20 Breadth-First Search Written by Irina Galata

In the previous chapter, you explored how you can use graphs to capture relationships between objects. Remember that objects are vertices, and the relationships between them are represented by edges.

Several algorithms exist to traverse or search through a graph's vertices. One such algorithm is the **breadth-first search** (BFS) algorithm.

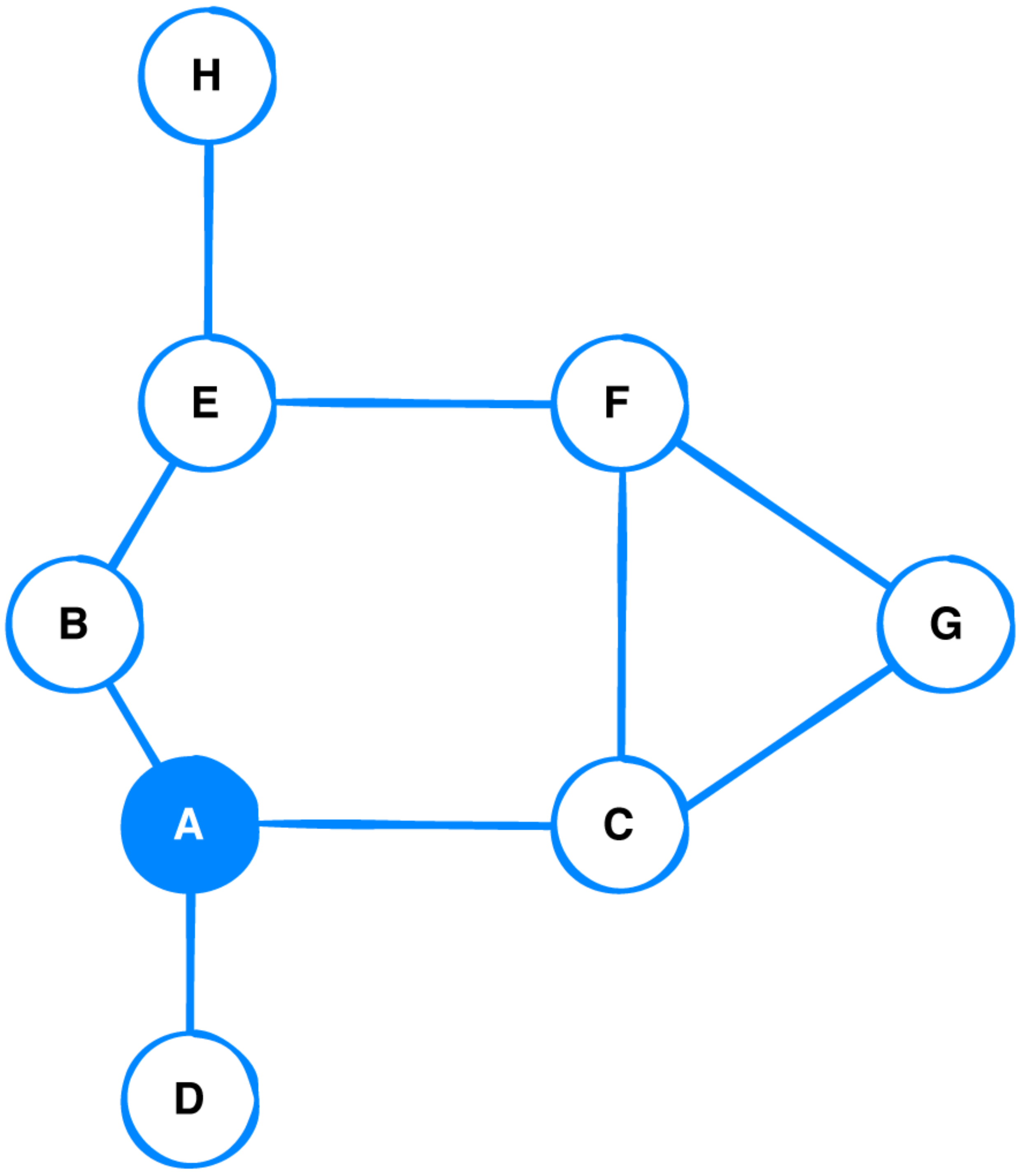
You can use BFS to solve a wide variety of problems:

1. Generating a minimum-spanning tree.
2. Finding potential paths between vertices.
3. Finding the shortest path between two vertices.

