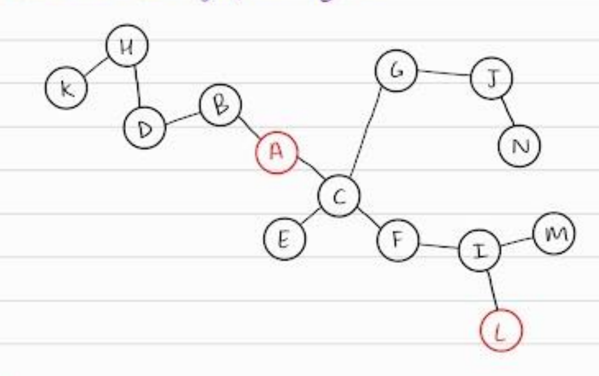
Breadth First Search

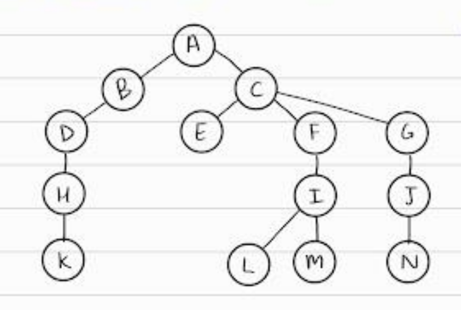
↳ "branches out" from start node
↳ utilizes a queue

Consider our graph again:



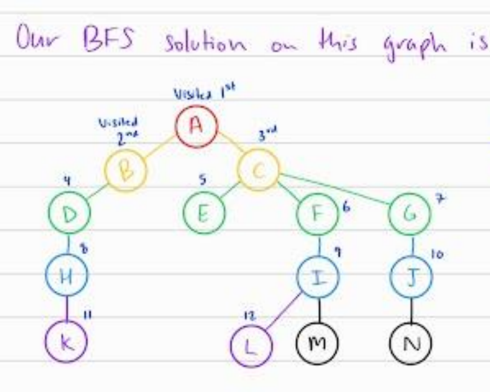
START: Node A
GOAL: Node L

Represented as a tree:



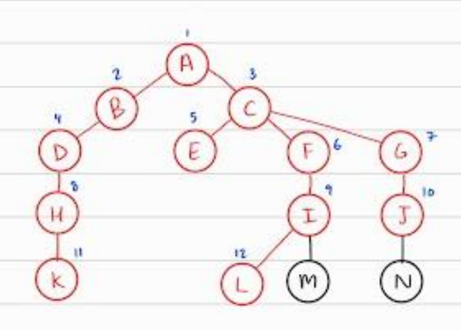
START: Node A
GOAL: Node L

Our BFS solution on this graph is:



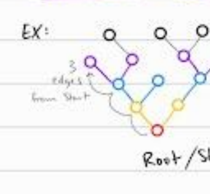
START: Node A
GOAL: Node L

or



Looking at the order of visited nodes, it clear that BFS records neighbors and explores them in the order they are visited!
See that the BFS algorithm begins at node and explores in incremental distances!

EX:

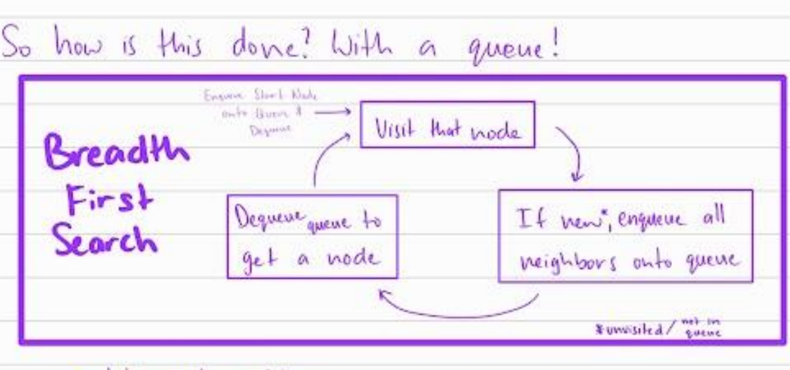


First visits Red, distance from start = 0
Orange, distance from start = 1
Blue, distance from start = 2
and so on...

