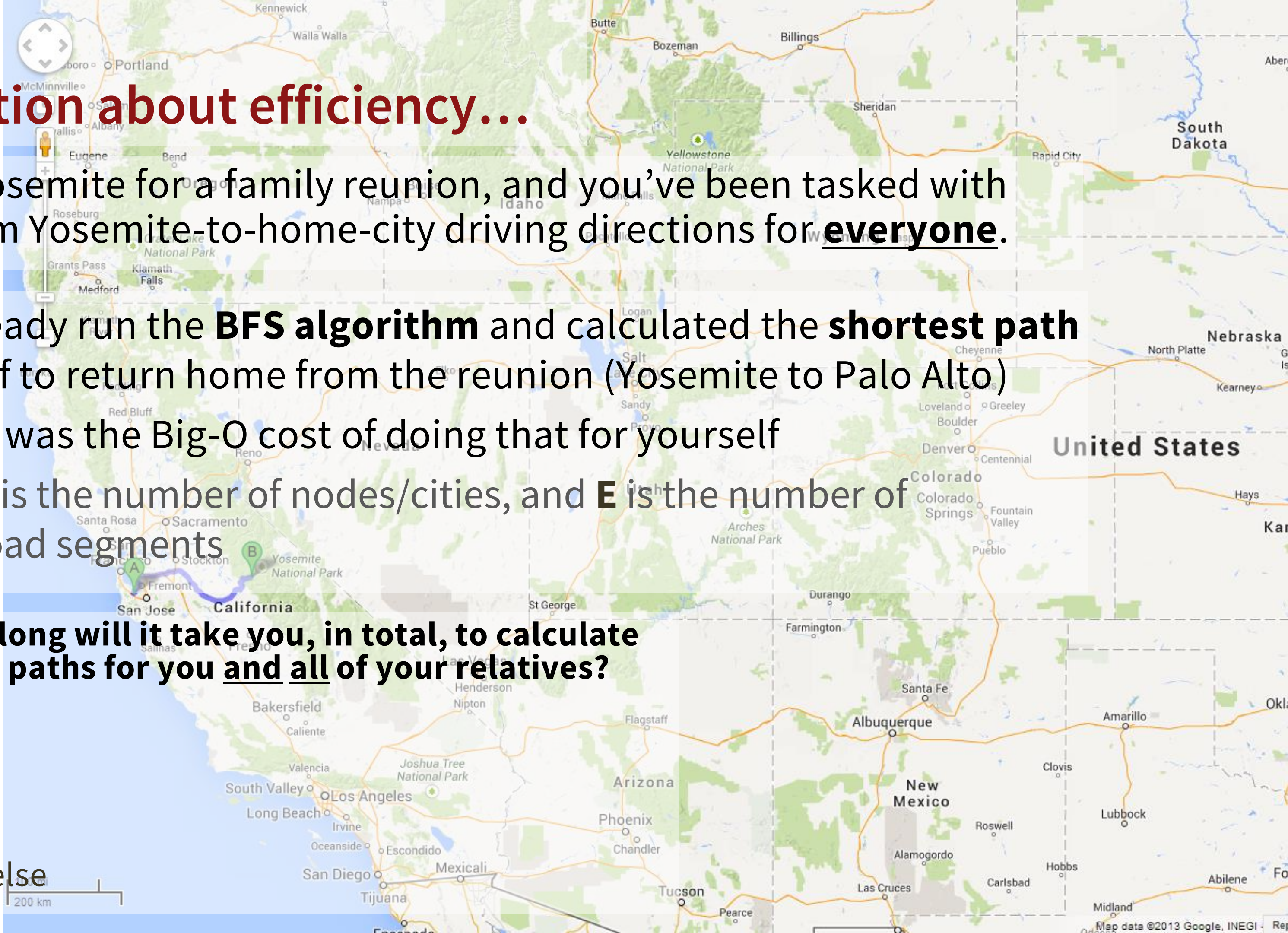
Quick question about efficiency…

You're all in Yosemite for a family reunion, and you've been tasked with making custom Yosemite-to-home-city driving directions for **everyone**.

