21 Depth-First Search Written by Irina Galata

In the previous chapter, you looked at breadth-first search (BFS) in which you had to explore every neighbor of a vertex before going to the next level. In this chapter, you'll look at **depth-first search (DFS)**, another algorithm for traversing or searching a graph.

There are many applications for DFS:

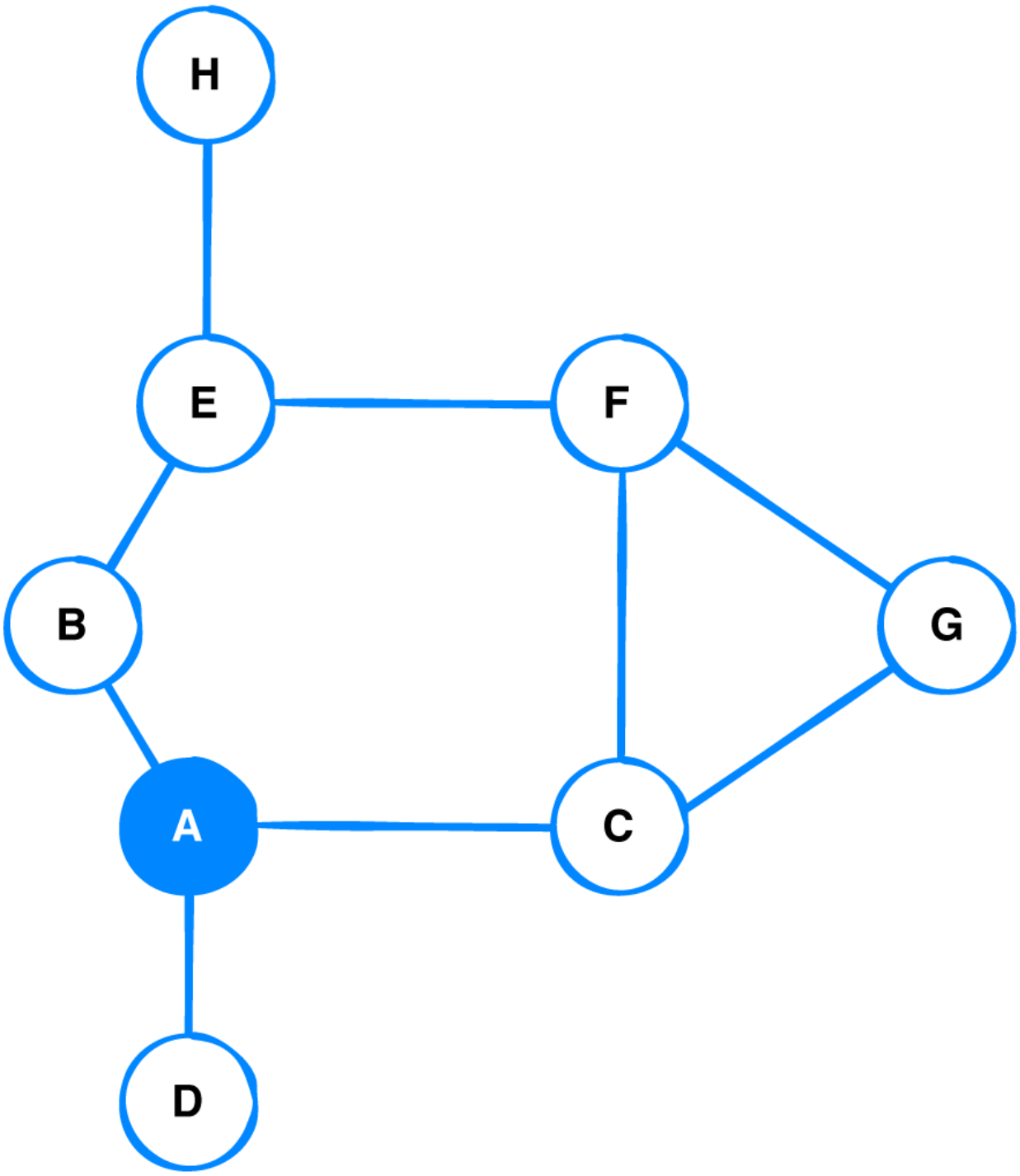
- Topological sorting.
- Detecting a cycle.
- Pathfinding, such as in maze puzzles.
- Finding connected components in a sparse graph.

To perform a DFS, you start with a given source vertex and attempt to explore a branch as far as possible until you reach the end. At this point, you'd backtrack (move a step back) and explore the next available branch until you find what you're looking for or until you've visited all the vertices.