Example

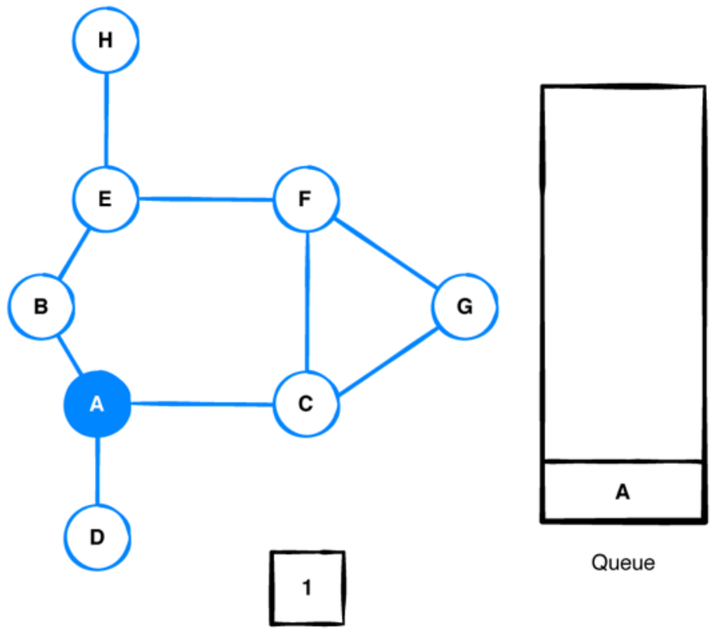
BFS starts by selecting any vertex in a graph. The algorithm then explores all neighbors of this vertex before traversing the neighbors of said neighbors and so forth. As the name suggests, this algorithm takes a **breadth-first** approach because it doesn't visit the children until all the siblings are visited.

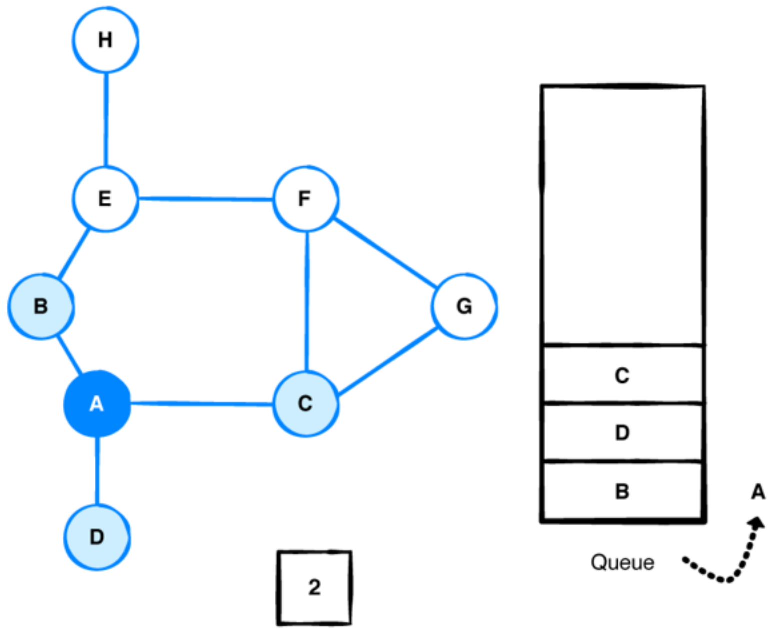
To get a better idea of how things work, you'll look at a BFS example using the following undirected graph:



> **Note**: Highlighted vertices represent vertices that have been visited.

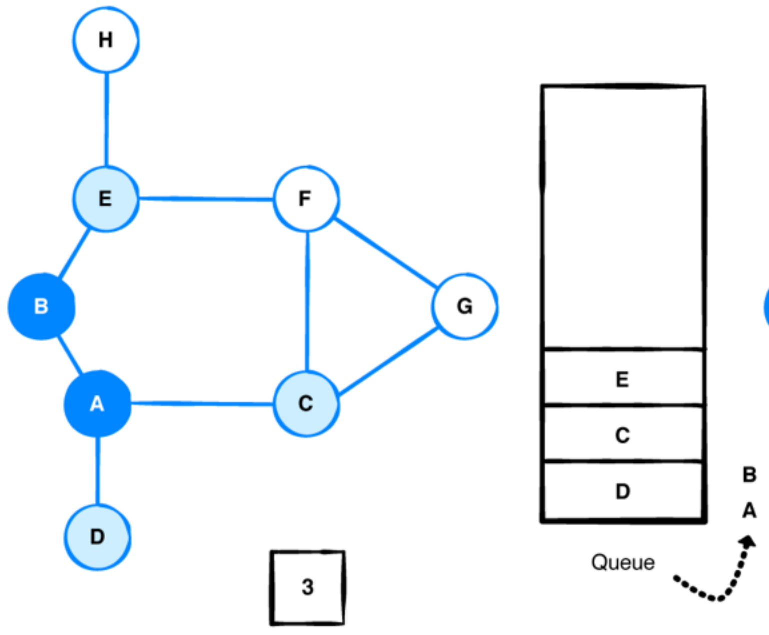
To keep track of which vertices to visit next, you can use a queue. The first in, first out approach of the queue guarantees that all of a vertex's neighbors are visited before you traverse one level deeper.