Whereas DFS goes straight to the deepest/farthest node.

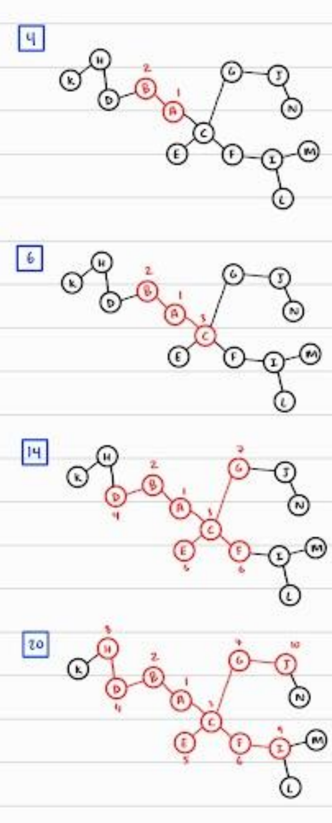
EX:

So how is this done? With a queue!



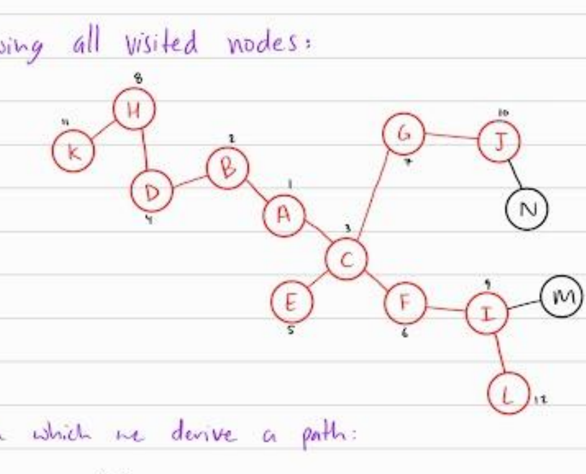
- let's visit neighbors Left → Right, so enqueue left neighbors first!

Forget trees! Let's walk through the solution to our graph:

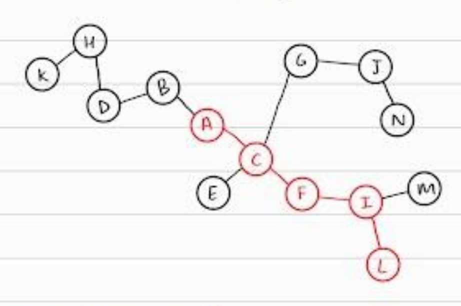


1. Initialize queue — Front [A] Back
2. Dequeue and visit Node → A
3. Enqueue neighbors of Node — Front [B C] Back
4. Dequeue and visit Node → B
5. Enqueue neighbors of Node — Front [C D] Back
6. → C
7. Front [D E F G] Back
8. → D
9. Front [E F G H] Back
10. → E
11. Front [F G H] Back
12. → F [G H I] ← our queue as an array
14. → G [H I J]
16. → H [I J K]
18. → I [J K L M]
20. → J [K L M N]
22. → K [L M N]
24. → L [M N] STOP, GOAL HAS BEEN FOUND!!