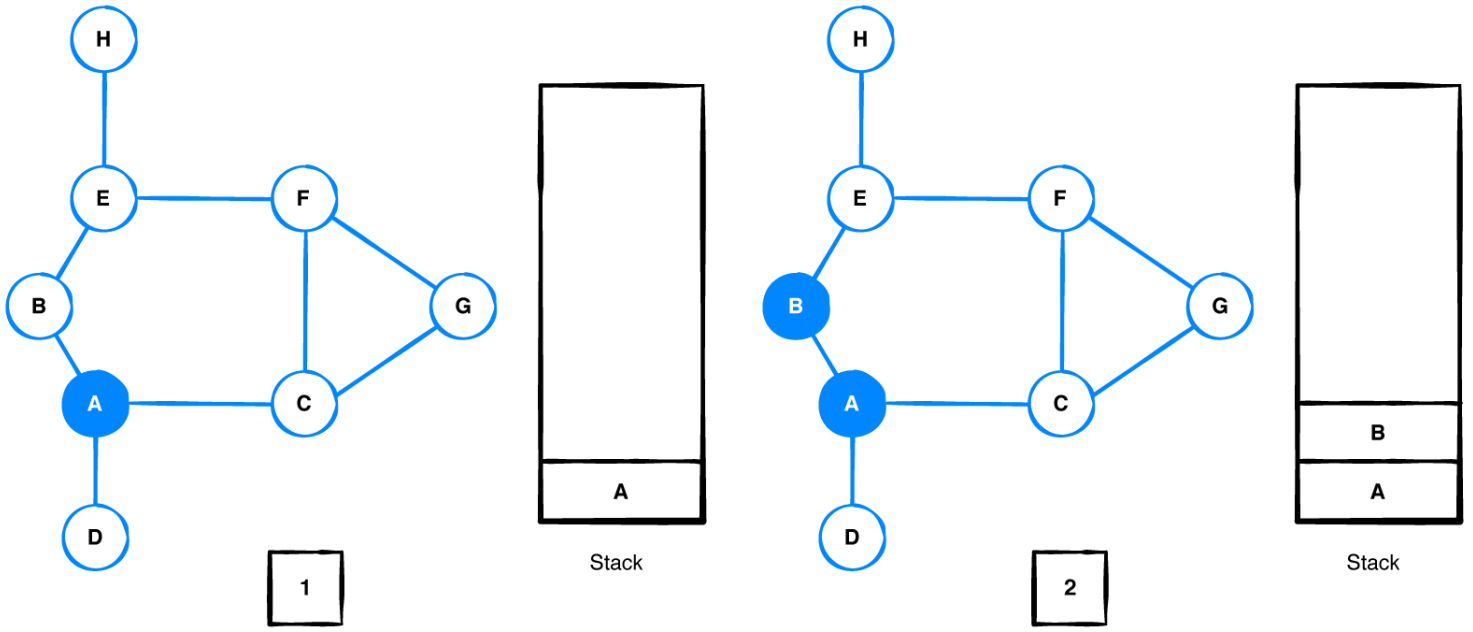
DFS example

The example graph below is the same as the previous chapter.

Using the same graph helps you to see the difference between BFS and DFS.



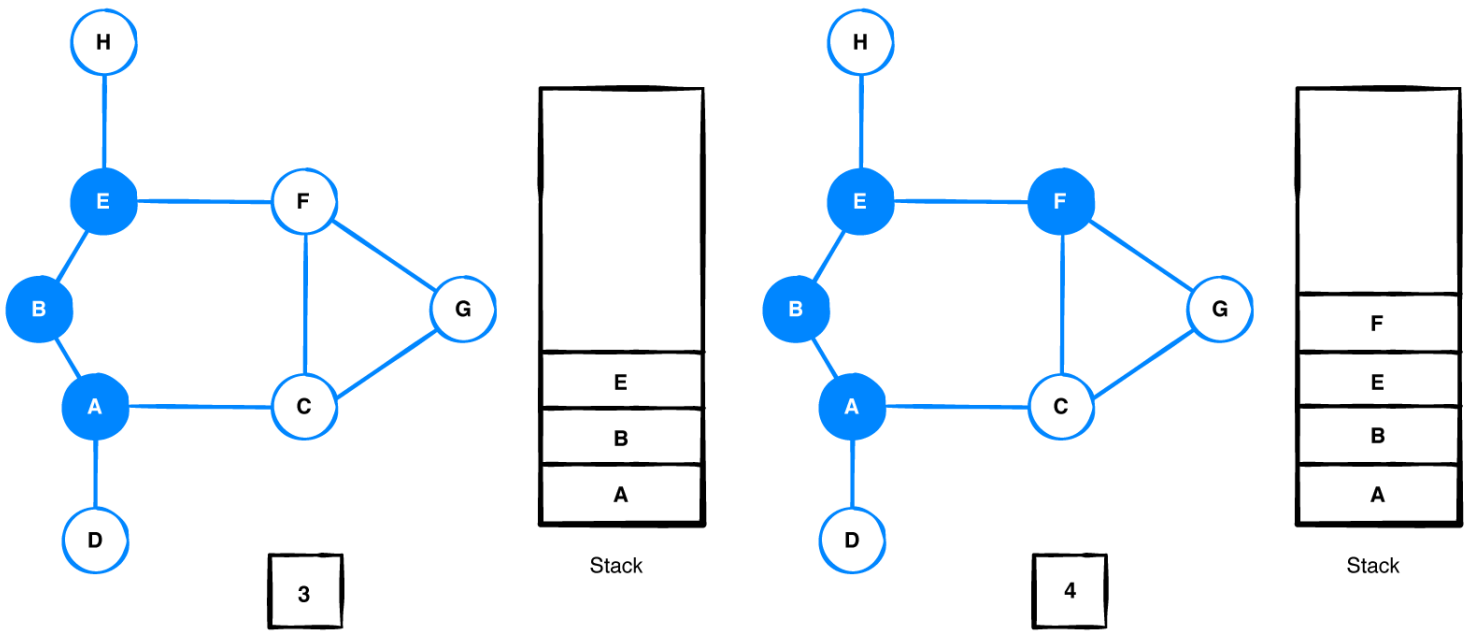
You'll use a stack to keep track of the levels you move through. The stack's LIFO approach helps with backtracking. Every push on the stack means that you move one level deeper. When you reach a dead end, you can pop to return to a previous level.