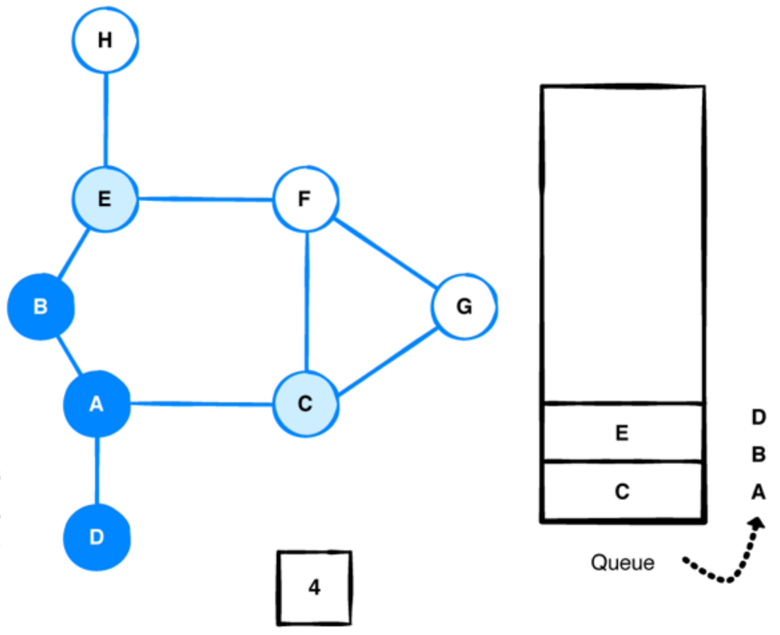




1. To begin, pick a source vertex to start from. In this example, you choose `A`, which is added to the queue.
2. As long as the queue is not empty, you can dequeue and visit the next vertex, in this case, `A`. Next, you need to add all `A`'s neighboring vertices `[B, D, C]` to the queue.

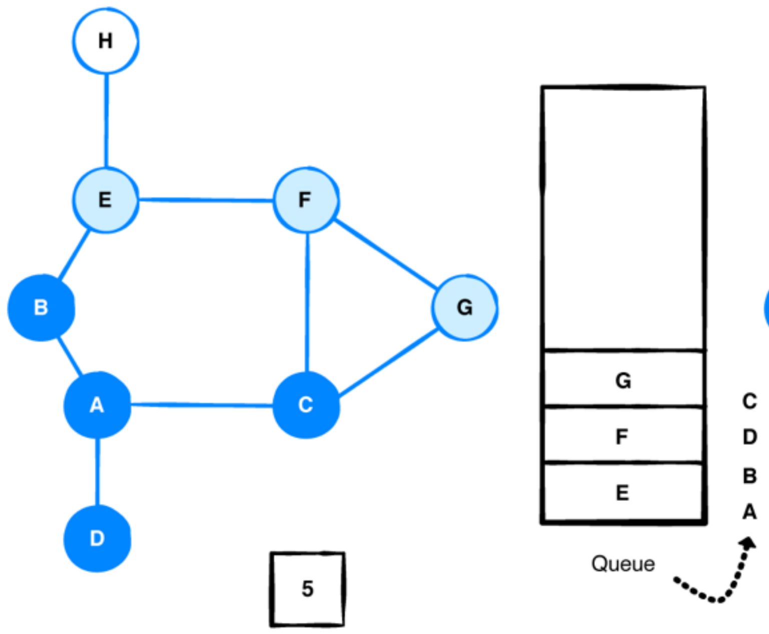
> **Note**: You only add a vertex to the queue when it has not yet been visited and is not already in the queue.

