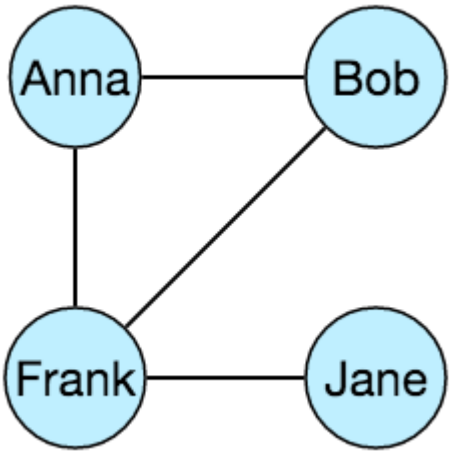
Graphs

This chapter will look at how to work with the graph data structure in JavaScript. We will see at how we can create graphs via Nodes and edges , access elements of a graph via depth wise and breadth wise search.

Introduction

A Graph is a data structure that is helpful in mapping several real world scenarios. e.g. You and your friends on Facebook form a graph called a social graph. Similarly, the roads between towns and cities form a graph commonly know as 'Map'.

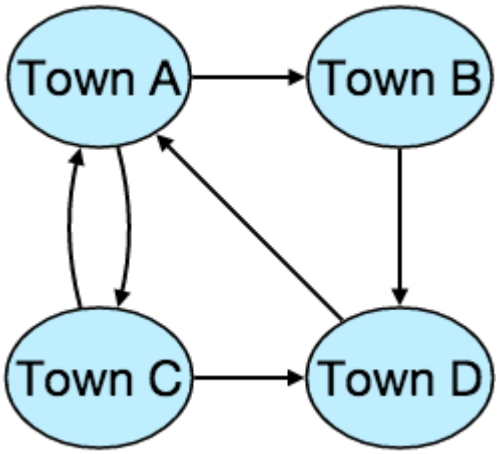
In computer science, a graph consists of a set of vertices and a set of edges. The vertices are the entities and edges define relationships between those entities. e.g. Bob, Frank and Anna are friends. Frank has a friend named Jane who isn't friend with Bob and Anna. In this example, all four people are vertices and edges represent a direct friendship. Let's look at the following graph



Friendship Graph

The above graph shows edges that don't have any direction. Such graphs are called ***Undirected Graphs***. Undirected graphs show that the connection goes both ways. Here we are using an undirected graph as friendship is a mutual relationship (or at least it is supposed to be).

The edges with arrows represent direction of the relationship and such graphs are called ***Directed Graphs***. e.g. let's draw a map of places where a directed edge represents that a road connects vertex A to B. If the edge goes from A to B, it means that the there is a road (one-way road) from A to B. Let's see an example



Graph of Roads

The above graph shows that if you are in Town A, you cannot directly go to Town D. You would either have to go through Town B or Town D. Similarly, from Town D, you cannot go to anywhere except Town A etc.

We've discussed some theory about graphs. Let's see how can we represent vertices and edges in a graph.

Representing Graph in Javascript

We've already established that a graph is a set of vertices and a set of edges. Representing vertices is easy. It's just a list of vertex names. Edges is a little tricky. For every edge, we need to store a source and destination vertex. There are two common ways to store edges. They are called

1. Adjacency Matrix
2. Adjacency List

Let's look at both of these techniques.

Adjacency Matrix

Adjacency Matrix is a square matrix (table) where first row and first column consists of vertices in the graph. The rest of the cells contain either 0 or 1 (it could be another number for weighted graphs but let's ignore that for a while). So each Row x Column intersection points to a cell and the value of that cell dictates whether the vertex represented by Row and vertex represented by Column are connected or not. If the value is 1 for cell V1 x V2, then V1 is connected to V2. An example would help here. Let's create the adjacency matrix for the town graph we saw above.

	Town A	Town B	Town C	Town D
Town A	0	1	1	0
Town B	0	0	0	1
Town C	1	0	0	1
Town D	1	0	0	0

Adjacency Matrix for our City

Now if you have to tell whether there is a direct road between TownA and Town D, you just look at TownA x TownD cell which is 0 so there is no road between Down D.

When we have an undirected graph, we call it a ***symmetric adjacency matrix*** because if a graph is undirected then, if V1 x V2 is an edge, then V2 x V1 will also be an edge. To clarify, let's look at the adjacency matrix for the friend graph that we saw above.

	Anna	Bob	Frank	Jane
Anna	0	1	1	0
Bob	1	0	1	0
Frank	1	1	0	1
Jane	0	0	1	0

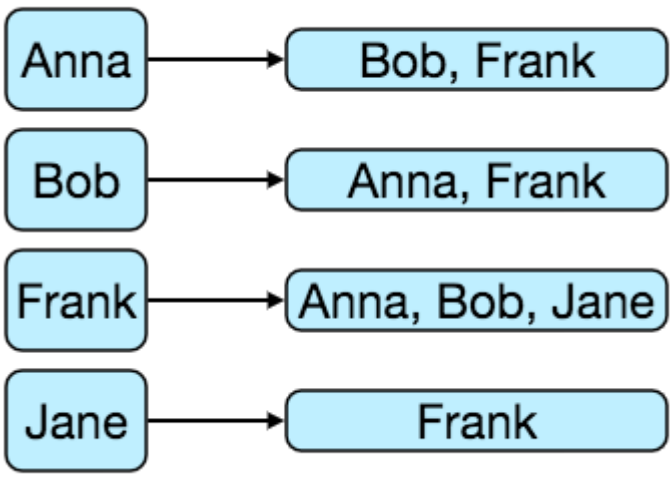
Adjacency Matrix of friendship Graph

Adjacency List

If you notice, in above adjacency matrices, there are a lot of zeroes. As graphs grow, graphs might have far less number of connections than the vertices. Hence, storing non-existence of an edge becomes wasteful e.g. Facebook has 1.5 Billion users but each user has 200-300 people as friends on average. If Facebook stored this friendship graph as an adjacency matrix, it will be a matrix of quintillion rows and columns. It's almost impossible to store such a big matrix. However, the key thing to notice here is that it will be a very sparse matrix and most of the cells will be zero. It makes sense to only store the connections that exist and if the connection doesn't exist, we assume that it's non-existent. This is called an adjacency list.