1. As in the previous chapter, you choose `A` as a starting vertex and add it to the stack.

2. As long as the stack is not empty, you visit the top vertex on the stack and push the first neighboring vertex that you haven't yet visited. In this case, you visit `A` and push `B`.

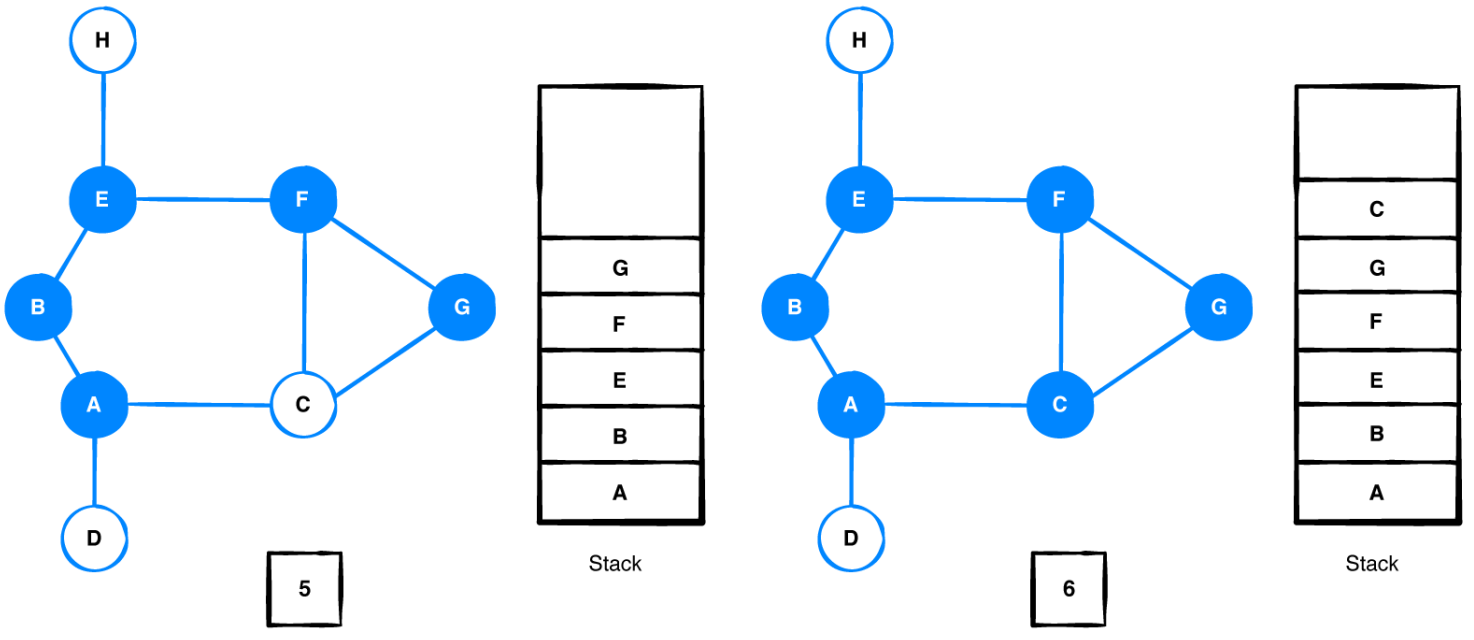
Recall from the previous chapter that the order in which you add edges influences the result of a search. In this case, the first edge added to `A` was an edge to `B`, so `B` is pushed first.



3. You visit `B` and push `E` because you already visited `A`.

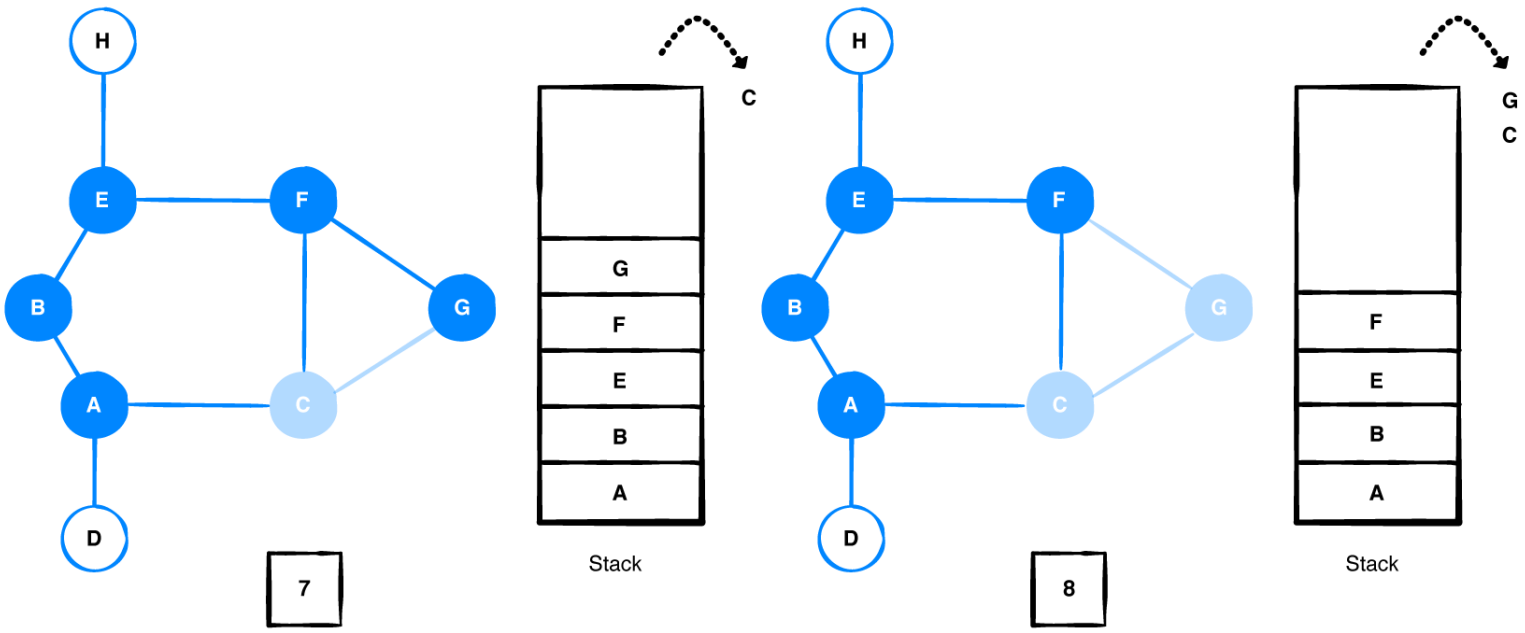
4. You visit `E` and push `F`.