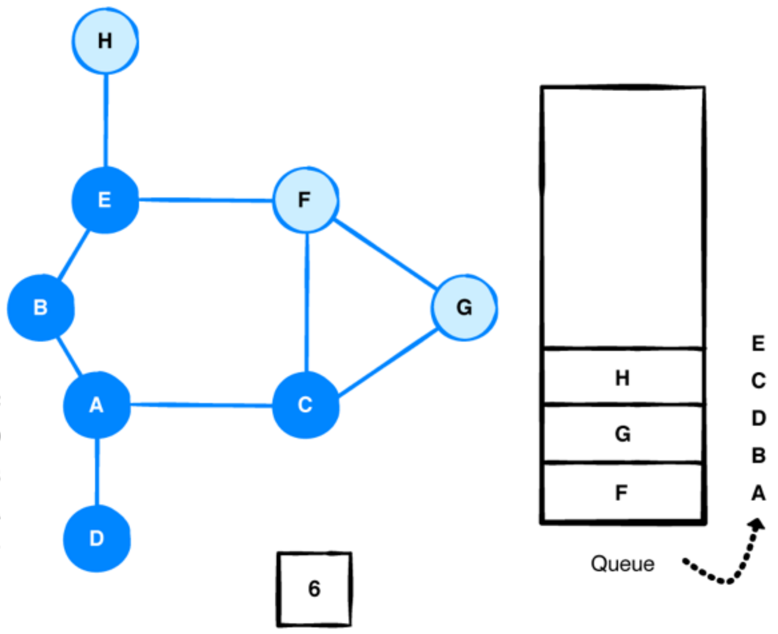




3. The queue is not empty, so you need to dequeue and visit the next vertex, which is `B`. You then need to add `B`'s neighbor `E` to the queue. `A` is already visited, so it doesn't get added. The queue now has `[D, C, E]`.

4. The next vertex to be dequeued is `D`. `D` does not have any neighbors that aren't visited. The queue now has `[C, E]`.

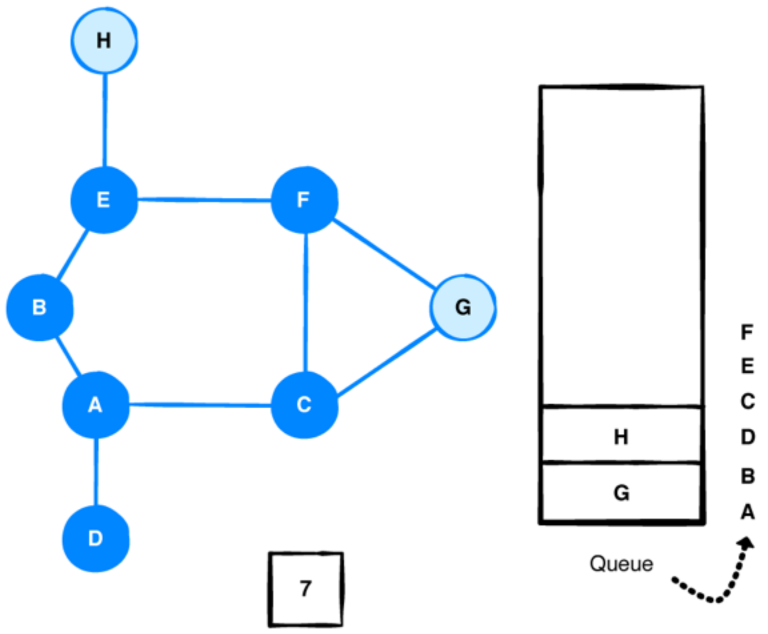


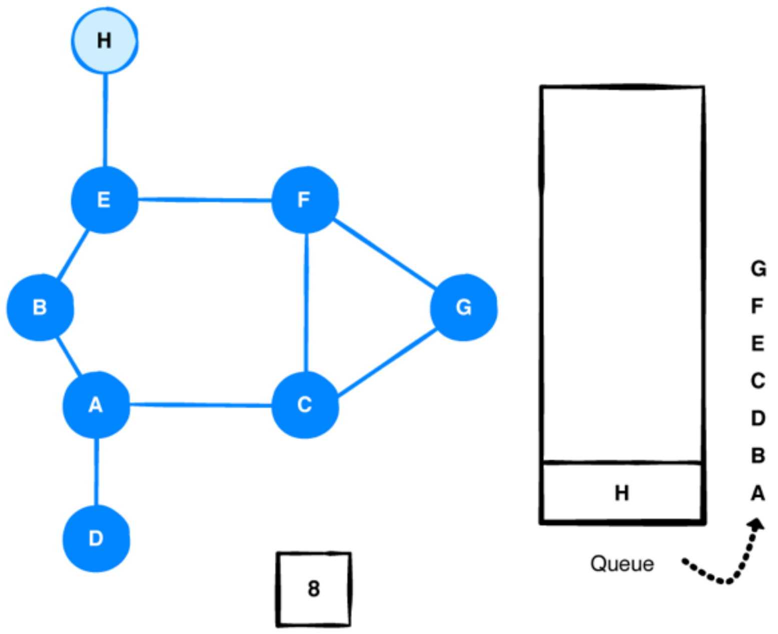


5. Next, you need to dequeue `C` and add its neighbors `[F, G]` to the queue. The queue now has `[E, F, G]`.

Note that you have now visited all of `A`'s neighbors, and BFS moves on to the second level of neighbors.

