Going Broad In A Graph: BFS Traversal

Vaidehi Joshi · Follow

Published in basecs · 15 min read · Sep 18, 2017

Breadth-first search graph traversal

Throughout the course of this series, we've made connections between things. We saw how linked lists were the foundations for stacks and queues, and how trees were a subset of graphs. As it turns out, everything that seems super complex — in the world of computer science, at least —

isn't as wildly complicated as it might first seem. Rather, everything is comprised of smaller, little things.

Realizing this is a big step to demystifying computer science, and everything that seems to hard about it. Large, intimidating ideas are really just smaller ones that have been built upon, extended, and restructured for another purpose. Once we can break down a problem into its smallest moving parts, that problem becomes far less overwhelming to try to understand (and solve!).

We're going to use this same methodology today as we take our first step into the the somewhat complicated world of graph traversal algorithms. Last week, we dipped our toes into the practical aspects of graph problems, by turning theory into practice. However, learning about a data structure isn't really enough; we need to know how to work with, how to manipulate it, and in many cases, how to search for something within it! Since so many computer science problems are really nothing more than graph search problems, it's useful to know how to search through graphs on a practical level.

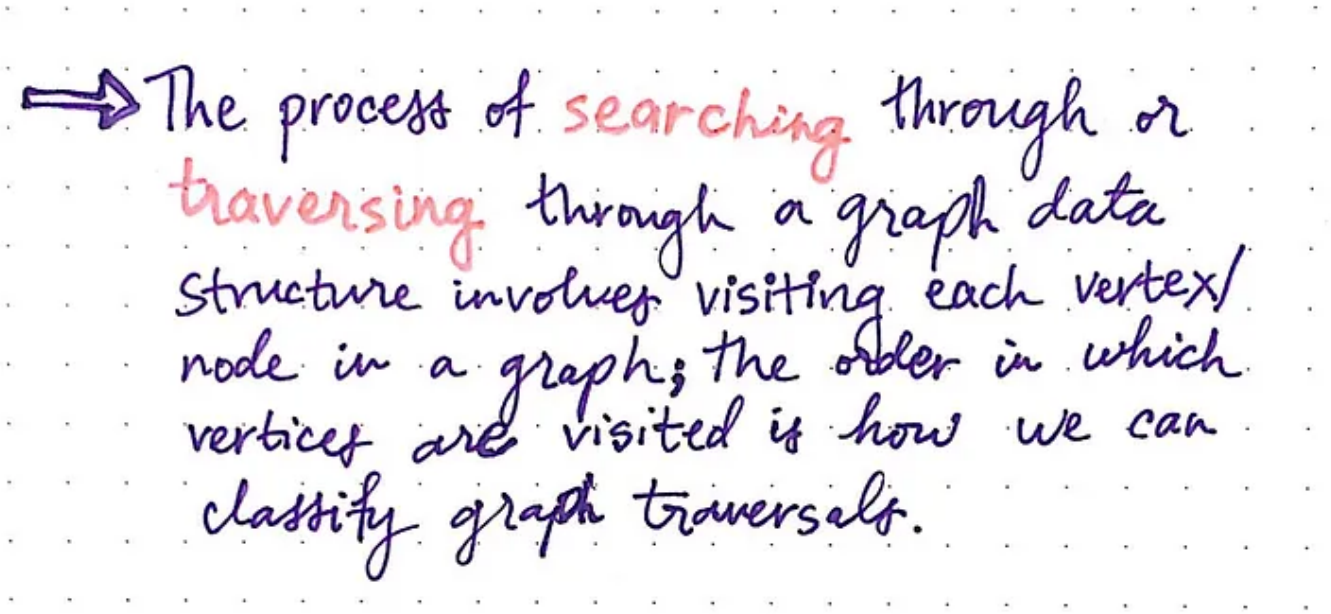
So, let's get to it!

Searching through a graph

If you're feeling a serious cases of déjà vu and thinking to yourself that we've already *covered* breadth-first search earlier in this series, then you'd be correct! In fact, we've covered depth-first search in this series as well! So why is it showing up again, since we already are experts at both of these algorithms?

Well, when we first learned about breadth-first search (BFS) and depth-first search (DFS) in this series, they were both in the context of tree traversal. Although trees are a subset of graphs, they are certainly very different structures. Thus, the process of graph traversal is *different enough* from tree traversal that it actually warrants revisiting these two search techniques in the context of graphs, rather than trees.

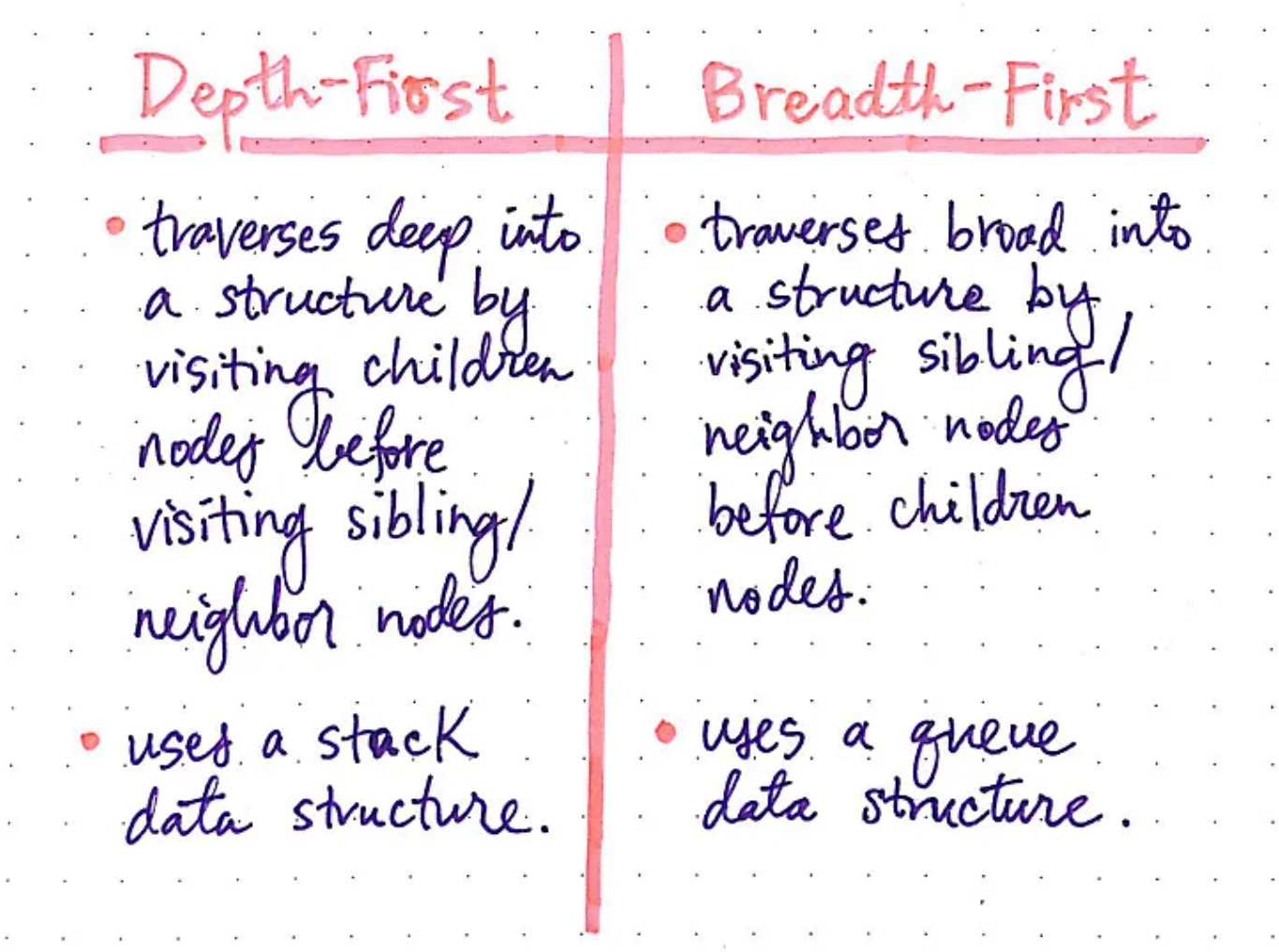
So, first things first: what do we mean when we say "traverse"? Let's start with a definition.