Note that, every time you push on the stack, you advance farther down a branch. Instead of visiting every adjacent vertex, you continue down a path until you reach the end and then backtrack.



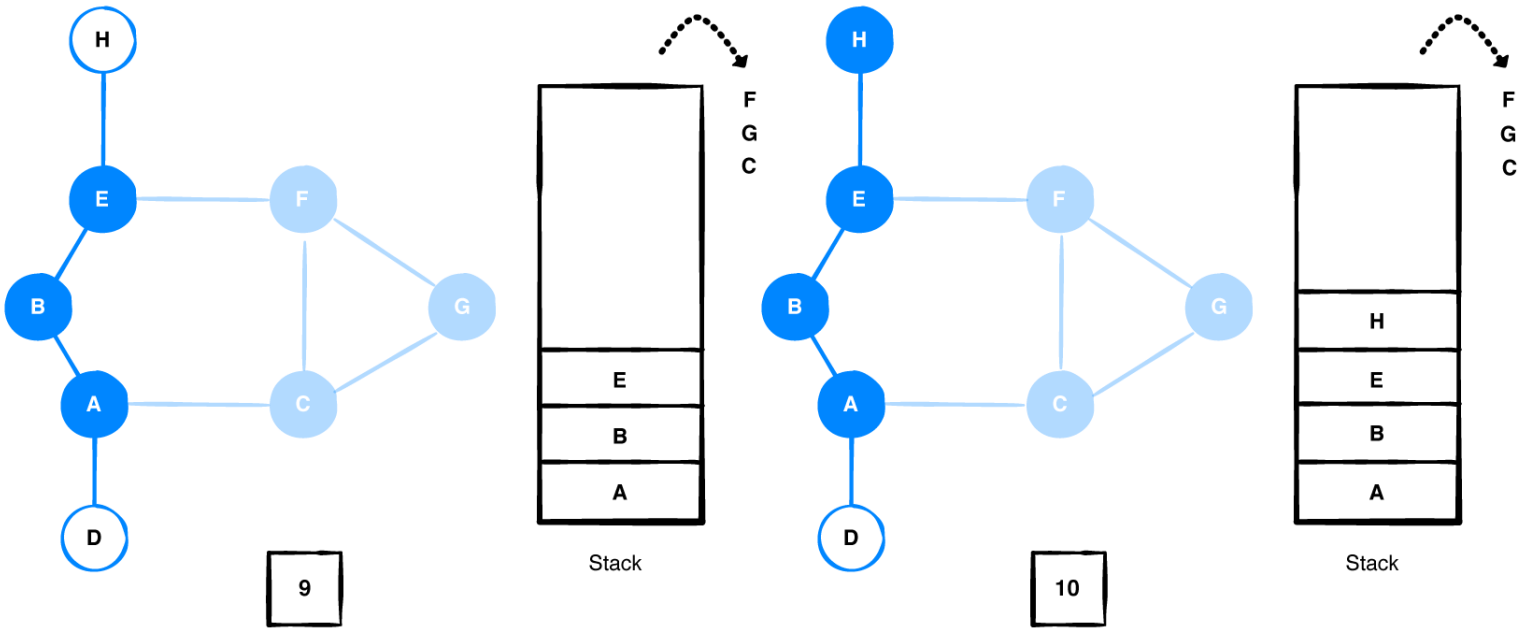
5. You visit `F` and push `G`.