6. You need to dequeue `E` and add `H` to the queue. The queue now has `[F, G, H]`. Note that you don't add `B` or `F` to the queue because `B` is already visited and `F` is already in the queue.

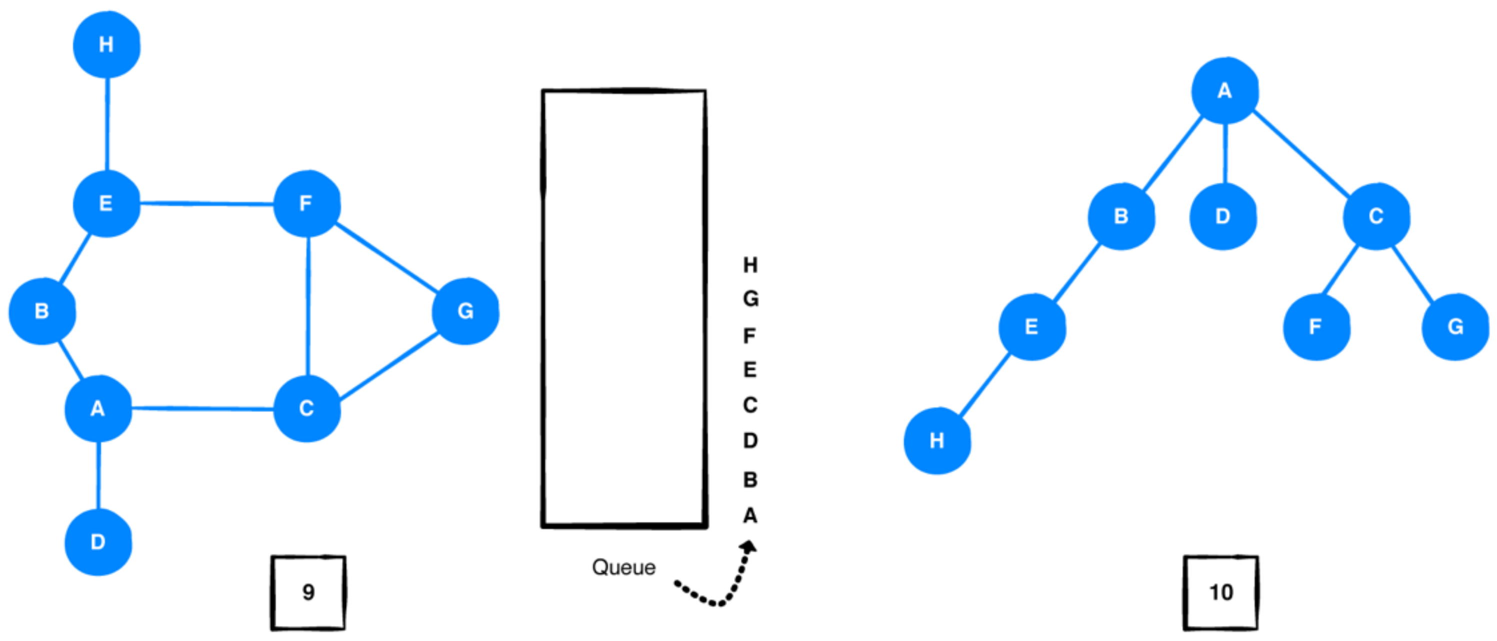




7. You need to dequeue `F`, and since all of its neighbors are already in

the queue or visited, you don’t need to add anything to the queue.

8. Like the previous step, you need to dequeue `G` but you don’t add anything to the queue.



9. Finally, you need to dequeue `H`. The breadth-first search is complete since the queue is now empty.

10. When exploring the vertices, you can construct a tree-like structure, showing the vertices at each level: First the vertex you started from, then its neighbors, then its neighbors’ neighbors and so on.

Implementation

Open the starter project for this chapter. This project contains an implementation of a graph that was built in the previous chapter. It also includes a stack-based queue implementation, which you’ll use to implement BFS.

