Algorithms on Graphs: Breadth First Search

Step 1

Now that we have learned about the basics of graphs and graph representation, we can finally return to the original real-world problems we had discussed. The first real-world problem was the "navigation problem": given a starting location *start* and a destination location *end*, can we find the shortest driving path from *start* to *end*? The second real-world problem was the "airport problem": given a starting airport *start* and a destination airport *end*, can we find the shortest series of flights from *start* to *end*? The third real-world problem was the "network routing problem": given a starting port *start* in a network and given a destination port *end* in a network, can we find the shortest path in the network from *start* to *end*?

It should be clear by now that all three of these problems, as well as all other real-world problems that are similar in nature, are actually reduced to the exact same computational problem when they are represented as graphs: given a starting node *start* and a destination node *end*, what is the shortest path from *start* to *end*? In the next sections, we will discuss various graph algorithms that solve this "shortest path problem".

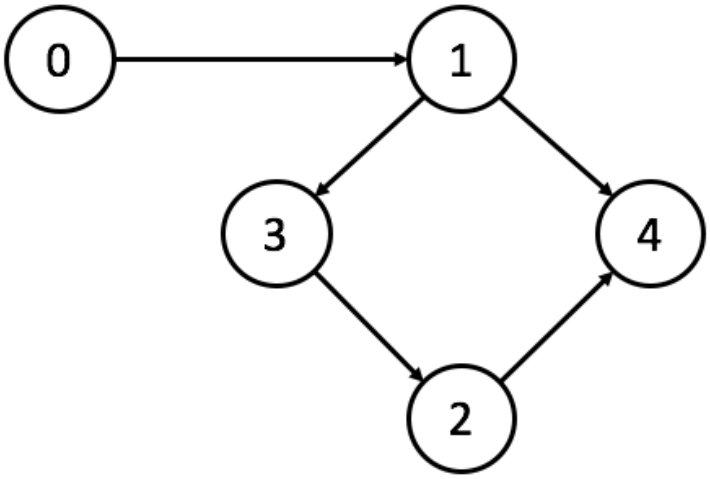
Step 2

Fun Fact: In many cases, you will often only be interested in finding the shortest path from a particular vertex to another *single* particular vertex. However, as of today, there is no known algorithm which can run more efficiently by only dealing with a particular single destination vertex as opposed to just finding the shortest path from one to *all* other vertices. As a result, the best optimization that we can do is to include an "early out" option and hope that it happens early enough in the computation (i.e. return from the algorithm once it has computed the shortest path from the source to the destination, but not necessarily all the rest of the vertices).

STOP and Think: Why does including an "early out" option **not** improve the worst-case time complexity of a shortest path algorithms?

Step 3

One algorithm to explore all vertices in a graph is called **Breadth First Search (BFS)**. The idea behind **BFS** is to explore all vertices reachable from the current vertex before moving onto the next. We will use the graph below as an example.



Suppose we start at 0. **BFS** will then choose to "visit" 1, since it is immediately reachable from 0. From 1, **BFS** will choose to "visit" 3 *and* 4 (not simultaneously, but in some arbitrary order) since they are *both* immediately reachable from 1. After, **BFS** will "visit" 2 as it reachable from 3.