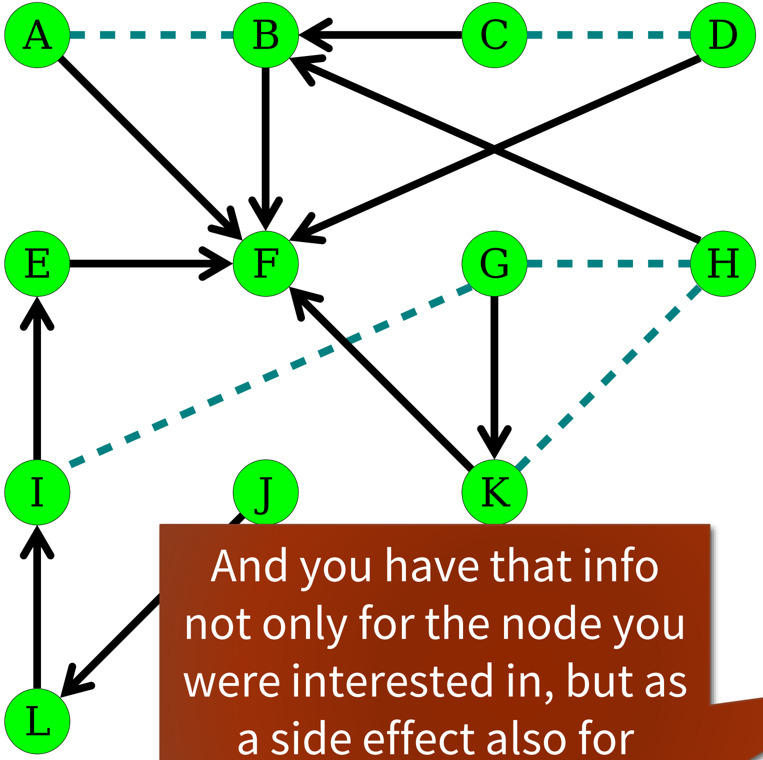
- You've already run the **BFS algorithm** and calculated the **shortest path** for yourself to return home from the reunion (Yosemite to Palo Alto)
- **$O(E \log_2 V)$** was the Big-O cost of doing that for yourself
  - Where **V** is the number of nodes/cities, and **E** is the number of edges/road segments

Your Turn: How long will it take you, in total, to calculate the shortest paths for you and all of your relatives?

A. $O(VE \log_2 V)$
B. $O(E \log_2 V^2)$
C. $O(V \log_2 E)$
D. $O(E \log_2 V)$
E. Something else



Breadth-First Search



And you have that info not only for the node you were interested in, but as a side effect also for every node in the graph!

THINGS TO NOTICE:

(1) We used a queue

(2) What's left is a kind of subset of the edges, in the form of 'parent' pointers

(3) If you follow the parent pointers from the desired end point, you will get back to the start point, and it will be the shortest way to do that

Quick question about efficiency…

You're all in Yosemite for a family reunion, and you've been tasked with making custom Yosemite-to-home-city driving directions for **<u>everyone</u>**.

- You've already run the **BFS algorithm** and calculated the **shortest path** for yourself to return home from the reunion (Yosemite to Palo Alto)
- **$O(E \log_2 V)$** was the Big-O cost of doing that for yourself
  - Where **V** is the number of nodes/cities, and **E** is the number of edges/road segments

Your Turn: How long will it take you, in total, to use BFS to calculate the shortest paths for you <u>and</u> <u>all</u> of your relatives?

A. $O(VE \log_2 V)$
B. $O(E \log_2 V^2)$
C. $O(V \log_2 E)$
D. $O(E \log_2 V)$
E. Something else

<u>No additional work</u> for BFS to determine the shortest paths for <u>all</u> your relatives, vs just for yourself!