In your starter project, you’ll notice **Graph.kt**. Add the following method to the `Graph` class:

```
fun breadthFirstSearch(source: Vertex<T>): ArrayList<Vertex<T>> {
  val queue = QueueStack<Vertex<T>>()
  val enqueued = mutableSetOf<Vertex<T>>()
  val visited = ArrayList<Vertex<T>>()
  // more to come ...

  return visited
}
```

Here, you defined the method `breadthFirstSearch()` which takes in a starting vertex. It uses three data structures:

1. **`queue`**: Keeps track of the neighboring vertices to visit next.
2. **`enqueued`**: Remembers which vertices have been enqueued, so you don't enqueue the same vertex twice.
3. **`visited`**: An array list that stores the order in which the vertices were explored.

Next, complete the method by replacing the comment with:

```
queue.enqueue(source) // 1
enqueued.add(source)

while (true) {
  val vertex = queue.dequeue() ?: break // 2

  visited.add(vertex) // 3

  val neighborEdges = edges(vertex) // 4
  neighborEdges.forEach {
    if (!enqueued.contains(it.destination)) { // 5
      queue.enqueue(it.destination)
      enqueued.add(it.destination)
    }
  }
}
```

Here's what's going on:

1. You initiate the BFS algorithm by first enqueuing the `source` vertex.

2. You continue to dequeue a vertex from the queue until the queue is empty.
3. Every time you dequeue a vertex from the queue, you add it to the list of visited vertices.
4. You then find all edges that start from the current vertex and iterate over them.
5. For each edge, you check to see if its destination vertex has been enqueued before, and if not, you add it to the code.

That's all there is to implementing BFS. It's time to give this algorithm a spin. Add the following code to the `main` method in the **Main.kt** file:

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

Take note of the order of the explored vertices using BFS:

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

One thing to keep in mind with neighboring vertices is that the order in which you visit them is determined by how you construct your graph. You could have added an edge between `A` and `C` before adding one between `A` and `B`. In this case, the output would list `C` before `B`.

Performance

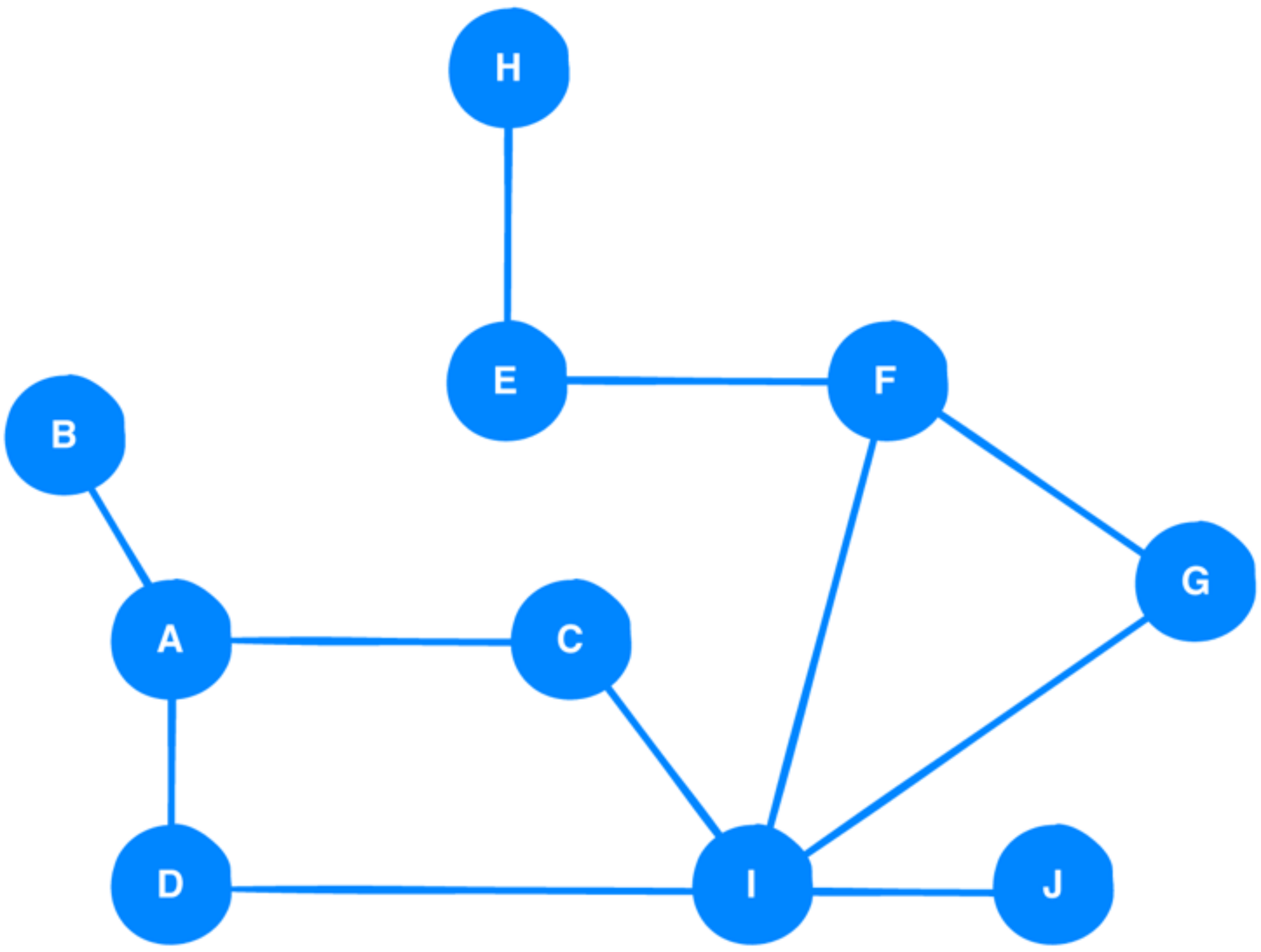
When traversing a graph using BFS, each vertex is enqueued once. This has a time complexity of *O*(V). During this traversal, you also visit all of the edges. The time it takes to visit all edges is *O*(E). This means that the overall time complexity for breadth-first search is *O*(V + E).