6. You visit `G` and push `C`.



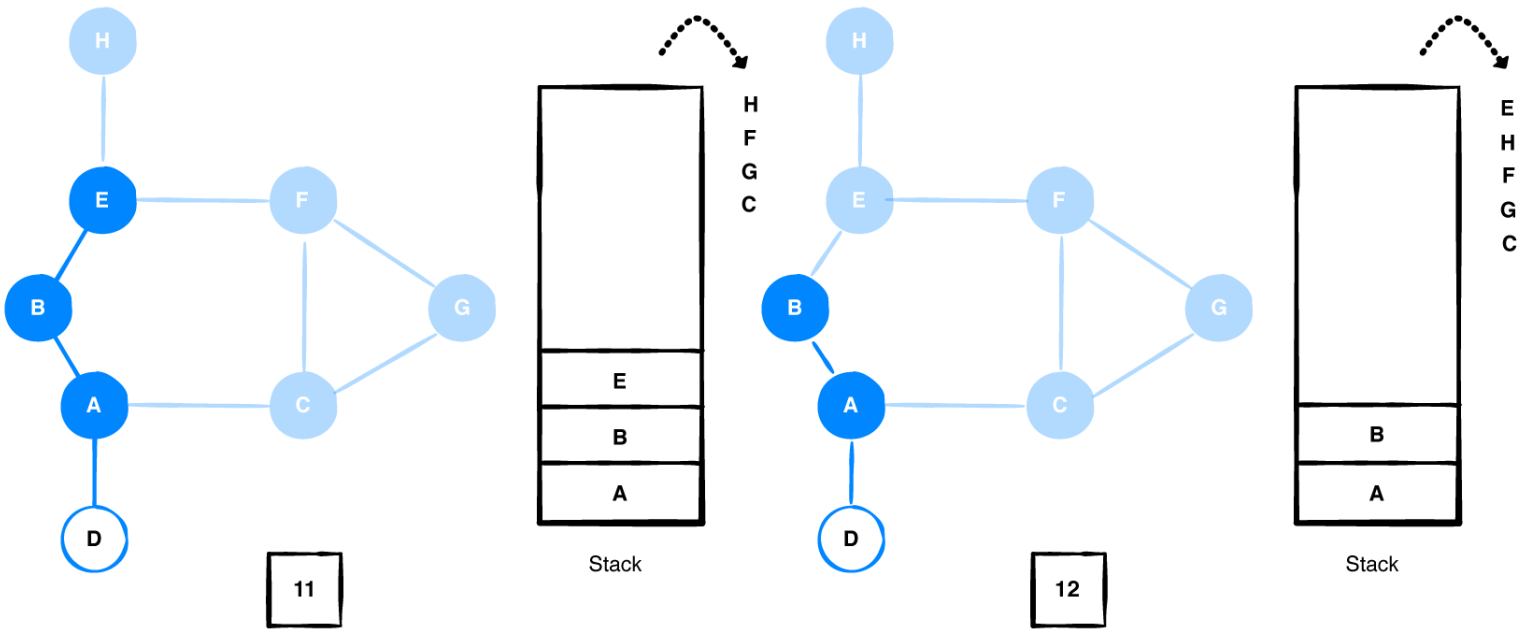
7. The next vertex to visit is `C`. It has neighbors `[A, F, G]`, but you already visited these. You reached a dead end, so it's time to backtrack by popping `C` off the stack.

8. This brings you back to `G`. It has neighbors `[F, C]`, but you also already visited these. Another dead end, pop `G`.



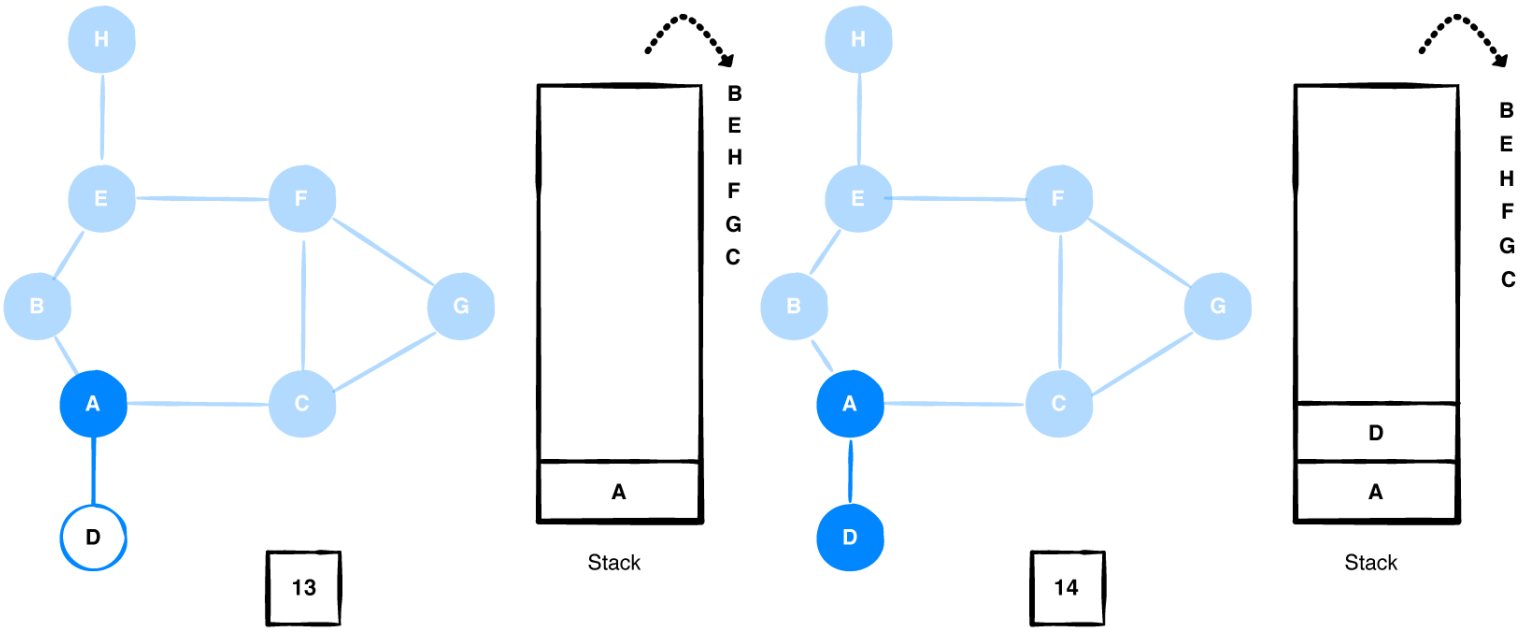
9. `F` also has no unvisited neighbors remaining, so pop `F`.

