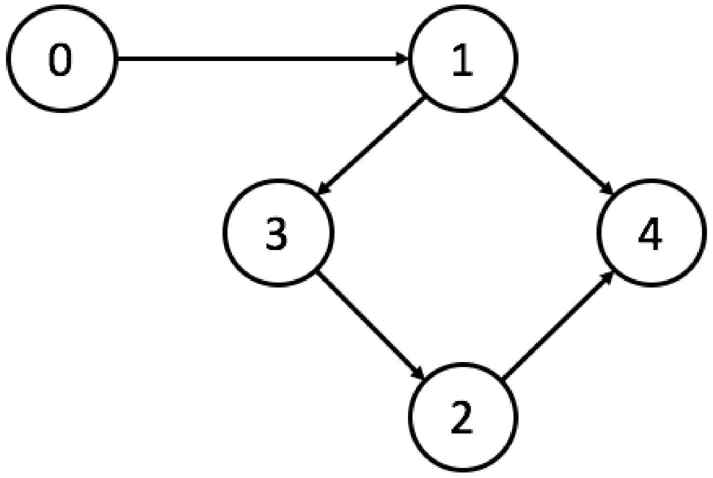
Algorithms on Graphs: Depth First Search

Step 1

Another algorithm to explore all vertices in a graph is called **Depth First Search (DFS)**. The idea behind **DFS** is to explore all the vertices going down from the current vertex, before moving on to the next neighboring vertex. Let's look at the same graph we looked at when we introduced BFS in the last lesson:



Suppose we start at 0. **DFS** will then choose to "visit" 1, since it is immediately reachable from 0. From 1, **DFS** can choose to "visit" either 3 *or* 4 since they are *both* immediately reachable from 1. Suppose **DFS** chooses to visit 3; after, **DFS** will "visit" 2 as it reachable from 3. **DFS** will finish by visiting 4 since it is immediately reachable from 2. Notice that the **DFS** algorithm chooses to explore 4 last, as opposed to 2 like in BFS.

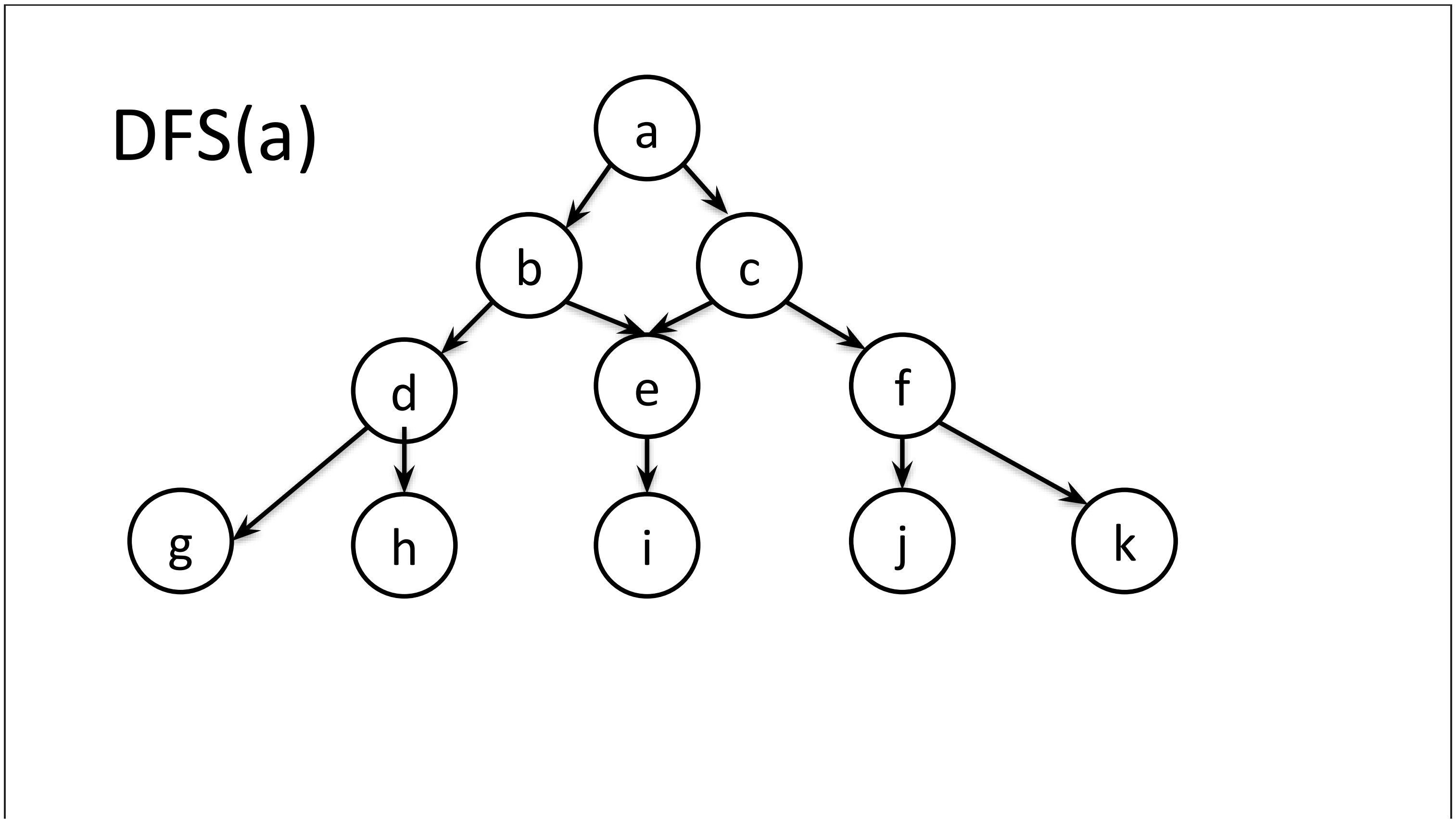
It is important to note that **DFS** could have chosen to explore 4 instead of 3 ($0 \rightarrow 1 \rightarrow 4$). You may notice that, in this case, 4 would not have any more immediately reachable vertex to explore, yet we would still have some unexplored vertices left. As a result, **DFS** would choose to explore an "unvisited" vertex reachable from the most recent explored vertex (1). In this case, DFS would then choose 3, as it is the next immediately reachable vertex from 1.