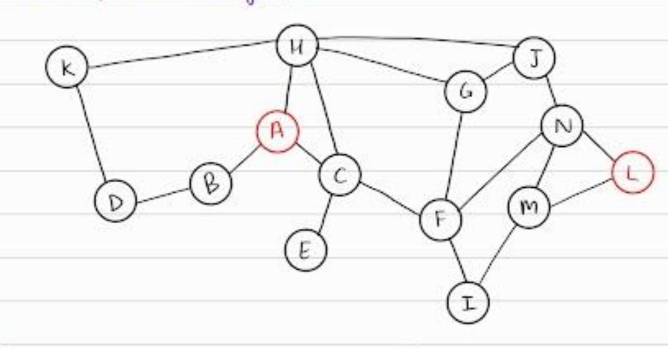
Showing all visited nodes:



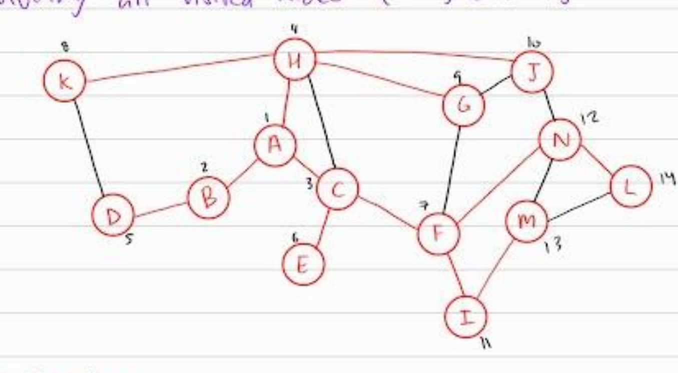
From which we derive a path:



Let's try another graph!



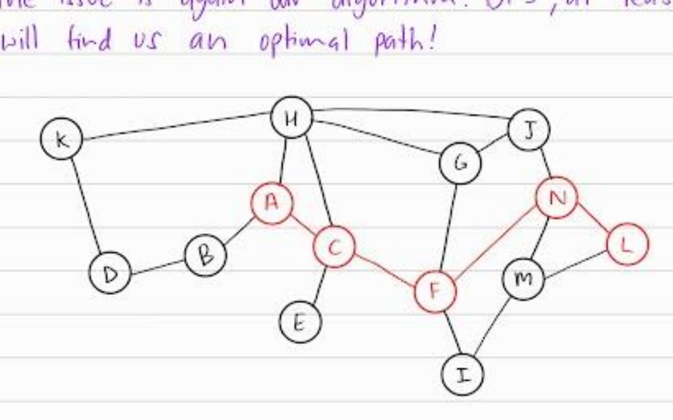
Showing all visited nodes: (visiting Left → Right



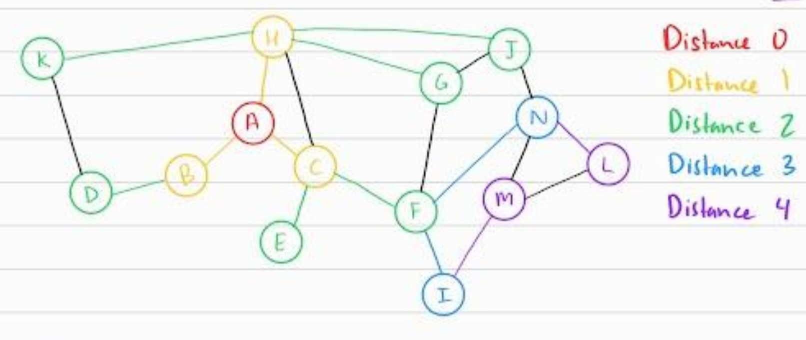
Visiting	Queue		
	[A]	K	[G J I N]
A	[B C H]	G	[J I N]
B	[C H D]	J	[I N]
C	[H D E F]	I	[N M]
H	[D E F K G J]	N	[M L
D	[E F K G J]	M	[L]
E	[F K G J]	L	[ ] STOP! GOAL HAS BEEN FOUND!
F	[K G J I N]		

But... our goal was visited last?? How was this intelligent?

The issue is again our algorithm. BFS, at least, will find us an optimal path!



If we think about (and color nodes according to distance from Start) we can see that our Goal is one of the farthest! Which we know BFS visits last.