The space complexity of BFS is *O*(V) since you have to store the vertices in three separate structures: `queue`, `enqueued` and `visited`.

Challenges

Challenge 1: How many nodes?

For the following undirected graph, list the **maximum** number of items ever in the queue. Assume that the starting vertex is **A**.



Solution 1

The maximum number of items ever in the queue is **3**.

Challenge 2: What about recursion?

In this chapter, you went over an iterative implementation of breadth-first search. Now, write a recursive implementation.

Solution 2

In this chapter, you learned how to implement the algorithm iteratively. Let's look at how you would implement it recursively.

```
fun bfs(source: Vertex<T>): ArrayList<Vertex<T>> {
  val queue = QueueStack<Vertex<T>>() // 1
  val enqueued = mutableSetOf<Vertex<T>>() // 2
  val visited = arrayListOf<Vertex<T>>() // 3

  queue.enqueue(source) // 4
  enqueued.add(source)

  bfs(queue, enqueued, visited) // 5

  return visited // 6
}
```

`bfs` takes in the `source` vertex to start traversing from:

1. `queue` keeps track of the neighboring vertices to visit next.
2. `enqueued` remembers which vertices have been added to the queue.
3. `visited` is a list that stores the order in which the vertices were explored.
4. Initiate the algorithm by inserting the `source` vertex.
5. Perform `bfs` recursively on the graph by calling a helper function.
6. Return the vertices visited in order.

The helper function looks like this:

```
private fun bfs(queue: QueueStack<Vertex<T>>, enqueued: MutableSet<Vertex<T
  val vertex = queue.dequeue() ?: return // 1
  visited.add(vertex) // 2

  val neighborEdges = edges(vertex) // 3
  neighborEdges.forEach {
    if (!enqueued.contains(it.destination)) { // 4
      queue.enqueue(it.destination)
      enqueued.add(it.destination)
    }
  }

  bfs(queue, enqueued, visited) // 5
}
```