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

1. Start at *s*. It has distance 0 from itself.
2. Consider a node adjacent to *s*. Call it *t*. It has distance 1. Mark it as visited.
3. Then consider a node adjacent to *t* that has not yet been visited. It has distance 2. Mark it as visited.
4. Etc. etc. until all nodes reachable from *s* are visited.

Step 2

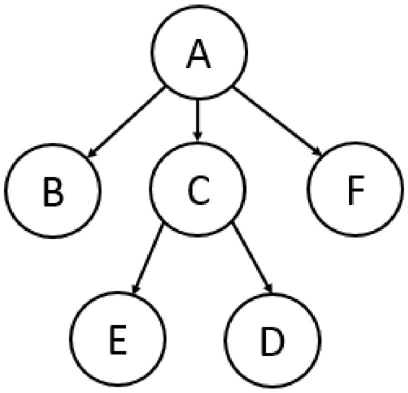
Depth First Search can be difficult to understand without being able to actually see the algorithm in action. As a result, we've included a visualization below that shows the order in which each vertex is explored. Use the arrows in the bottom left to walk through the slides.



| Slide 1 | Slides

Step 3

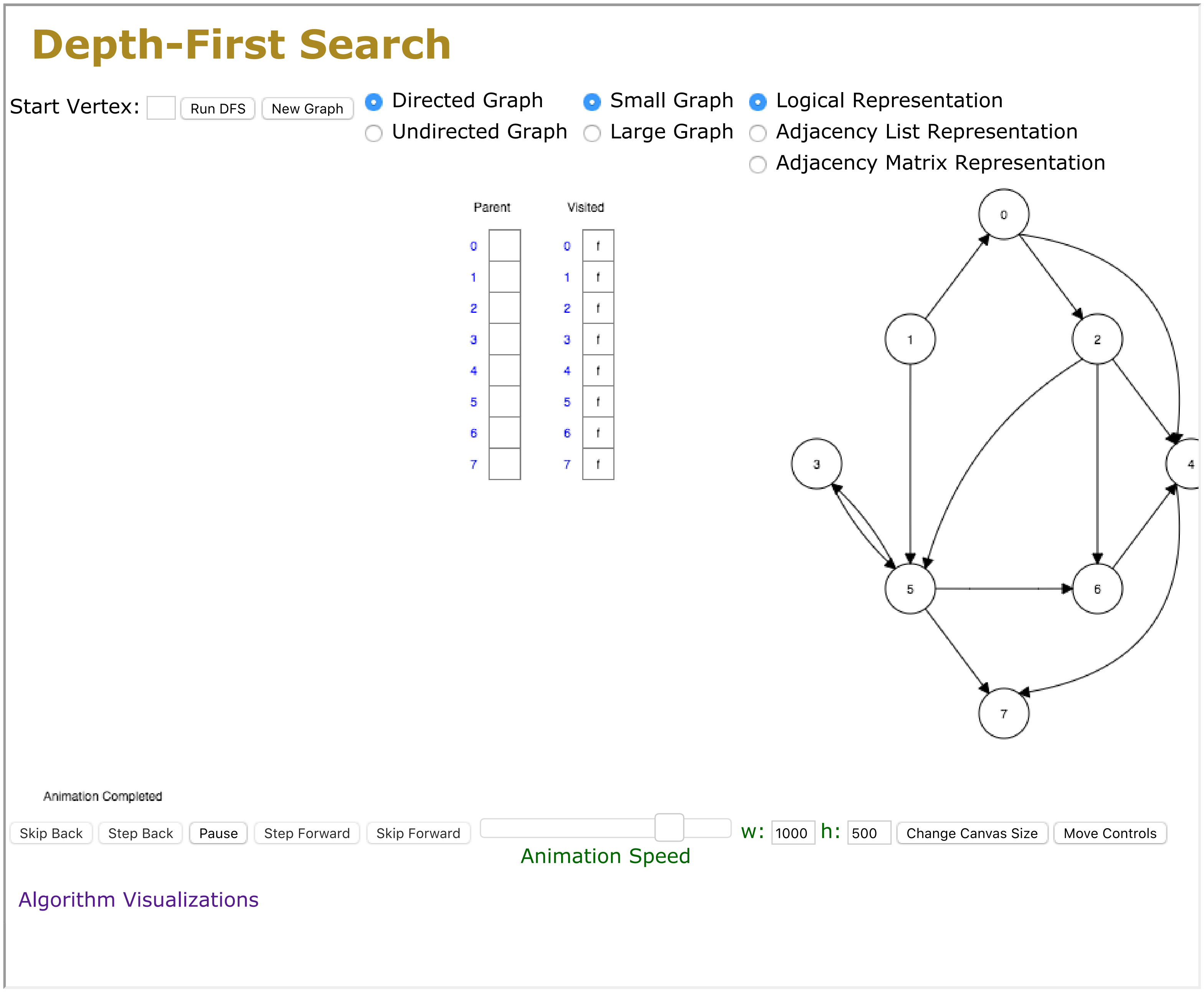
EXERCISE BREAK: Sort the vertices from top to bottom in the order that **Depth First Search** would explore them in (the top most vertex