10. Now, you're back at `E`. Its neighbor `H` is still unvisited, so you push `H` on the stack.



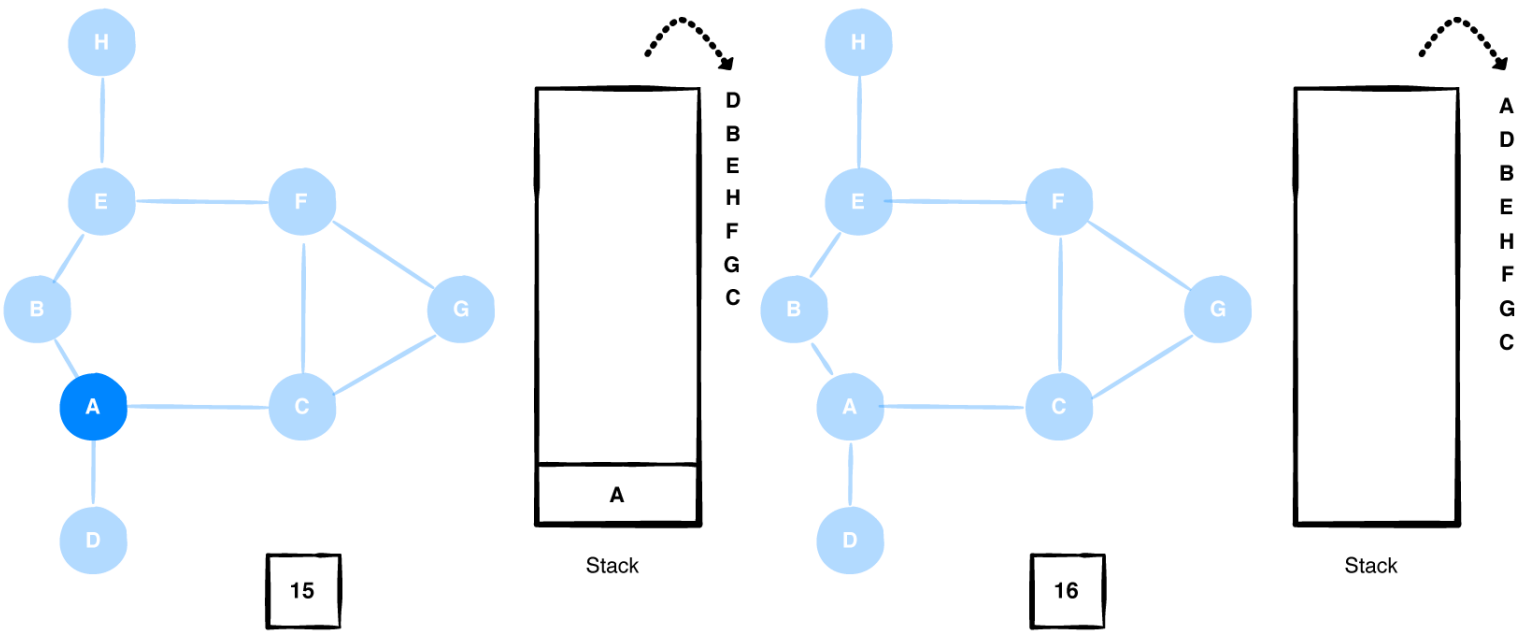
11. Visiting `H` results in another dead end, so pop `H`.

12. `E` also doesn't have any available neighbors, so pop it.