Graph traversal: a definition

The act of ***searching*** or ***traversing*** through a graph data structure is fairly simple: it just means that we're probably visiting every single vertex (and by proxy, every single edge) in the graph. At it's very core, the only difference between traversing a graph by breadth or by depth is *the order in which* we visit the vertices in a graph. In other words, the order in which the vertices of a graph are visited is actually how we can classify different graph traversal algorithms.

Since we're already familiar with DFS and BFS in the context of trees, let's quickly compare the two at a high level before diving into BFS in detail.

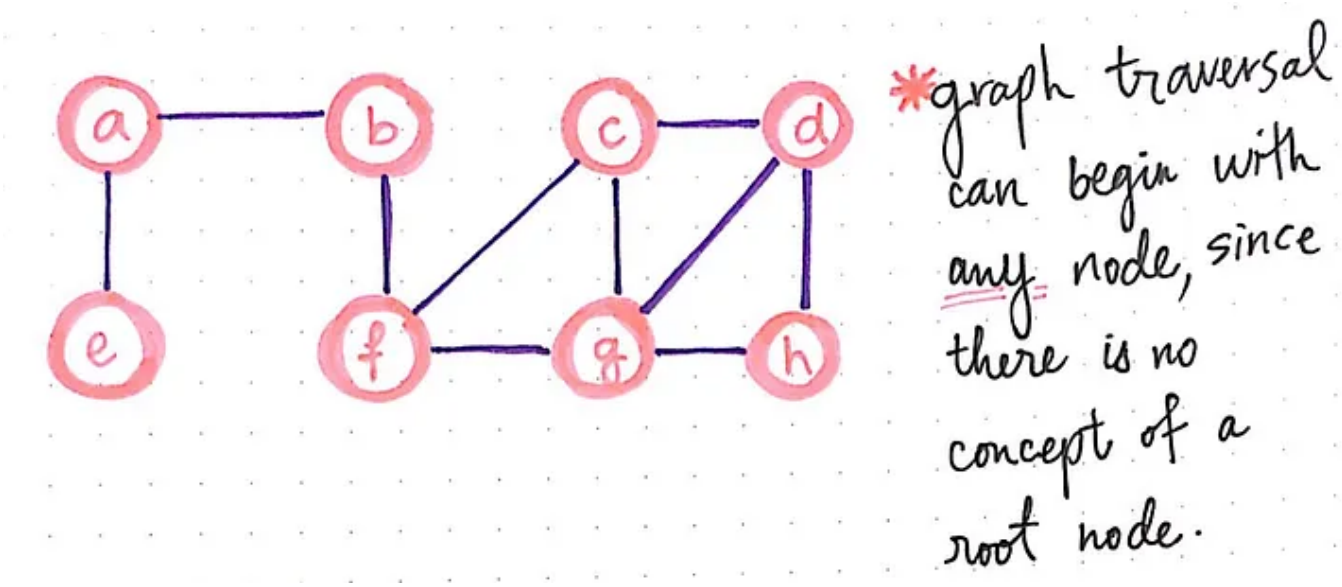


DFS and BFS: a quick review

Similar to how it is implemented on tree data structures, ***depth-first search*** traverses deep into a graph structures by visiting children nodes before visiting sibling — or *neighboring* — nodes. In both tree and graph traversal, the DFS algorithm uses a stack data structure. By comparison, the ***breadth-first search algorithm*** traverses broadly into a structure, by visiting neighboring sibling nodes before visiting children nodes. In both tree and graph traversal, the BFS algorithm implements a queue data structure.

We'll see that there are quite a few differences between these two algorithms based on whether they're being applied to a tree structure or a graph structure. But, let's get back to our task at hand: understanding the inner workings of BFS as applied to graphs!

In order to understand BFS graph traversal, we'll work with a single example: the undirected graph that is shown here.



Graph traversal can begin anywhere!

Before we can decide how to traverse through this graph, we need a starting point. As it turns out, the first — and probably the major — difference between graph and tree traversal is deciding where to start searching! In a tree traversal, we always start with the root node, and, for BFS, work our way down through the tree structure, level by level. But, when dealing with a graph, there is not obvious start, since there is no concept of a "root" node.

Graph traversal can begin with any vertex in the graph, so we'll choose one arbitrarily. For our example, using the eight-node graph illustrated above, we can randomly choose node `b` as our starting point.