being the first to be explored), starting with vertex a. You should break ties in order-of-exploration by giving priority to the the vertex that contains the alphabetically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



To solve this problem please visit https://stepik.org/lesson/29037/step/3

Step 4

To better understand the behavior of **Depth First Search** algorithm, here is a visualization created by David Galles at the University of San Francisco.



[✓ Directed Graph] Undirected Graph

[✓ Small Graph] Large Graph

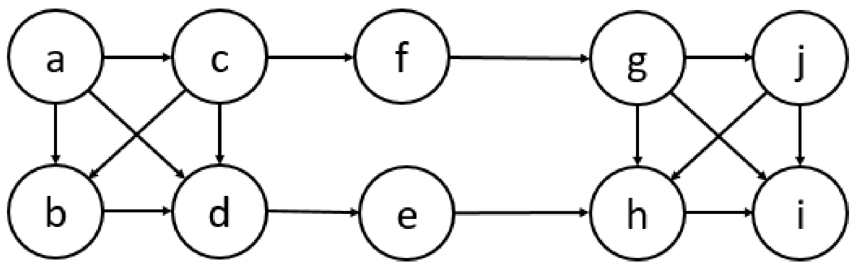
[✓ Logical Representation] Adjacency List Representation Adjacency Matrix Representation

w: 1000 h: 500

	Parent		Visited
0		0	f
1		1	f
2		2	f
3		3	f
4		4	f
5		5	f
6		6	f
7		7	f

Step 5

EXERCISE BREAK: Sort the vertices from top to bottom in the order that **Depth 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.