13. The same is true for `B`, so pop `B`.

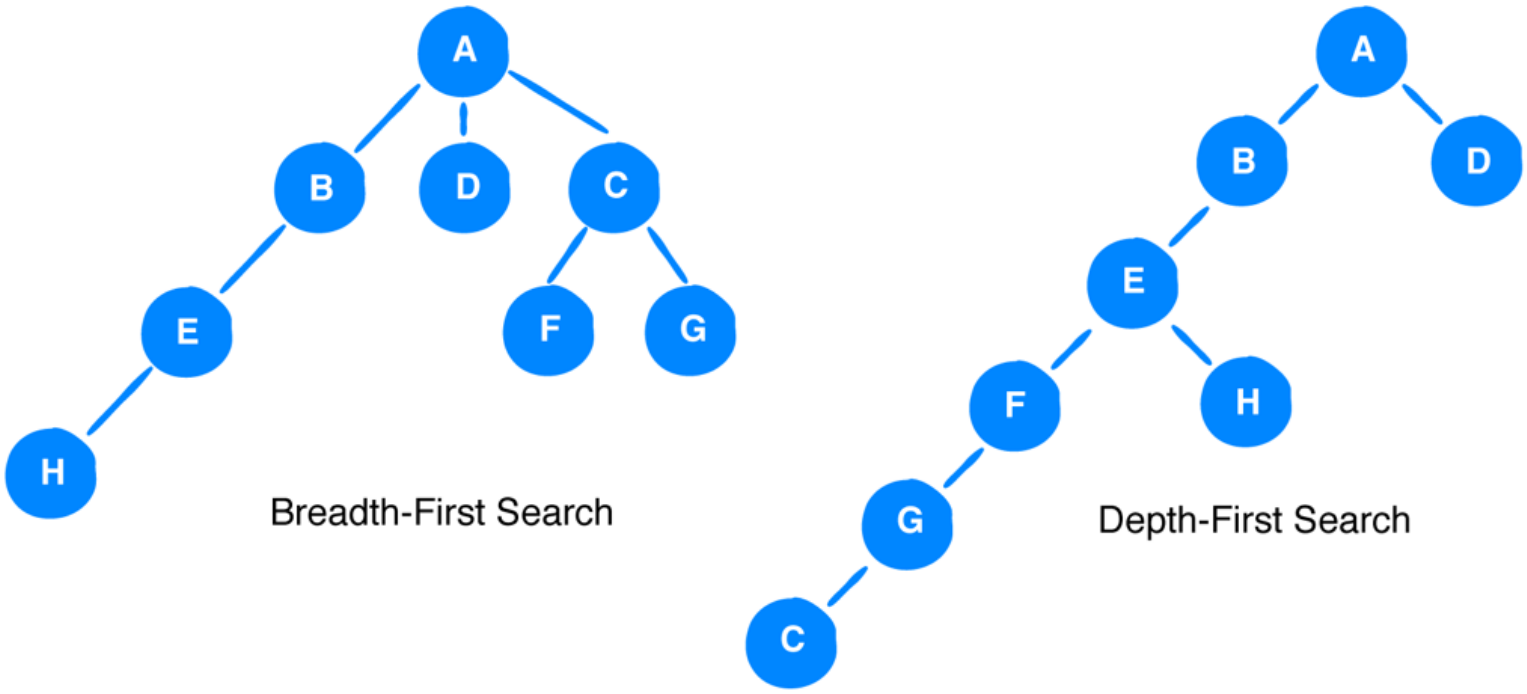
14. This brings you back to `A`, whose neighbor `D` still needs a visit, so you push `D` on the stack.



15. Visiting `D` results in another dead end, so pop `D`.

16. You're back at `A`, but this time, there are no available neighbors to push, so you pop `A`. The stack is now empty, and the DFS is complete.

When exploring the vertices, you can construct a tree-like structure, showing the branches you've visited. You can see how deep DFS went when compared to BFS.



Implementation

Open the starter project for this chapter. This project contains an implementation of a graph, as well as a stack that you'll use to implement DFS.

Look at **Main.kt**, and you'll see a pre-built sample graph. This is the graph you'll be working with.

To implement DFS, add the following inside `Graph`:

```
fun depthFirstSearch(source: Vertex<T>): ArrayList<Vertex<T>> {
  val stack = StackImpl<Vertex<T>>()
  val visited = arrayListOf<Vertex<T>>()
  val pushed = mutableSetOf<Vertex<T>>()

  stack.push(source)
  pushed.add(source)
  visited.add(source)

  // more to come ...

  return visited
}
```

With this code you define `depthFirstSearch()`, a new method that takes in a starting vertex and returns a list of vertices in the order they were visited. It uses three data structures:

1. **`stack`**: Used to store your path through the graph.
2. **`pushed`**: Remembers which vertices were already pushed so that you don't visit the same vertex twice. It's a `MutableSet` to ensure fast *O*(1) lookup.
3. **`visited`**: A list that stores the order in which the vertices were visited.

In the first step you insert the vertex passed as parameter to the three data structures. You do this because this is the first to be visited and it's the starting point in order to navigate the neighbors.

Next, complete the method by replacing the comment with:

```
outer@ while (true) {
  if (stack.isEmpty) break

  val vertex = stack.peek()!! // 1
  val neighbors = edges(vertex) // 2

  if (neighbors.isEmpty()) { // 3
    stack.pop()
    continue
  }

  for (i in 0 until neighbors.size) { // 4
    val destination = neighbors[i].destination
    if (destination !in pushed) {
      stack.push(destination)
      pushed.add(destination)
      visited.add(destination)
      continue@outer // 5
    }
  }
  stack.pop() // 6
}
```

Here's what's going on:

1. You continue to check the top of the stack for a vertex until the stack is empty. You've labeled this loop `outer` so that you have a way to continue to the next vertex, even within nested loops.
2. You find all the neighboring edges for the current vertex.
3. If there are no edges, you pop the vertex off the stack and continue to the next one.
4. Here, you loop through every edge connected to the current vertex and check to see if the neighboring vertex has been seen. If not, you push it onto the stack and add it to the `visited` list.

It may seem a bit premature to mark this vertex as visited — you haven't popped it yet — but since vertices are visited in the order in which they are added to the stack, it results in the correct order. 5. Now that you've found a neighbor to visit, you continue the `outer` loop and move to the newly pushed neighbor. 6. If the current vertex did not have any unvisited neighbors, you know that you reached a dead end and can pop it off the stack.

Once the stack is empty, the DFS algorithm is complete. All you have to do is return the visited vertices in the order you visited them.

To test your code, add the following to `main()`:

```
val vertices = graph.depthFirstSearch(a)
vertices.forEach {
  println(it.data)
}
```

If you run the `main()` in the **Main.kt** file, you'll see the following output for the order of the visited nodes using a DFS:

```
A
```

```
B
E
H
F
C
G
D
```

Performance

DFS visits every vertex at least once. This has a time complexity of *O*(V).