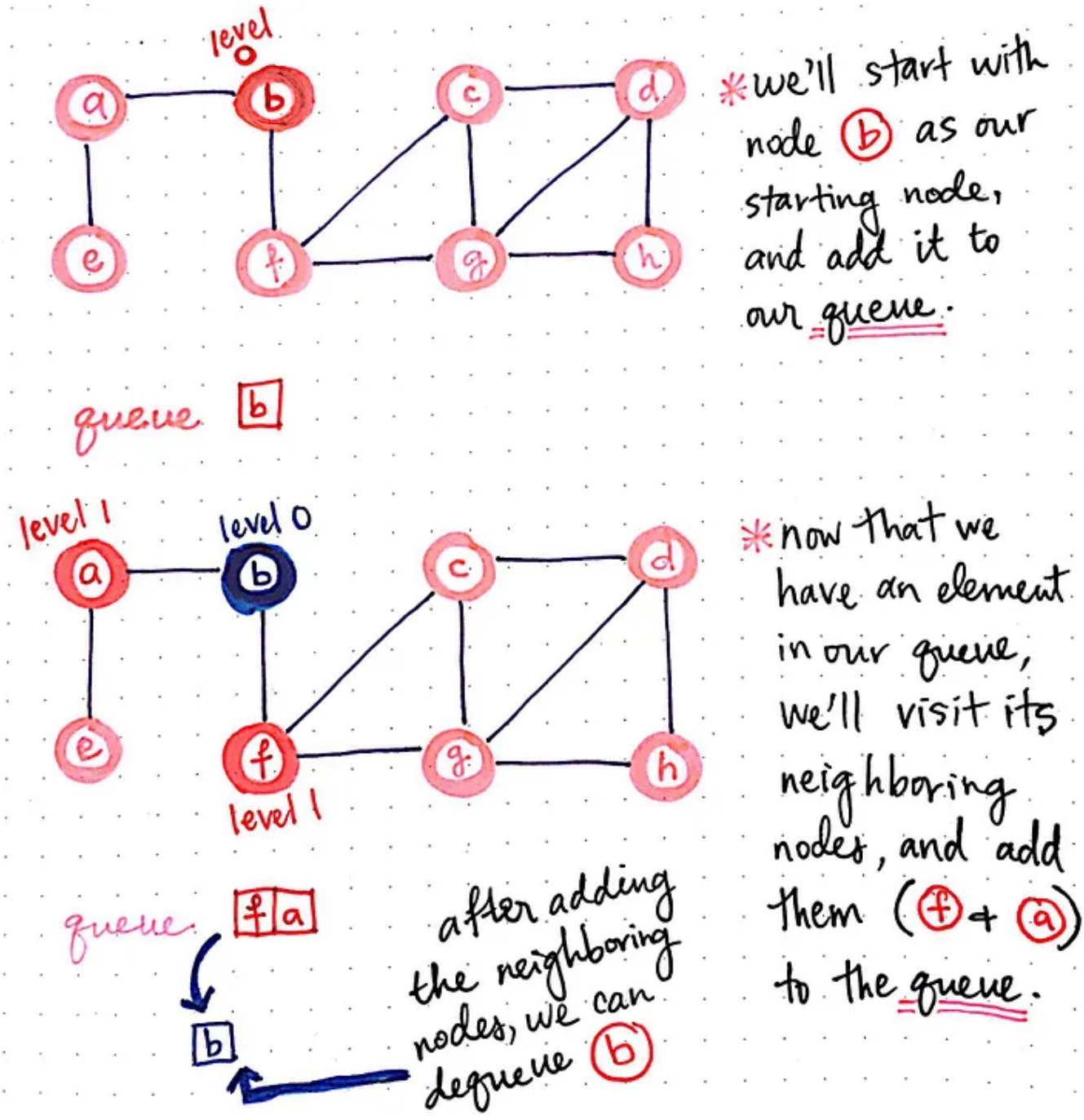
Step-by-step BFS traversal

Once we've chosen a starting point for our search, we have one major thing out of the way. The process of actually searching *by breadth* in a graph can be boiled down to a few steps, which we'll continue to repeat again and again, until we have no more nodes left to check.

The backbone of a breadth-first graph traversal consists of these basic steps:

1. Add a node/vertex from the graph to a queue of nodes to be "visited".
2. Visit the topmost node in the queue, and mark it as such.
3. If that node has any neighbors, check to see if they have been "visited" or not.
4. Add any neighboring nodes that still need to be "visited" to the queue.
5. Remove the node we've visited from the queue.

These five steps are continually repeated for every node in the graph, until we have no more nodes left to check in our queue. Of course, seeing these steps written out isn't nearly as powerful (or helpful!) as seeing an example of BFS in action, so let's use our example graph and run a breadth-first search on it, starting with node `b` as the "parent" node of our search.



BFS, part 1

Once we have node `b` as the starting point, we'll add it to our queue of nodes that we need to check. This is an important first step, since we will systemically keep iterating through our queue and "visit" all of the vertices that we add to it.

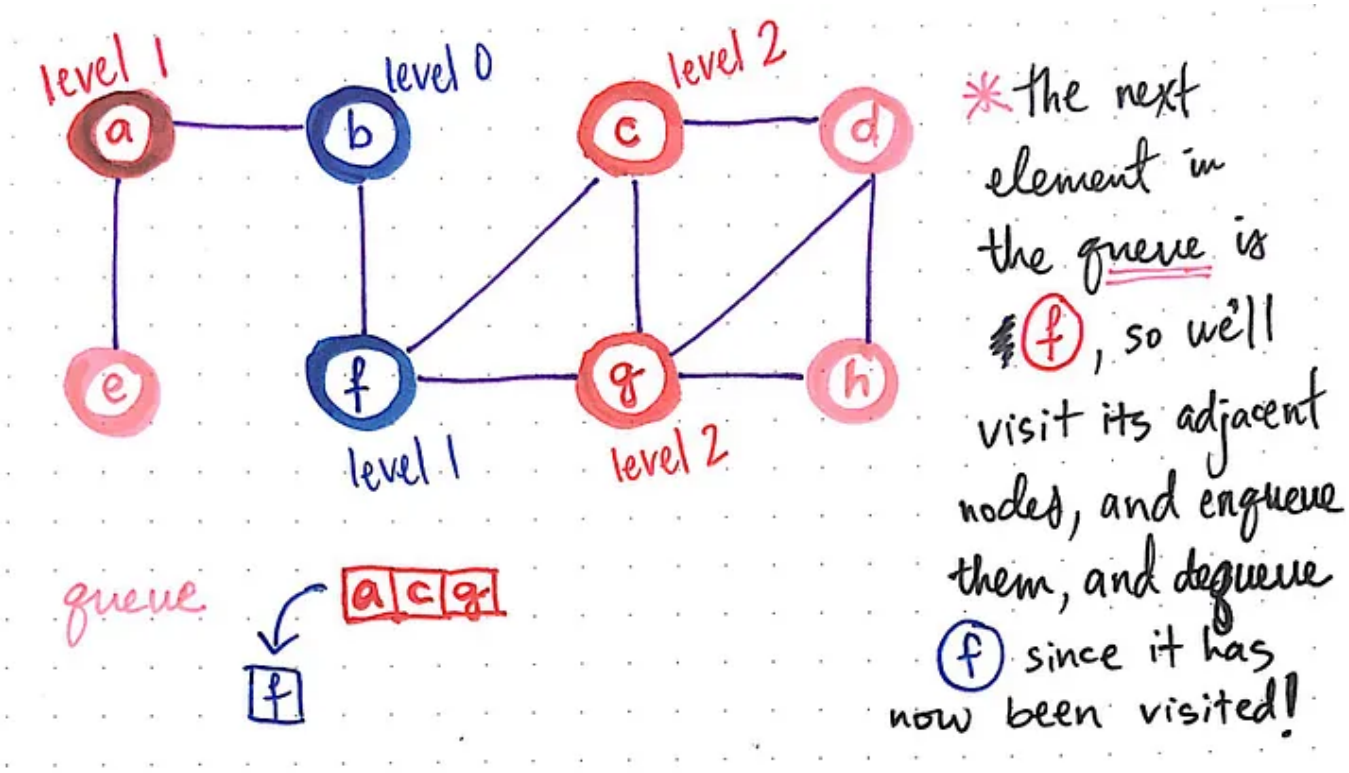
Once we have an element (node `b`) in our queue, we'll "visit" it, and mark it as "visited". The process of visiting a node effectively means that we're

noting that it exists, and also checking its neighboring nodes. In this case, the neighboring nodes of `b` are nodes `f` and `a`; so, we'll add them to our queue.

Once we've visited node `b` and added its neighbors to the queue, we can dequeue node `b`, since we've done everything that we need to with it. Effectively, we're dequeuing node `b`, and enqueuing nodes `f` and `a`. Note that either node `a` or `f` could have been added first in the queue; since they both are neighbors, it doesn't matter what order they're added to the queue, as long as they're added together.