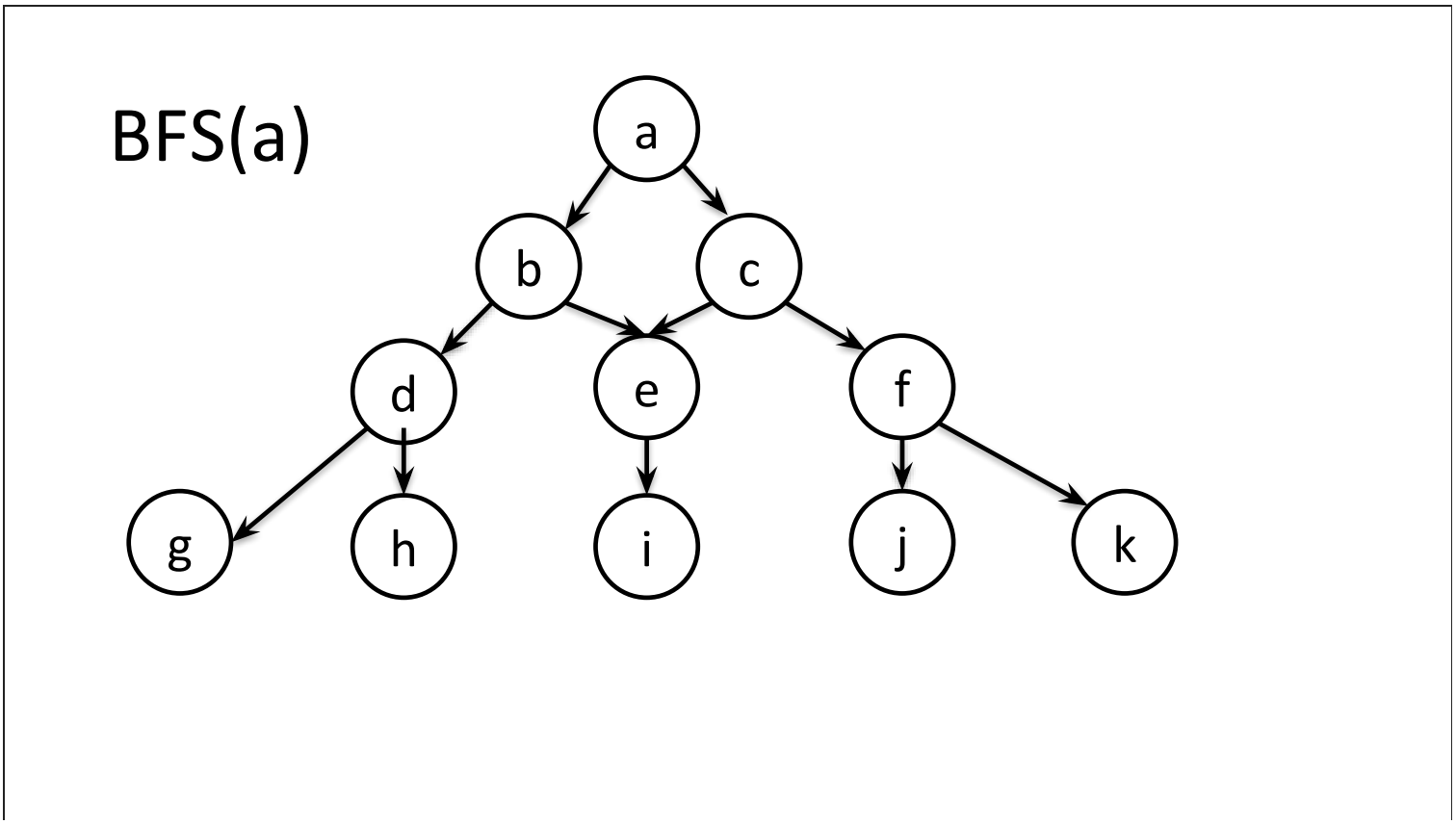
The rough algorithm for **BFS** can be thought of as:

1. Begin at the starting node *s*. It has distance 0 from itself.

2. Consider nodes adjacent to *s*. They have distance 1. Mark them as visited.
3. Then consider nodes that have not yet been visited that are adjacent to those at distance 1. They have distance 2. Mark them as visited.
4. Etc. etc. until all nodes reachable from *s* are visited.

Step 4

Breadth First Search can be difficult to understand without being able to actually see the algorithm in action. As a result, we've included a visualization below that shows the order in which each vertex is explored. Use the arrows in the bottom left to walk through the slides.