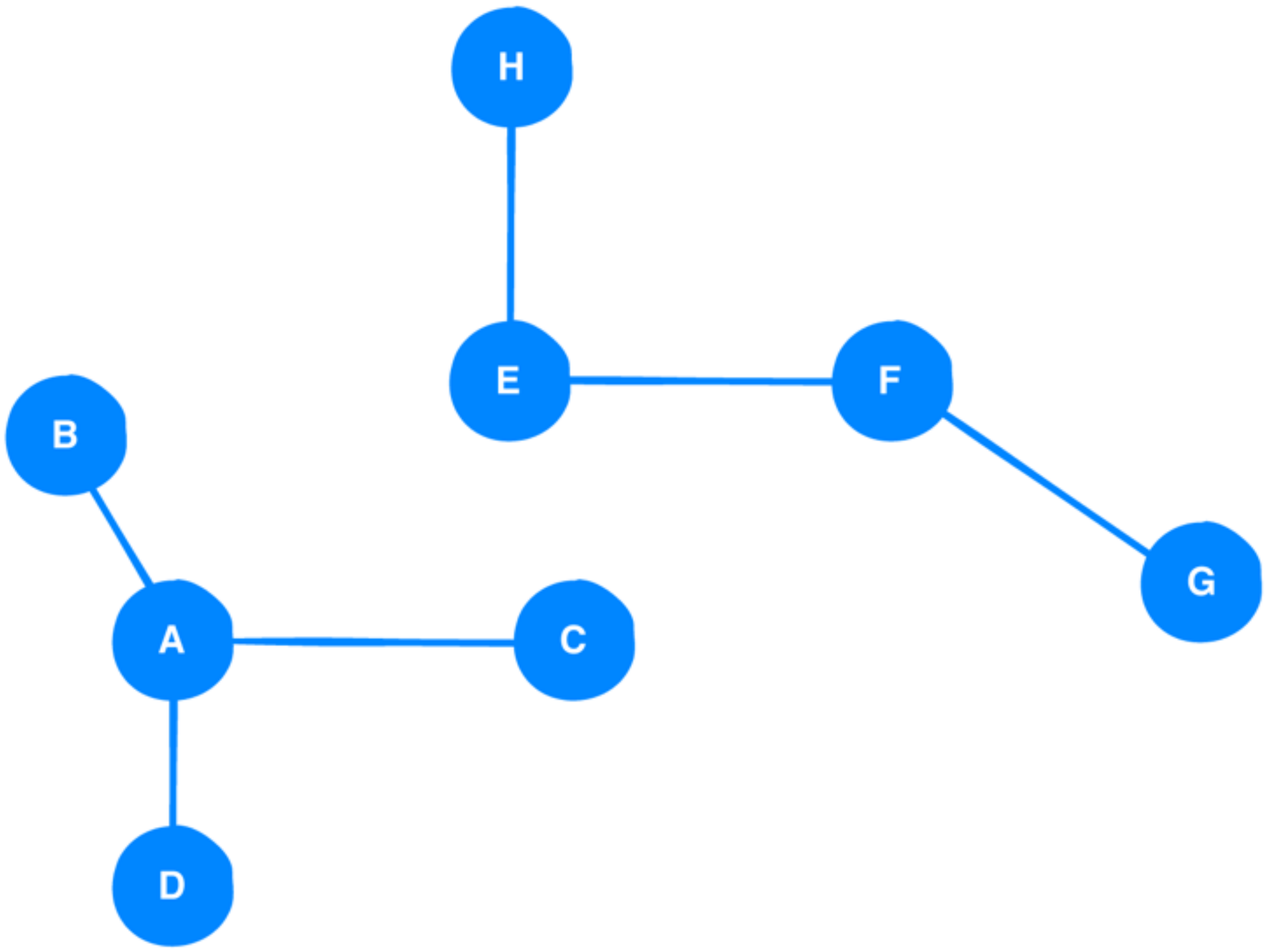
Here's how it works:

1. We start from the first node we dequeue from the queue of all vertices. Then we recursively continue to dequeue a vertex from the queue till it's empty.
2. Mark the vertex as visited.
3. For every neighboring edge from the current `vertex`.
4. Check to see if the adjacent vertices have been visited before inserting into the queue.
5. Recursively perform `bfs` until the queue is empty.

The overall time complexity for breadth-first search is *O*(V + E).

Challenge 3: Detect disconnects

Add a method to `Graph` to detect if a graph is disconnected. An example of a disconnected graph is shown below:



Solution 3

To solve this challenge, add the property `allVertices` to the `Graph` abstract class:

```
abstract val allVertices: ArrayList<Vertex<T>>
```

Then, implement this property in `AdjacencyMatrix` and `AdjacencyList` respectively:

```
override val allVertices: ArrayList<Vertex<T>>
  get() = vertices
```

```
override val allVertices: ArrayList<Vertex<T>>
  get() = ArrayList(adjacencies.keys)
```

A graph is said to be disconnected if no path exists between two nodes.

```
fun isDisconnected(): Boolean {
  val firstVertex = allVertices.firstOrNull() ?: return false // 1

  val visited = breadthFirstSearch(firstVertex) // 2
  allVertices.forEach {  // 3
    if (!visited.contains(it)) return true
  }

  return false
}
```

Here's how it works:

1. If there are no vertices, treat the graph as connected.
2. Perform a breadth-first search starting from the first vertex. This will return all the visited nodes.
3. Go through every vertex in the graph and check to see if it has been visited before.

The graph is considered disconnected if a vertex is missing in the `visited` list.

Key points

- Breadth-first search (BFS) is an algorithm for traversing or searching a graph.
- BFS explores all of the current vertex's neighbors before traversing the next level of vertices.
- It's generally good to use this algorithm when your graph structure has a lot of neighboring vertices or when you need to find out every possible outcome.
- The queue data structure is used to prioritize traversing a vertex's neighboring edges before diving down a level deeper.