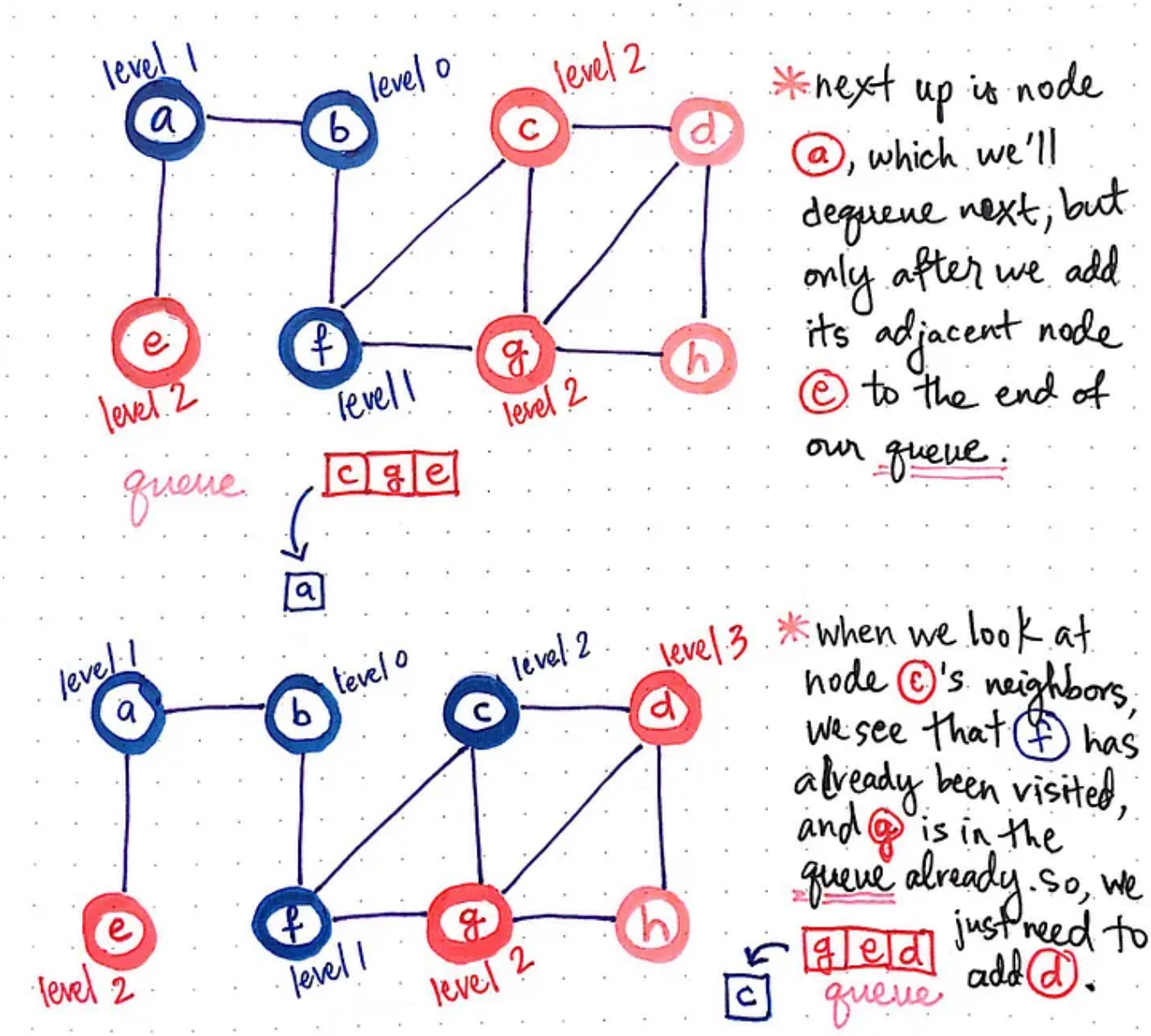
We'll also notice that each set of neighbors that we add to this queue causes us to move further and further away from our arbitrarily-chosen "parent" node. For example, if our "parent" node, node `b` was at a level `0` in our graph, then its neighboring nodes are one level away from the parent, putting them at a level `1` in the graph.

Now, we have two elements in the queue: `f` and `a`. We'll repeat the same steps for each of them. Since the next element is vertex `f`, we'll visit its adjacent nodes, mark node `f` as visited, which is shown by coloring it blue. We'll also add its neighbors (vertices `c` and `g`) to the queue.



BFS, part 2

Once we've added both `c` and `g` (in any order we choose), we can dequeue node `f` from the top of the queue, since we've finished the work of visiting it! We can see that both nodes `c` and `g` are two steps away from the "parent" node `b`; in other words, they're two nodes away from node `b`, which puts them at a level `2` distance from the main "parent" node.



BFS, part 3

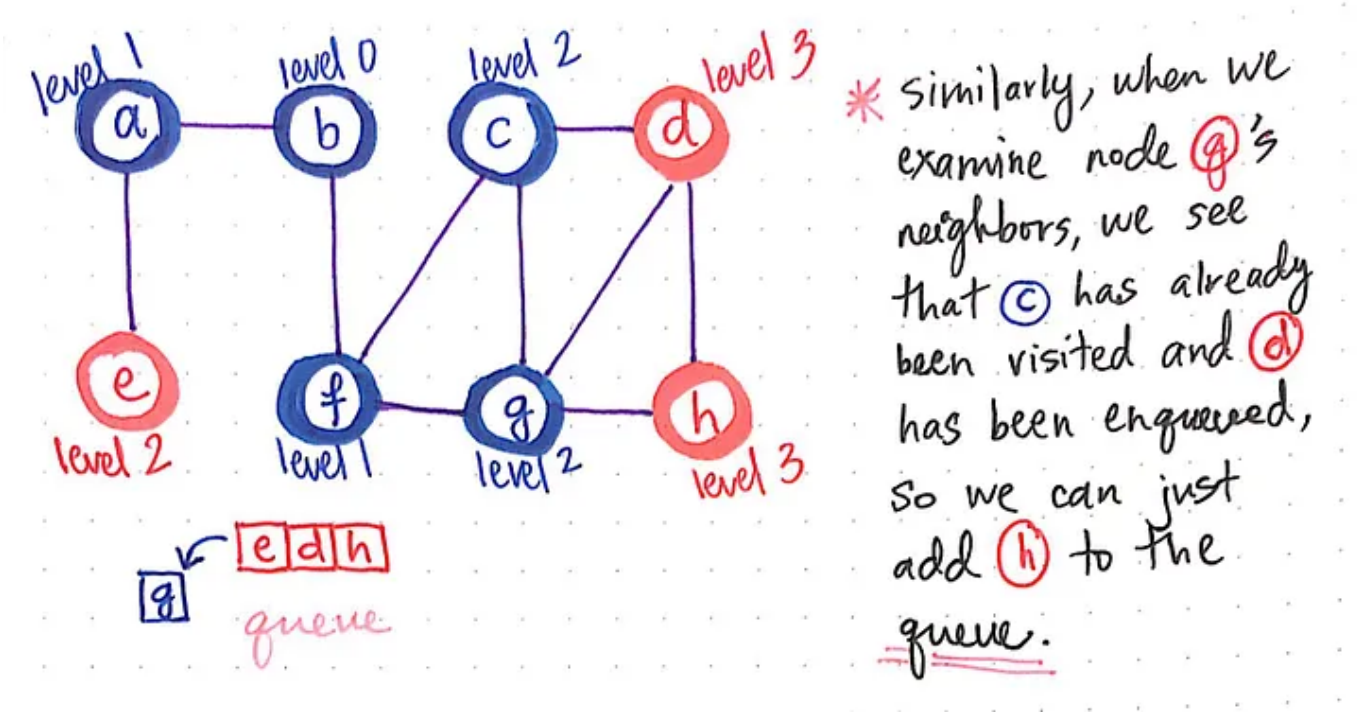
Next up is node `a`, which is at the front of the queue. We know what to do, right?