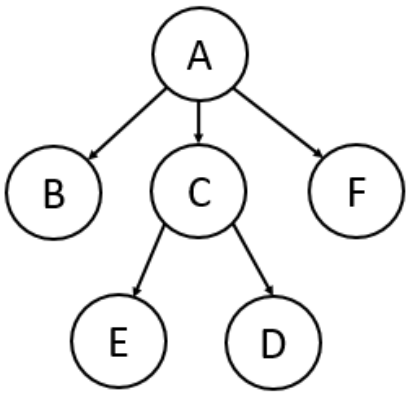


| Slide 1 |

Slides

Step 5

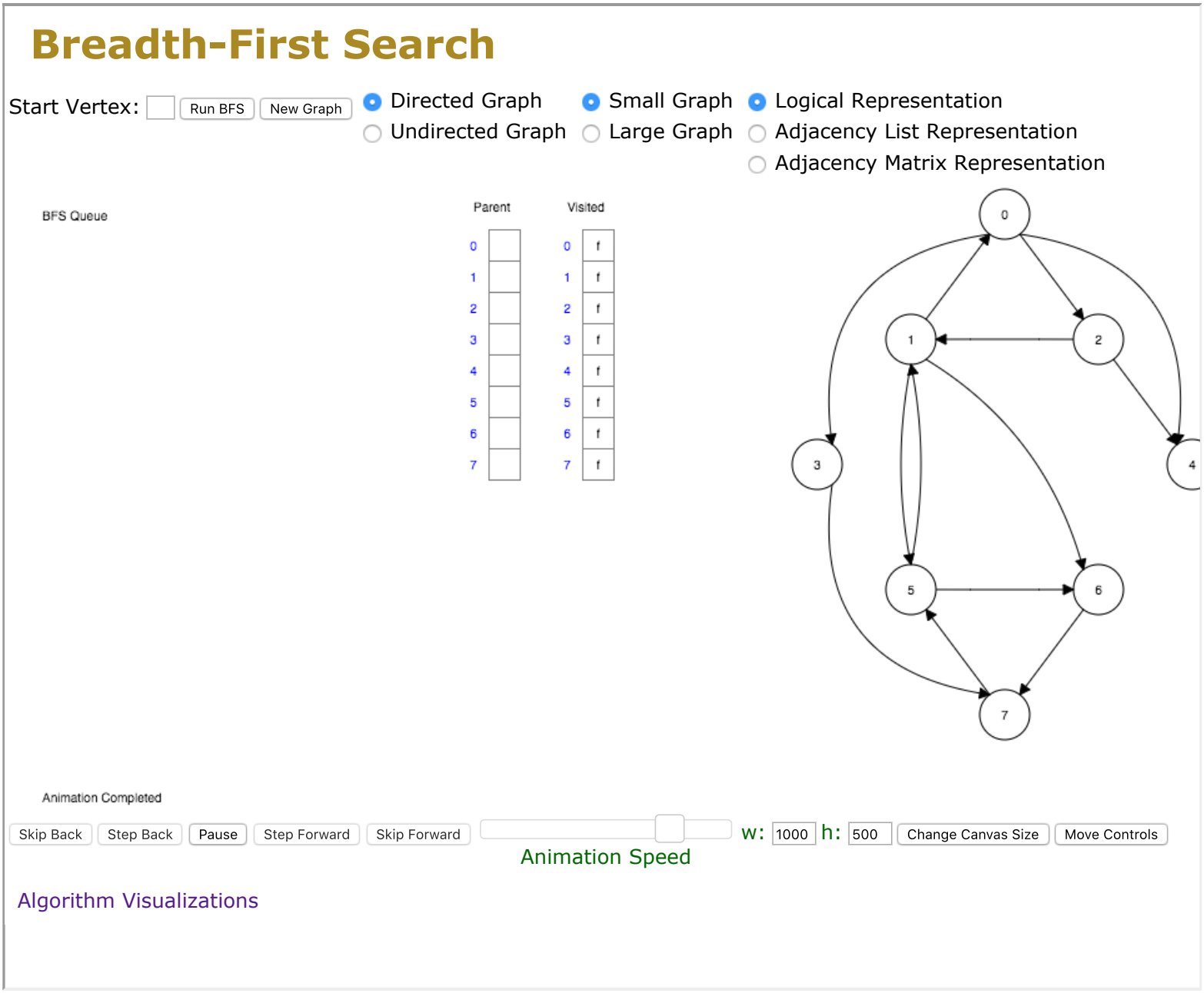
EXERCISE BREAK: Sort the vertices from top to bottom in the order that **Breadth First Search** would explore them in (the top-most vertex being the first to be explored), starting with vertex A. You should break ties in order-of-exploration by giving priority to the the vertex that contains the alphabetically smaller letter. Feel free to look back at the previous step to remind yourself of the **BFS** algorithm.