Things to Understand

Q: How do we derive a path once our algorithm finds the goal?

Hint (or maybe the answer):

- The key-value data structure has the form

  key_0 : value_0
  key_1 : value_1
  ⋮

Where for each key there is a value.
Each key must be unique!

All values don't need to be unique.

Think of a dictionary:
bicycle : "..."
boat : "..."
⋮
caviar : "..."

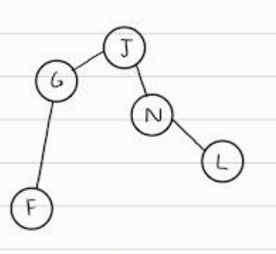
What kind of information can we represent with a dictionary? (aka key-value data structure)

Tons!!

What if we want keys to be people, and the associated value to be their age?

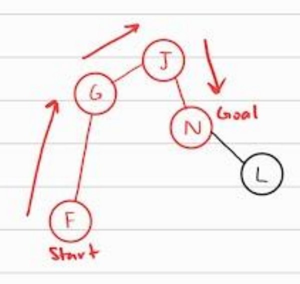
Elliot : 13
Lucia : 17
Grace : 8
Marisol : 4
⋮

What if we want to have each key be a node, and its value to be that node's neighbors?



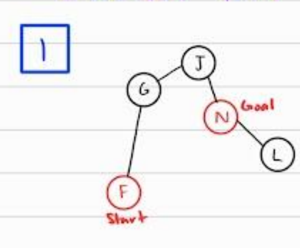
gives
G : F, J
J : G, N
N : J, L, F
F : G
L : N

How about if I read a path, where each key is a node in the path, and their value is the previous node AKA their predecessor

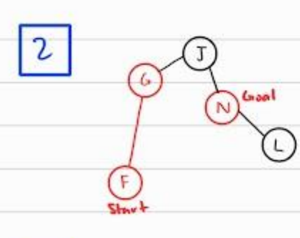


gives
F : None /Null
G : F
J : G
N : J

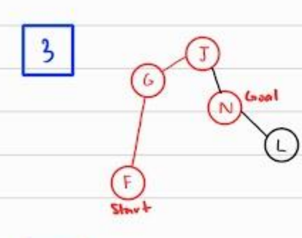
We can build that dictionary as we traversed that path.



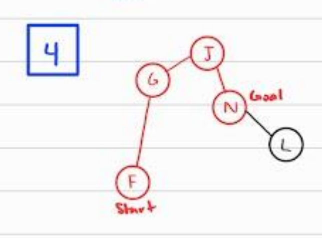
dictionary = { F : None }



dictionary = { F : None
G : F }



dictionary = { F : None
G : F
J : G }



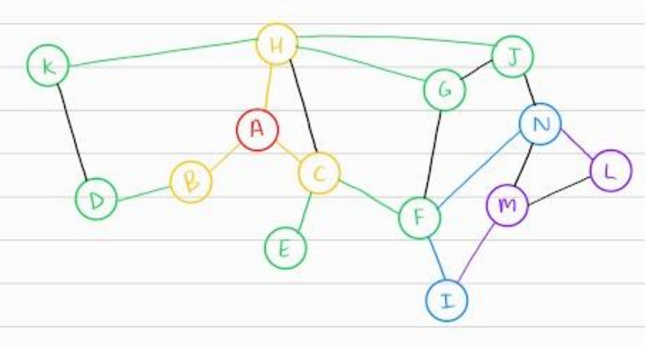
dictionary = { F : None
G : F
J : G
N : J }

Nice! Then beginning at the Goal, we can look to the value/predecessor,
then look at their predecessor,
⋮
until we reach the Start.

To get a final path:

F → G → J → N.

Doing so from a previous example:



predecessor_dictionary = { A : None,
B : A,
C : A,
H : A,
⋮
F : C
K : H
G : H
⋮
N : F
I : F
L : F }

Then begin at Goal and follow predecessors.

L → F → C → A !