To solve this problem please visit https://stepik.org/lesson/29037/step/5

Step 6

Just like BFS took advantage of the queue ADT to explore the vertices in a particular order, **Depth First Search** takes advantage of the **stack** ADT. The intuition behind choosing a stack for **DFS** stems from a stack's *first-in last-out* property. With this property, we guarantee that we will explore all the vertices pushed (put on the stack) most recently first, before moving on to vertices that were adjacent to a vertex from before.

```
DFSShortestPath(u,v):
    s = an empty stack
    push (0,u) to s // (0,u) -> (length from u, current vertex)
    while s is not empty:
        (length,curr) = s.pop()
        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 s
    return "FAIL" // if I reach this point, then no path exists from u to v
```

Notice that the only change in the algorithm from the BFS algorithm we provided earlier is that all mentions of "queue" have been turned into "stack." (Isn't it crazy how the replacement of one data structure can create an entirely new algorithm?)

Step 7

Also, it is cool to note that the nature of Depth First Search's exploration makes implementing the algorithm recursively absurdly simple. Take a look at this pseudocode:

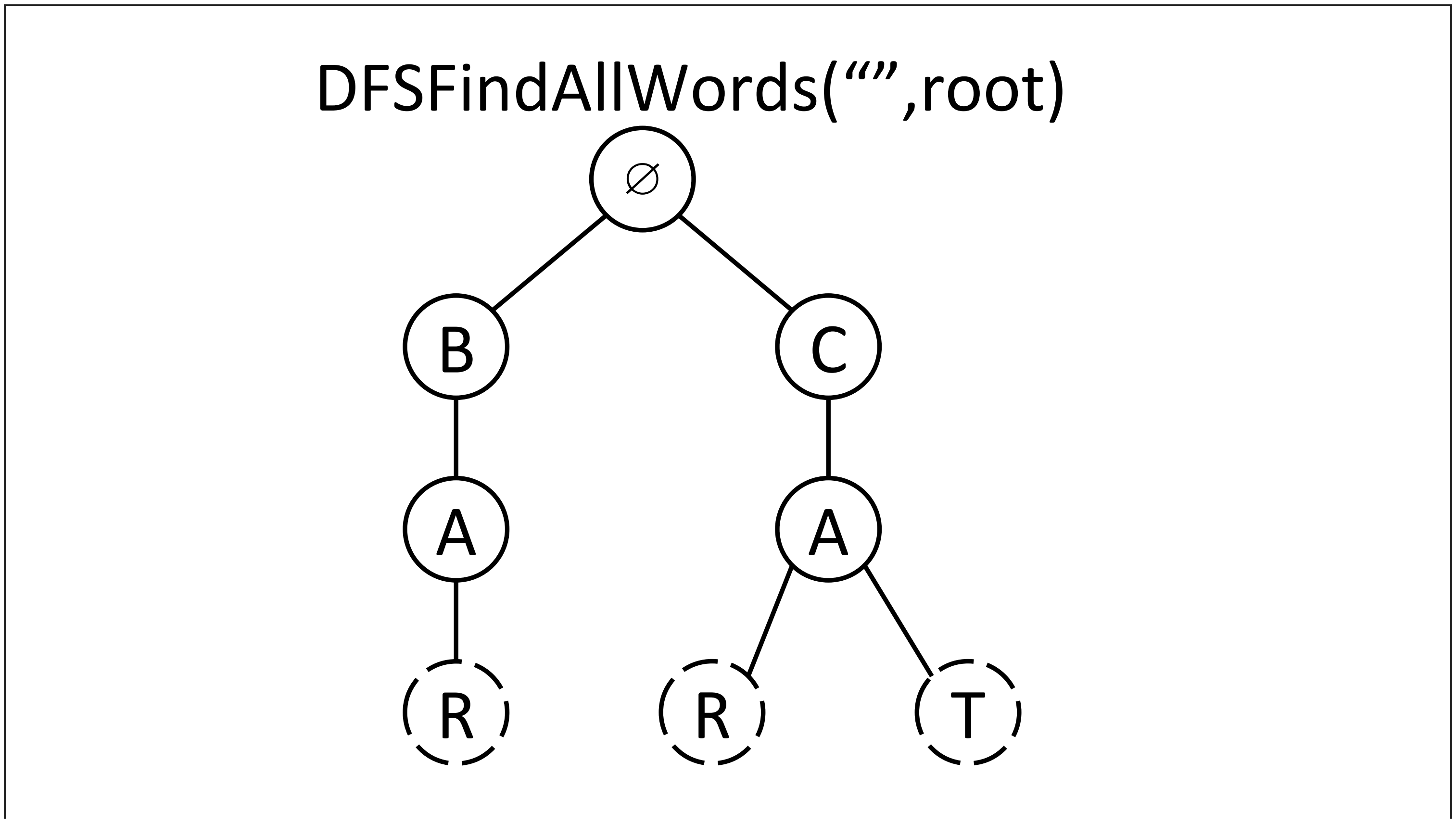
```
DFSRecursion(s): //where s is the starting vertex
    mark s as explored
    for each unexplored neighbor v of s:
        DFSRecursion(v)
```