We'll add its adjacent nodes — it only has one, node `e` — to the end of our queue. We'll mark node `a` as "visited" by coloring it blue, and then we can dequeue it from the front of the queue. Now, our queue has references to vertex `c`, `g`, and `e` in it.

Again, we'll repeat the same steps for node `c`, which is at the front of the queue. Here is where it gets interesting!

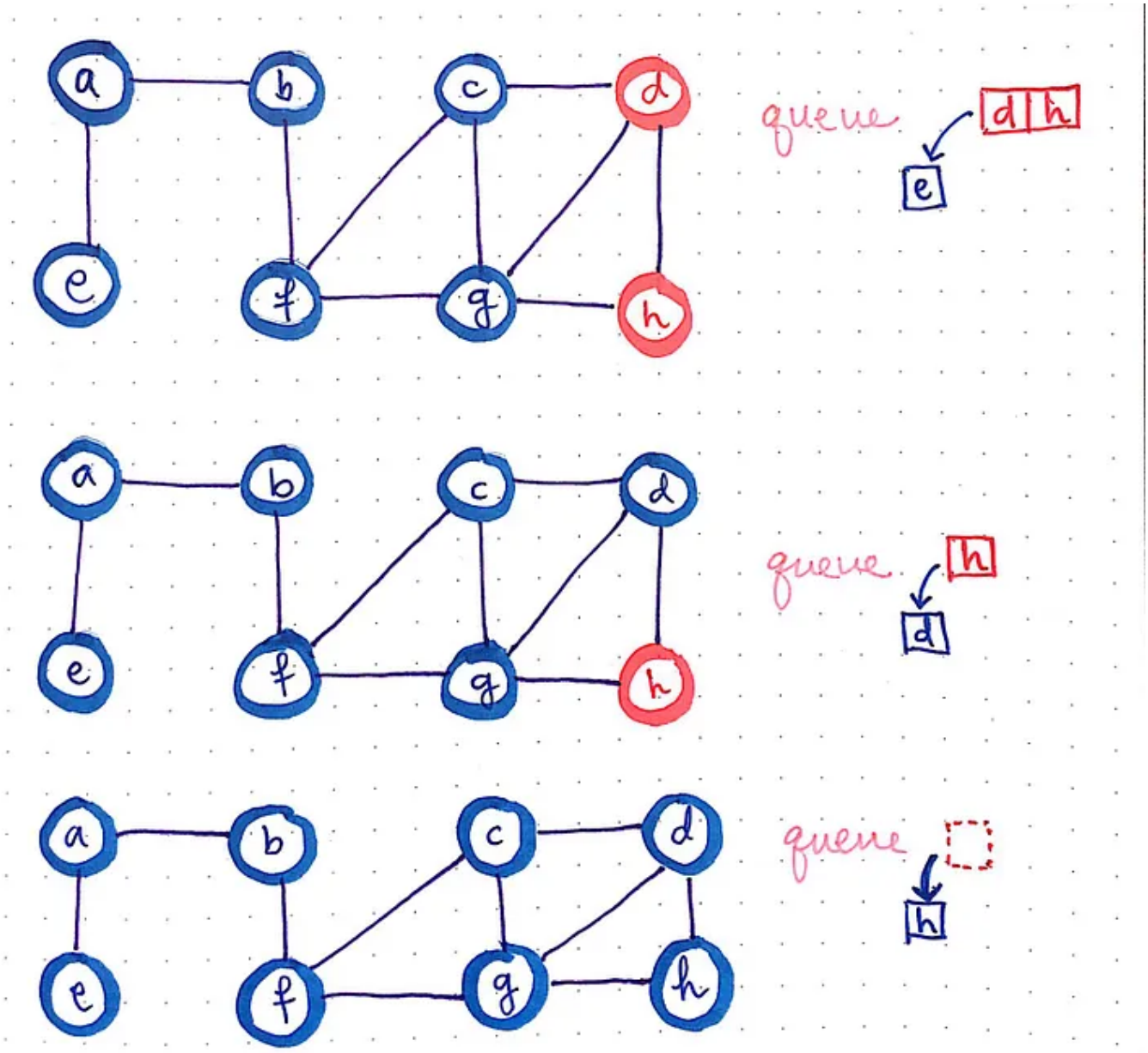
When we look at node `c`'s adjacent neighbor nodes, we'll see that one of its neighbors is node `f`. But, we've *already visited* node `f`! Similarly, another one of node `c`'s neighbors is node `g`, which is already in the queue. So, we know it's already in line to be visited, so we don't need to add it to the queue again. This leaves node `c`'s third and final neighbor: node `d`. This one has neither been visited, nor has been enqueued, so we can actually *do something* with this adjacent node. Well, to be more specific, we can add it to the queue.

Once we've checked all of node `c`'s neighbors, we can dequeue it, and move on with our lives! We'll move on to look at the next element in the queue, which is node `g`. It's a similar situation here, too. Its neighbor, node `c`, has already been visited, and another neighbor, node `d` was just enqueued, so there's nothing to do for either of those two vertices.



BFS, part 4

The only neighbor that needs handling is node `h`, which we'll add to the end of our queue. Once we've done that, we can dequeue node `g` from the front of the queue.



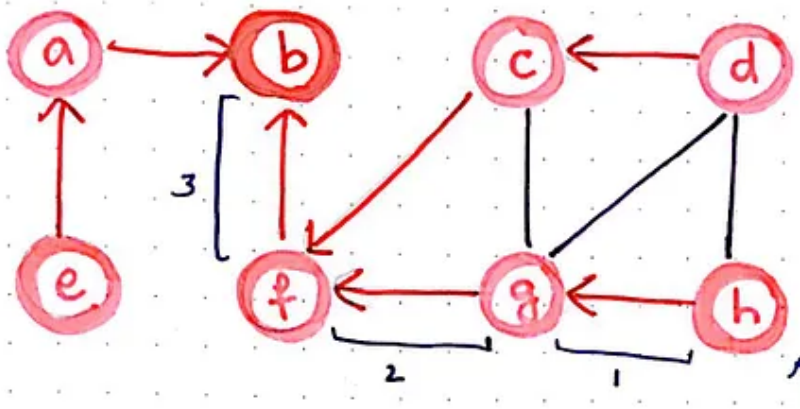
BFS, part 5

We'll notice now that node `h`, which we just enqueued, is three levels deep from the "parent" node; that is to say, it is a level `3` node, since it is 3 nodes away from our starting node `b`.