When traversing a graph in DFS, you have to check all neighboring vertices to find one that's available to visit. The time complexity of this is *O*(E) because in the worst case, you have to visit every edge in the graph.

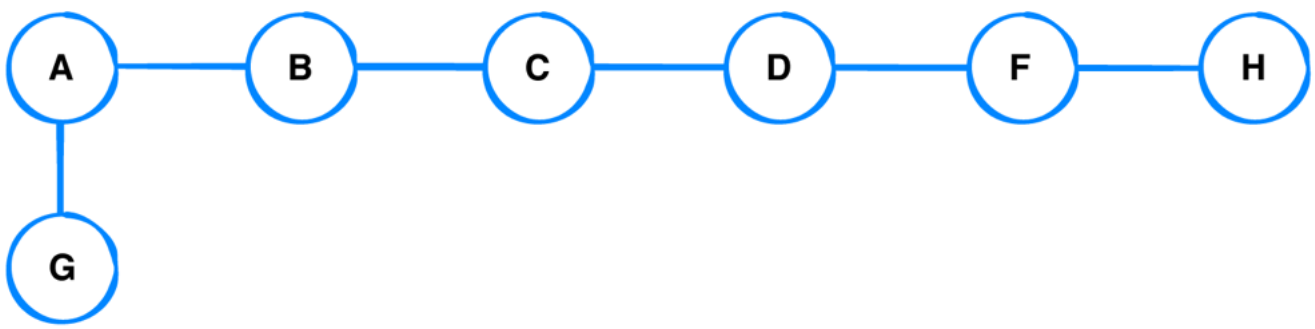
Overall, the time complexity for depth-first search is *O*(V + E).

The space complexity of depth-first search is *O*(V) because you have to store vertices in three separate data structures: `stack`, `pushed` and `visited`.

Challenges

Challenge 1: Depth First Search

For each of the following two examples, which traversal (depth-first or breadth-first) is better for discovering if a path exists between the two nodes? Explain why.



- Path from **A** to **F**.
- Path from **A** to **G**.

Solution 1

- Path from **A** to **F**: Use depth-first, because the path you're looking for is deeper in the graph.
- Path from **A** to **G**: Use breadth-first, because the path you're looking for is near the root.

Challenge 2: Depth First Search

In this chapter, you went over an iterative implementation of depth-first search. Write a recursive implementation.

Solution 2

Let's look at how you can implement DFS recursively.

```
fun depthFirstSearchRecursive(start: Vertex<T>): ArrayList<Vertex<T>> {
  val visited = arrayListOf<Vertex<T>>() // 1
  val pushed = mutableSetOf<Vertex<T>>() // 2

  depthFirstSearch(start, visited, pushed) // 3

  return visited
}
```

Here's what's happening:

1. `visited` keeps track of the vertices visited in order.
2. `pushed` keeps tracks of which vertices have been visited.
3. Perform depth-first search recursively by calling a helper function.

The helper function looks like this:

```
fun depthFirstSearch(
```

```
    source: Vertex<T>,
    visited: ArrayList<Vertex<T>>,
    pushed: MutableSet<Vertex<T>>
) {
  pushed.add(source) // 1
  visited.add(source)

  val neighbors = edges(source)
  neighbors.forEach { // 2
    if (it.destination !in pushed) {
      depthFirstSearch(it.destination, visited, pushed) // 3
    }
  }
}
```

Here's how it works:

1. Insert the `source` vertex into the queue, and mark it as visited.
2. For every neighboring edge...
3. As long as the adjacent vertex has not been visited yet, continue to dive deeper down the branch recursively.

Overall, the time complexity for depth-first search is *O*(V + E).

Challenge 3: Depth First Search Challenge

Add a method to `Graph` to detect if a **directed** graph has a cycle.

Solution 3

A graph is said to have a cycle when there's a path of edges and vertices leading back to the same source.

```
fun hasCycle(source: Vertex<T>): Boolean {
  val pushed = arrayListOf<Vertex<T>>() // 1
  return hasCycle(source, pushed) // 2
}
```

Here's how it works:

1. `pushed` is used to keep track of all the vertices visited.
2. Recursively check to see if there's a cycle in the graph by calling a helper function.

The helper function looks like this:

```
fun hasCycle(source: Vertex<T>, pushed: MutableSet<Vertex<T>>): Boolean {
  pushed.add(source) // 1

  val neighbors = edges(source) // 2
  neighbors.forEach {
    if (it.destination !in pushed && hasCycle(it.destination, pushed)) { //
      return true
    } else if (it.destination in pushed) { // 4
      return true
    }
  }
  pushed.remove(source) // 5
  return false // 6
}
```

And it works like this:

1. To initiate the algorithm, first, insert the source vertex.
2. For every neighboring edge...
3. If the adjacent vertex has not been visited before, recursively dive deeper down a branch to check for a cycle.
4. If the adjacent vertex has been visited before, you've found a cycle.
5. Remove the `source` vertex so that you can continue to find other paths with a potential cycle.
6. No cycle has been found.

You're essentially performing a depth-first graph traversal by recursively diving down one path until you find a cycle. You're backtracking by

popping off the stack to find another path. The time-complexity is $O(V + E)$.

Key points

- Depth-first search (DFS) is another algorithm to traverse or search a graph.
- DFS explores a branch as far as possible until it reaches the end.
- Leverage a stack data structure to keep track of how deep you are in the graph. Only pop off the stack when you reach a dead end.

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.