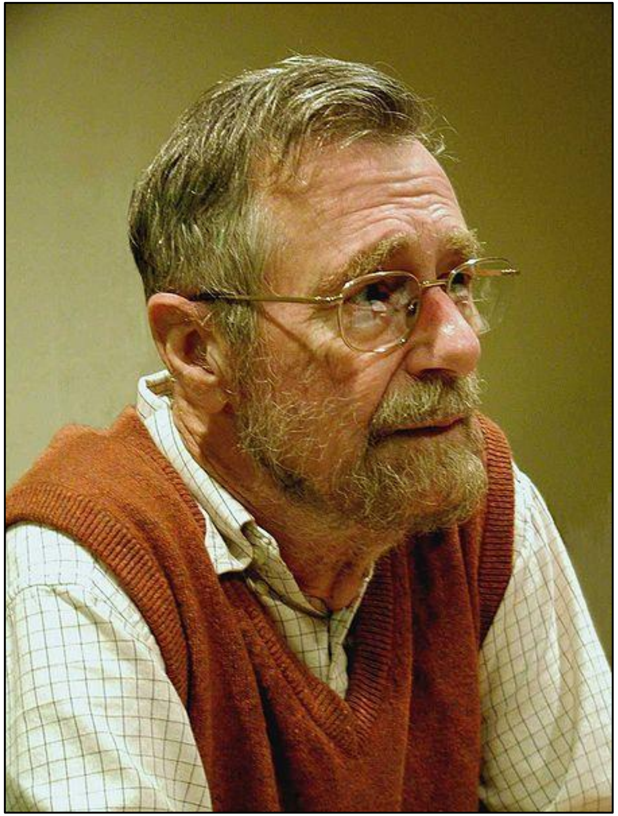
Dijkstra's Shortest Paths

(LIKE BREADTH-FIRST SEARCH, BUT **TAKES INTO ACCOUNT WEIGHT/DISTANCE** BETWEEN NODES)

Edsger Dijkstra

1930-2002

- THE multiprogramming system (operating system)
  - Layers of abstraction!!
- Complier for a language that can do recursion
- Dining Philosopher's Problem (resource contention and deadlock)
- Dijkstra's algorithm
- "Goto considered harmful" (title given to his letter)



The Structure of the "THE"-Multiprogramming System

Edsger W. Dijkstra
Technological University, Eindhoven, The Netherlands

A multiprogramming system is described in which all activities are divided over a number of sequential processes. These sequential processes are placed at various hierarchical levels, in each of which one or more independent abstractions have been implemented. The hierarchical structure proved to be vital for the verification of the logical soundness of the design and the correctness of its implementation.

KEY WORDS AND PHRASES: operating system, multiprogramming system, system hierarchy, system structure, real-time debugging, program verification, synchronizing primitives, cooperating sequential processes, system levels, input-output buffering, multiprogramming, processor sharing, multiprocessing
CR CATEGORIES: 4.30, 4.32

Introduction

In response to a call explicitly asking for papers "on timely research and development efforts," I present a progress report on the multiprogramming effort at the Department of Mathematics at the Technological University in Eindhoven.

Having very limited resources (viz. a group of six people of, on the average, half-time availability) and wishing to contribute to the art of system design—including all the stages of conception, construction, and verification,

Accordingly, I shall try to go beyond just reporting what we have done and how, and I shall try to formulate as well what we have learned.

I should like to end the introduction with two short remarks on working conditions, which I make for the sake of completeness. I shall not stress these points any further.

One remark is that production speed is severely slowed down if one works with half-time people who have other obligations as well. This is at least a factor of four; probably it is worse. The people themselves lose time and energy in switching over; the group as a whole loses decision speed as discussions, when needed, have often to be postponed until all people concerned are available.

The other remark is that the members of the group (mostly mathematicians) have previously enjoyed as good students a university training of five to eight years and are of Master's or Ph.D. level. I mention this explicitly because at least in my country the intellectual level needed for system design is in general grossly underestimated. I am convinced more than ever that this type of work is very difficult, and that every effort to do it with other than the best people is doomed to either failure or moderate success at enormous expense.

The Tool and the Goal

The system has been designed for a Dutch machine, the EL X8 (N.V. Electrologica, Rijswijk (ZH)). Charac-

On the cruelty of really teaching computing science

The second part of this talk pursues some of the scientific and educational consequences of the assumption that computers represent a radical novelty. In order to give this assumption clear contents, we have to be much more precise as to what we mean in this context by the adjective "radical". We shall do so in the first part of this talk, in which we shall furthermore supply evidence in support of our assumption.

The usual way in which we plan today for to-morrow is in yesterday's vocabulary. We do so, be-cause we try to get away with the concepts we are familiar with and that have acquired their meanings

Dijkstra's Algorithm

- Mark all nodes as gray.
- Mark the initial node *s* as yellow and at candidate distance **0**.
- Enqueue *s* into the priority queue with priority **0**.
- While not all nodes have been visited:
- Dequeue the lowest-cost node *u* from the priority queue.
- Color *u* green. The candidate distance *d* that is currently stored for node *u* is the length of the shortest path from *s* to *u*.
- If *u* is the destination node *t*, you have found the shortest path from *s* to *t* and are done.
- For each node *v* connected to *u* by an edge of length *L*:
  - If *v* is gray:
    - Color *v* yellow.
    - Mark *v*'s distance as *d* + *L*.
    - Set *v*'s parent to be *u*.
    - Enqueue *v* into the priority queue with priority *d* + *L*.
  - If *v* is yellow and the candidate distance to *v* is greater than *d* + *L*:
    - Update *v*'s candidate distance to be *d* + *L*.
    - Update *v*'s parent to be *u*.
    - Update *v*'s priority in the priority queue to *d* + *L*.