Notice that declaring/using a stack is not needed since the recursive nature of the algorithm provides the stack implementation for us.

STOP and Think: It is important to note that there does not exist such a simple recursive pseudocode for Breath First Search; why might this be the case?

Step 8

Below is another visual example of the **Depth First Search** algorithm. Here we take advantage of the algorithm's nature (the fact that it uses a stack) to easily find words within a graph starting from the root. Use the arrows in the bottom left to walk through the slides.



| **Slide 1** |

Slides

Step 9

EXERCISE BREAK: As we by now have seen, **DFS** seems to behave very similarly to **BFS**. Consequently, how do their time complexities compare?

Hopefully you should remember that the only difference in the pseudocode for **BFS** and **DFS** was their use of a queue and stack, respectively. As a result, your intuition should tell you that we only need to compare the run times for those two abstract data types to see if we expect their run times to differ or stay the same.

Review Question: What is the worst-case time complexity of the three operations of a stack: top, pop, and push? Select all that apply. (Multiple operations can have the same worst-case time complexity)

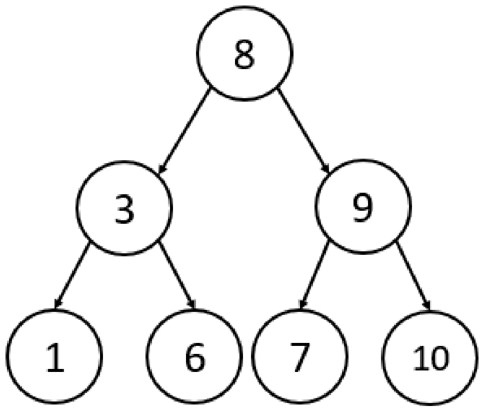
To solve this problem please visit https://stepik.org/lesson/29037/step/9

Step 10

From the last step we see that the worst case time complexities of both a queue and a stack are the same. As a result, we can intuitively come to the correct conclusion that both DFS and BFS have a worst case time complexity of O(**|V|** +**|E|**).

Notice however that though their time complexity is the same, their memory management is totally different! In other words, to traverse the graph, BFS had to store all of a vertex's neighbors in each step in a queue, to later be able to traverse those neighbors. On the other hand, if you take a look at the recursive implementation of DFS, DFS only needed to store the previous neighbor to explore the next one. As a result, for certain graph structures, DFS can be more memory efficient.

For example, take a look at this graph below, which also happens to be a Binary Search Tree:



For **DFS**, there will be a certain time in the algorithm where it will need to store in memory a single strand of vertices going *down* the tree (i.e. vertices 8-3-1 will be stored in memory all at once).

For **BFS**, there will be a certain time in the algorithm where it will need to store in memory a single strand of vertices going *across* the tree (i.e. vertices 1-6-7-10 will be stored in memory all at once).

Even within this simple example, we already see that **BFS** is beginning to need more space to perform. Now imagine if the tree grows larger? The width of the tree will begin to grow exponentially larger than the height of the tree and as a result, **BFS** will start to require an exponentially more amount of memory! As a result, in practice, **BFS** can begin to slow down in performance because of all the extra memory accesses it will need to perform.

Now that we have seen two very solid algorithms that are used to traverse graphs, we will begin to take a look at their potential algorithmic drawbacks in the next lesson.