Our queue now has nodes `e`, `d`, and `h` in it. When we "visit" node `e`, we see that it has one neighbor, node `a`, which has already been visited. So, there's nothing we can do here except for mark `e` as visited, and dequeue it.

It's a similar story for nodes `d` and `h`. In both of these cases, each of these two node's neighbors have either 1) already been visited or 2) are already in the queue, waiting to *be* visited.

Eventually, we find that we've not only added all the nodes of the graph into the queue, but we've also cycled through all the nodes in the queue. At that point, we're just visiting a more, marking it as visited, and dequeuing it. Eventually, we finished iterating through the whole queue, and when the queue is empty, we're done with our traversal!

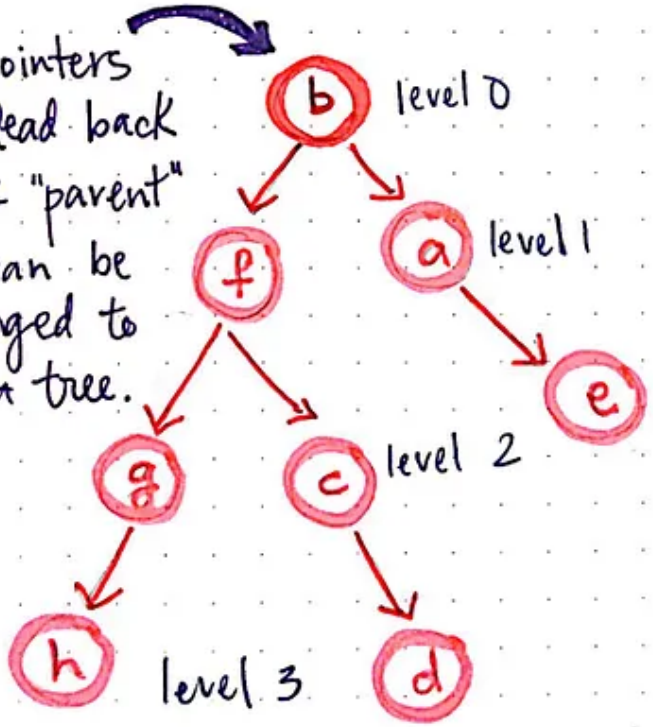


Shortest Paths

* It is often helpful to keep track of the "parents" of the nodes that we visit; whenever we visit a node, we can keep track of the parents before it, and end up with a shortest path through the graph.

[node (h) is a level 3 node, and has 3 edges to return back to the main "parent" node]

* The pointers that lead back to the "parent" node can be rearranged to form a tree.



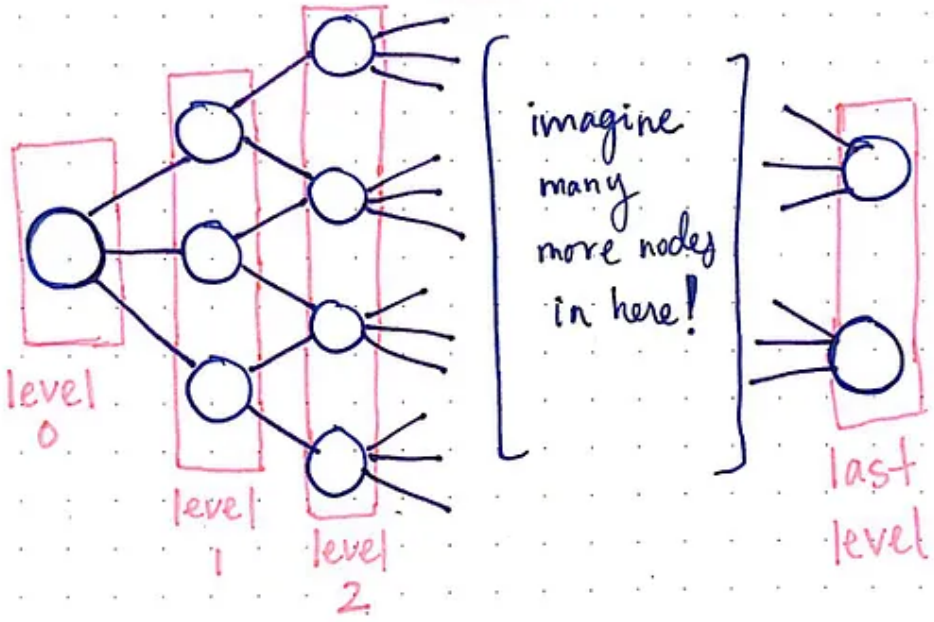
If we backtrack from a node and follow one of its parents until we reach the main "parent" node, we will have found one of the shortest paths to that node, which will be equivalent to the level of that node.

A unique thing about BFS is that it lends itself quite nicely to determining the ***shortest path*** between any node in the graph and the "parent" node. In fact, most BFS implementations will keep track of every single node's "parent" nodes that come before it. This is helpful because we can use the pointers of the path that we take to get to a node from the starting node in order to determine a shortest path in the graph.

The ***parent pointers*** that lead back to the starting node can be rearranged to form a tree. When we visualize these parent pointers as a tree structure (rather than as a graph), we can start to see the levels of each adjacent node come into play, and become far more obvious. Furthermore, if we *backtrack* from a node and follow its parent pointers back to the main starting node, we'll see that the level of the node in the tree corresponds to its level in the graph, and that the level of a node tells us how many steps we'll need to take to get from that node back to the starting "parent" node, node `b`. The pointers actually show us at least one of these "shortest paths", although it is likely that there will many different "shortest paths" for most of the graphs that we will deal with.

For example, node `h` is a level 3 node and has 3 edges/pointers that we can follow to return to the parent node, node `b`. There could also be a path from node `h` to `b` that is much longer than 3 steps, or there could be multiple 3-step "shortest paths" from node `h` to `b`. The important thing is that we have at least *one* of those shortest paths easily accessible to us by virtue of what we backtracked and found through our BFS graph traversal.

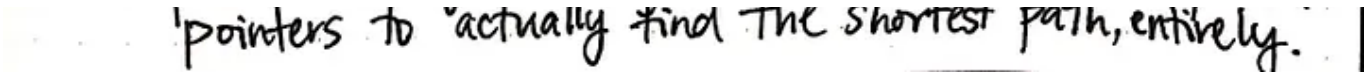
The power of using BFS to traverse through a graph is that it can easily tell us the shortest way to get from node x to node y.



* There will usually be many ways to traverse through a graph using node x as a starting point and node y as an ending point.

* There might even be many "shortest paths" from x to y.

* If we know the level of a node in relation to the "parent" node, we know how many steps the shortest path will require us to take; we can follow the parent



The power of BFS traversal

> The power of using breadth-first search to traverse through a graph is that it can easily tell us the shortest way to get from one node to another.

This is particularly helpful for huge graphs, which are fairly common in computer science problems. For example, in the image illustrated here, we might have many nodes and many levels, and want to be able to know how many steps to get from level `0` to the last level (whatever that might be). Having easy access to the shortest path(s) in this graph is super useful in solving this problem.