To solve this problem please visit https://stepik.org/lesson/28946/step/5

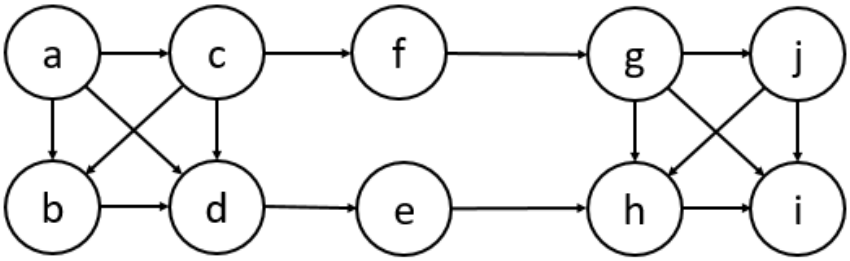
Step 6

Here is another interactive visualization, created by David Galles at the University of San Francisco, to help you further explore the BFS algorithm.



Step 7

EXERCISE BREAK: Sort the vertices from top to bottom in the order that **Breadth First Search** would explore them in (the topmost vertex being the first to be explored), starting with vertex *a*. You should break ties in order-of-exploration by giving priority to the the vertex that contains the alphabetically smaller letter.



To solve this problem please visit https://stepik.org/lesson/28946/step/7

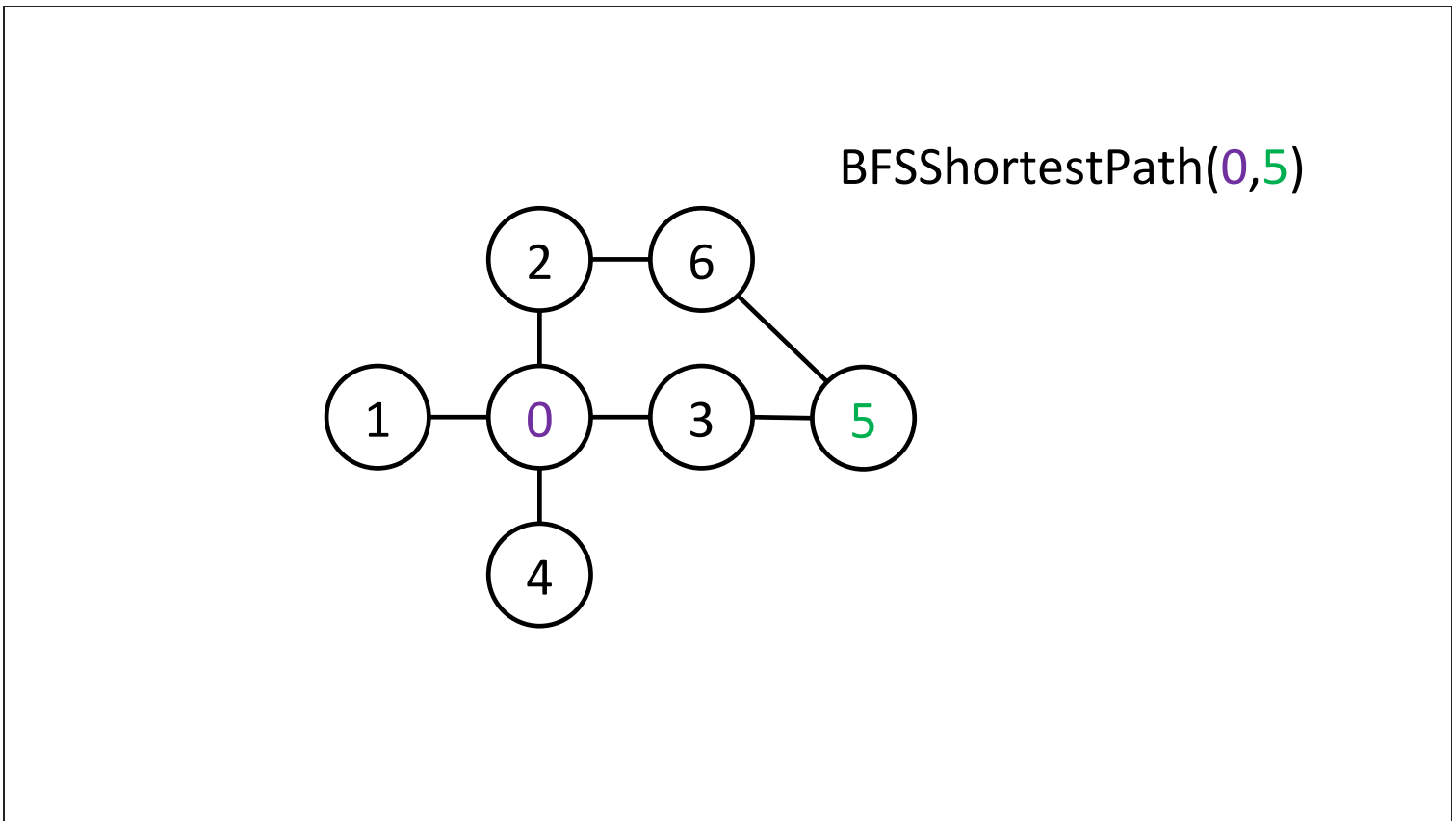
Step 8

In terms of the actual implementation of **Breadth First Search**, we take advantage of the **queue** ADT to determine the order in which to explore vertices. The intuition behind choosing a queue for **BFS** stems from a queue's *first-in first-out* property. With this property, we guarantee that we will explore all the vertices enqueued (put on the queue) first, before moving on to the vertices adjacent to the next dequeued vertex.

```
BFSShortestPath(u,v):
    q = an empty queue
    add (0,u) to q // (0,u) -> (length from u, current vertex)
    while q is not empty:
        (length,curr) = q.dequeue()
        if curr == v: // if we have reached the vertex we are searching for
            return length
        for all outgoing edges (curr,w) from curr: // otherwise explore all neighbors
            if w has not yet been visited:
                add (length+1,w) to q
    return "FAIL" // if I reach this point, then no path exists from u to v
```

Step 9

Below is a visual example of how the **Breadth First Search** algorithm takes advantage of the queue data structure to find the shortest path between two nodes. Use the arrows in the bottom left to walk through the slides.



| Slide 1 |

Slides

STOP and Think: Does the first time we encounter a vertex guarantee that we have found the shortest path to it?

Step 10

EXERCISE BREAK: We now know that **Breadth First Search** can perform well to find the shortest path between vertex *u* to *v*. But *how* well can it perform? Let's look at another pseudocode of **BFS** to derive its worst-case Big-O time complexity.

Choose the tightest time complexity (in Big-O notation) for the corresponding line of code. Note that |*V*| corresponds to the *total number of vertices* in the graph and |*E*| corresponds to the *total number of edges* in the graph.