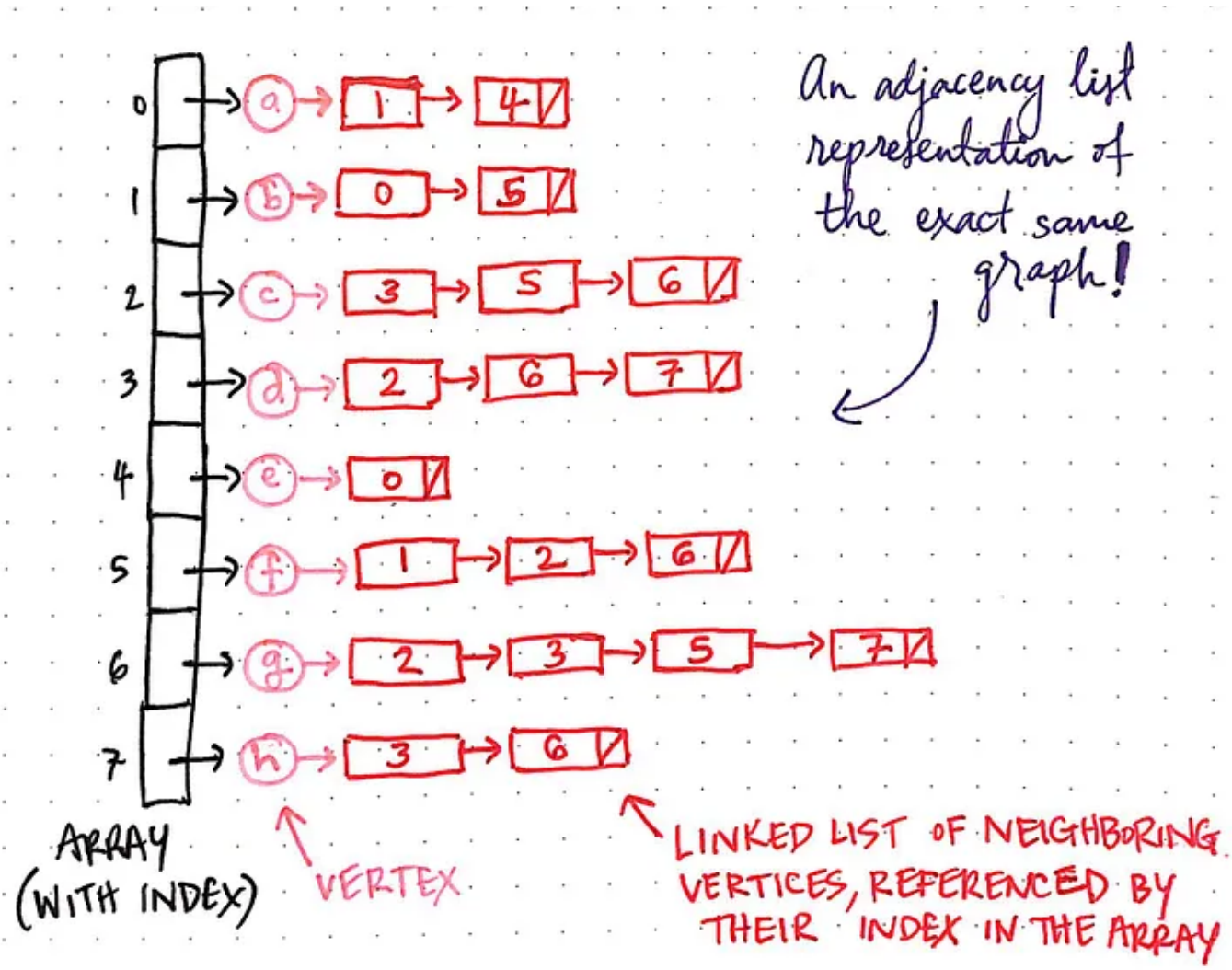
Of course, this is dependent upon us *keeping track* of the parent pointers; in most implementations of BFS, we'll see an array of lists (or something similar) being used to keep track of the shortest paths between the starting "parent" node and each ending node in a path. It's also worth noting that, in larger graphs, there will always be many ways to traverse from one node to another, and there will likely be many "shortest paths" between one node and another. But, if we know the level of a node in relation to the "parent" node that we start from, by proxy we also know the minimum number of steps to take in order to find the shortest path from any one node back to the "parent" node that we started with.

The complexities of breadth-first search

Learning about an algorithm like breadth-first search graph is exciting, but it's not nearly as fun without knowing how efficient it actually will be in

traversing through a graph! The best way to understand the runtime complexity of BFS graph traversal is by examining how it actually operates on a *graph representation* — in other words, by examining how the algorithm will function on a real graph, in its programmatic format.

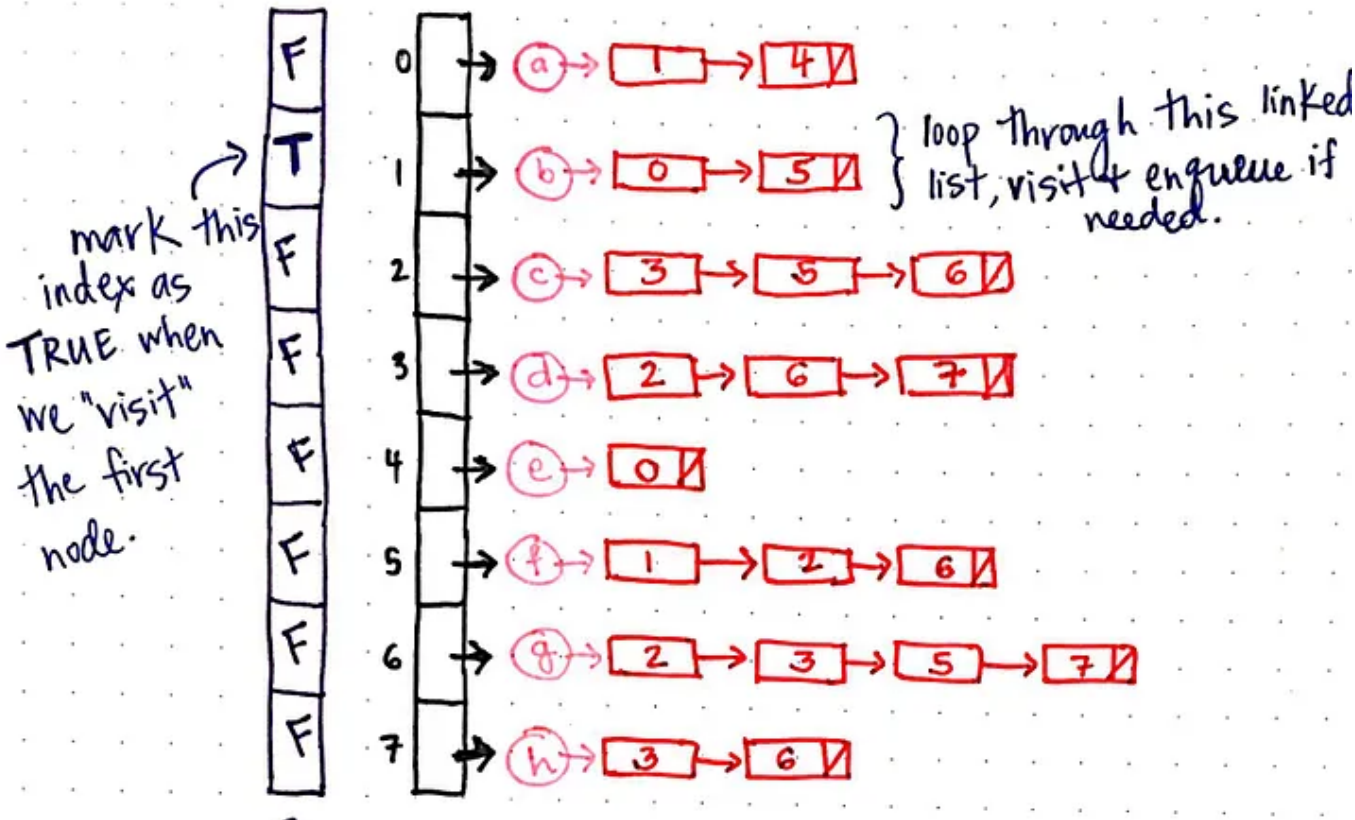
Last week, we learned a bit about the practical points of graphs, including graph representation. One of the most common forms of representing a graph is using an ***adjacency list***, which we know to be a hybrid between an *edge list* and an *adjacency matrix*. The drawing below is what our graph would look like in an adjacency list representation.