To solve this problem please visit https://stepik.org/lesson/28946/step/10

Step 11

By adding all of the time complexities together from the previous challenge, it may seem at a first glance that we end up with an overall **O(|*V*| + |*E*|²)** time complexity. **However**, this is *not* actually the case! Take a look at the "`while the queue is not empty`" line; we see that the time complexity is **O(|*E*|)**. Now take a look at the the contents of the while loop: more specifically, the "`for each of v's adjacent nodes that have not yet been visited:`" line; we see that the time complexity is also **O(|*E*|)**. At first glance, it may seem that having an **O(|*E*|)** loop with a **O(|*E*|)** body would mean that, for *each* edge, we would need to iterate through *all* the edges (which would in fact be O(|*E*|²). **However,** if you pay attention to the details of the algorithm, you will notice that, in reality, we are merely iterating through only the adjacent edges of one vertex at a time. As a result, we might in practice iterate over some edges more than once in total (if neighbors are shared between vertexes), but we will not need to iterate through *all* edges *each* time.

Consequently, when adding the overall run-time complexities, it is common to say that the outer while loop "takes into account" the for-loop inside and as a result we can conclude that the tightest time complexity for Breadth First Search is **O(|*V*| + |*E*|)**. This makes it an efficient algorithm to find paths to vertices as we traverse layer-by-layer from our starting point.

STOP and Think: Say we have a dense graph (i.e., roughly $|V|^2$ edges) with 1,000,000,000 nodes on which we wish to perform BFS. If each edge in the queue could be represented by even just one byte, roughly how much space would the BFS queue require at its peak?

Step 12

As can be seen, **Breadth First Search** is an excellent algorithm for traversing a graph, and it can be used to solve many problems in weighted and unweighted graph problems. With regard to the example we started with, it should hopefully be clear that BFS allows us to find the shortest path (i.e., least edges) from one node to any other node in a graph.

However, hopefully the lasting question on the previous step peaked your curiosity: we mentioned earlier in this chapter that graphs can get quite massive when working with real data, so what happens if the memory requirements of BFS become too great for modern computers to be able to handle memory-wise? Is there an alternative graph traversal algorithm that might be able to save the day?