An adjacency list representation of a graph using BFS

We'll notice that there is an indexed array, each with a reference to a vertex in the graph. Each vertex has, in turn, references to a linked list of neighboring vertices. The linked list of a vertex contains references to the index of neighboring nodes for that vertex.

So, for example, vertex `e` has only one neighboring node, `a`. Hence, the adjacent linked list for vertex `e` contains a reference to index `0` in the array, which is where node `a` lives. If we take a quick glance through this adjacency list, we'll see that it maps directly to the same graph that we've been dealing with.



* When we choose the arbitrary starting ("parent") node to visit we will mark it as visited by changing its stated in the "visited" array.
* After marking a vertex as "visited", we can enqueue it.
* When we dequeue a vertex, we examine its neighboring nodes by iterating/looping through that vertex's adjacency linked list.
* If a neighboring node hasn't been visited, isn't TRUE in the "visited" array, we mark it as "visited" and enqueue it.
* We continue doing this til we reach a NULL pointer in the linked list + queue is empty.

When we implement BFS on this adjacency list graph representation, we’ll need one additional data structure: a “visited” array that we’ll use to keep track of which vertices have been visited (and which ones haven’t). This array will contain only `FALSE` boolean values to start. But, when we visit the very first node, we’ll mark it to `TRUE`, and eventually the entire array will be filled with `TRUE` values at the end of the BFS algorithm.

We can start to understand how much time it actually takes to run BFS on a graph if we consider how the algorithm would operate on this adjacency list.

For example, let’s take the first step of adding node `b` to the queue and actually visiting it; what all would that involve?

1. When we choose the arbitrary starting “parent” node of `b`, we’d mark it as visited by changing its state in the “visited” array.
2. Then, after changing its value from `FALSE` to `TRUE` in the “visited” array, we’d enqueue it.
3. Next, when dequeing the vertex, we need to examine its neighboring nodes, and iterate (loop) through its adjacent linked list. Node `b` has two neighboring nodes, located at indices `0` and `5` respectively.
4. If either of those neighboring nodes hasn’t been visited (doesn’t have a state of `TRUE` in the “visited” array), we’d mark it as visited, and enqueue it.
5. Finally, we’ll keep doing this until we reach a `NULL` (empty) pointer in the linked list, before moving on to the next node in the queue. Once the queue is totally empty, we know that we’re done running the algorithm.

* We only iterate/loop through a vertex's adjacency list once. That is to say, for each vertex, we look at its edges/neighboring nodes only once.

* If the graph is undirected, an edge would appear twice, in two different adjacency lists. If the graph is directed, an edge would appear only once.

Time Complexity

visiting n nodes + number of edges in the adjacency list to be visited

every node is enqueued once

for an undirected graph, 2|E|

for a directed graph, |E|

The runtime complexity of BFS graph traversal is O(V+E).

The runtime complexity of BFS graph traversal

Thinking about what we need to do for one single node, the runtime complexity becomes a bit more obvious. For every single vertex in the graph, we will end up iterating through the vertex's adjacency list *once*. That is to say, we will need to look at its neighboring nodes (which, in the adjacency list, is really jsut a representation of all of its edges) only one time — when we go to dequeue it, and enqueue any necessary neighbors.

However, we will recall that, if the graph is *undirected,* each edge will appear twice in an adjacency list, once for each node that is connected to it. But if the graph is *directed,* then each edge only appears once, in only one node's adjacency list.

> If we must visit every node once, and check every edge in its adjacency list, the runtime complexity for both a directed and undirected graph is the sum of the vertices and their edges as represented by the graph in its adjacency list representation, or O(V + E)

For an *undirected* graph, this means we visit all the vertices (V), and $2|E|$ edges. For a *directed* graph, this means we visit all the vertices (V) and $|E|$ edges. The size of an adjacency list representation of a graph is directly related to how much time it will take us to traverse through it using breadth-first search, which makes it a ***linear*** algorithm!

The BFS graph traversal algorithm is a popular search solution, and is particularly handy for quickly determining the shortest path between two locations in a graph. However, what's even more interesting to me is the strange, and somewhat sad origin of this search algorithm.

It was actually first derived by a German engineer named Konrad Zuse, who created the world's first programmable computer, and is often referred to as the inventor of the modern computer. Zuse first theorized the BFS graph traversal algorithm in 1945 as a solution for finding the connected components, or two connected vertices, of a graph data structure. He wrote about the BFS algorithm in his PhD thesis, which wasn't even published until 1972. The plot twist in this story is that, when Zuse first wrote about BFS in his thesis, his idea was actually *rejected*!

The BFS algorithm was later "reinvented" fourteen years later by Edward F. Moore in 1959, an American math professor, who (re)created the algorithm as a solution for finding the shortest path between two points in a maze. Just a few years later, in 1961, it was independently discovered by a researcher named C.Y. Lee, who was working at Bell Telephone Labs at the time. Lee's paper, *An Algorithm for Path Connections and Its Applications*, describes BFS as a wire routing algorithm to find an optimal path between two points on an IBM 704 computer.

It's wild to think that the first person to come across breadth-first search as a solution to graph traversal was rejected for his innovative idea. If anything, it's a lesson in persistence, and a reminder that sometimes, even the most well-known, complex, and widely-used solutions in computer science had to fight for their right to exist.

Resources

Because BFS (and DFS!) are both well-known algorithms, it's not too difficult to find resources that dive into the nitty-gritty of how they operate. If you're looking to learn more, or want to read about some the more complex aspects of this algorithms, here are some great resources to start with!

1. Graph Traversal Visualizations, VisuAlgo
2. Breadth-First Search, Department of Computer Science, Harvard
3. Breadth-first Search (BFS) on Graphs Part 2 — Implementation, Sesh Venugopal
4. Breadth-First Search (BFS), MIT OpenCourseWare
5. Graph Traversals — Breadth First and Depth First, CollegeQuery

Programming | Data Structures | Algorithms | Computer Science | Tech

Written by Vaidehi Joshi

29K Followers · Editor for basecs

Writing words, writing code. Sometimes doing both at once.

Follow

More from Vaidehi Joshi and basecs