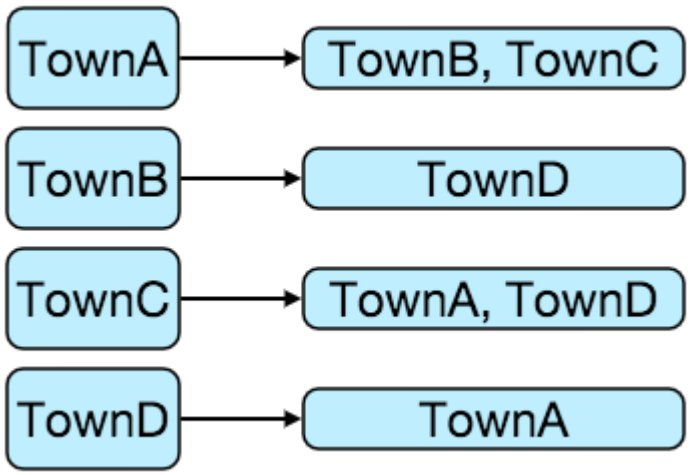
More precisely, Adjacency list is an array of lists where each element is the array is a vertex and each vertex points to a list of vertices it connects to.

Let's look our friendship graph stored as an adjacency list.



Adjacency list for Friendship Graph

Similarly, here's the adjacency list for our city graph.



Adjacency list for Friendship Graph

In our implementation, we are going to opt for Adjacency Lists over Adjacency Matrix as adjacency lists are more space efficient. However, if you end up in a scenario where your adjacency matrix is going to be nearly full, you should opt for adjacency matrix.

Let's have a quick quiz on graphs

Graphs

1 When the edges of a graph have a direction , that type of graph is known as

2 A vertex in the Graph is also referred to as

Check Answers

Implementing a Graph in Javascript

For simplicity, we assume that the vertices are either strings or integers and a string or its equivalent integer points to the same node (If you have followed the Dictionary tutorial, you probably know where are we going with this. We are going to use a Javascript object to store list of vertices just like we did in the dictionary). Alternatively, we could have used our Dictionary data structure here. Here's our Graph data structure.

```
function Graph() {
  this._vertices = {};
  this._vertexCount = 0;
}
```

Adding a Vertex to Graph

For adding a vertex, we add the vertex to our vertices object. The key is the name of the vertex and value is another object which is actually going to be the list of edges.

```
Graph.prototype.addVertex = function(v) {
  var vertexType = typeof(v);

  if (vertexType != 'number' &&
      vertexType != 'string') {
    throw 'Vertex can only be a string or number';
  }

  if (this._vertices.hasOwnProperty(v)) {
    throw 'Duplicate Vertex is not allowed';
  }

  this._vertices[v] = {};
  this._vertexCount++;
}
```

Adding Edges to a vertex

Adding edges to a vertex is easy. We'll just add the edges to the vertex's adjacency list. Only thing to notice here is that we ensure that the edges exist as vertices in our graph.

```
Graph.prototype.addEdges = function(v, edges) {
  if (!this._vertices.hasOwnProperty(v)) {
    throw 'Vertex not found';
  }

  var edgesObj = this._vertices[v];

  for (var i = 0; i < edges.length; i++) {
    var edge = edges[i];

    if (this._vertices.hasOwnProperty(v)) {
      throw 'Invalid vertex cannot be added as edge';
        }

    edgesObj[edge] = true;
  }
}
```

Getters for Vertices and Edges

Implementing Getters is quite straightforward. We will iterate over the lists (both vertices and edges) and return them. Let's look at how are we implementing getter for vertices.

```
Graph.prototype.getVertices = function() {
  var vertices = [];
  for (var v in this._vertices) {
    vertices.push(v);
  }

  return vertices;
}
```

... and similarly, getter for edges given a vertex is implemented below.

```
Graph.prototype.getEdges = function(v) {
  if (!this._vertices.hasOwnProperty(v)) {
    throw 'Vertex not found';
  }

  var edgesObj = this._vertices[v];
  var edges = [];
```

```
  for (var e in edgesObj) {
    edges.push(e);
  }

  return edges;
}
```

Let's see our Graph code in action

It's time to see our graph code in action. We'll create the friendship graph that we've been using as example.

> Run the following code to see Graph in action.

JavaScript

HTML

CSS (SCSS)

```
var graph = new Graph();
graph.addVertex('Anna');
graph.addVertex('Bob');
graph.addVertex('Frank');
graph.addVertex('Jane');

graph.addEdges('Anna', ['Bob', 'Frank']);
graph.addEdges('Bob', ['Anna', 'Frank']);
graph.addEdges('Frank', ['Anna', 'Bob', 'Jane']);
graph.addEdges('Jane', ['Frank']);

var vertices = graph.getVertices();
for(var i=0; i<vertices.length; i++){
  var edges = graph.getEdges(vertices[i]);
  console.log(vertices[i] + ' is friends with '+ edges);
}
```

Console

Clear

```
Anna is friends with Bob,Frank
Bob is friends with Anna,Frank
Frank is friends with Anna,Bob,Jane
Jane is friends with Frank
```

Graph Search (or Traversal) Algorithms

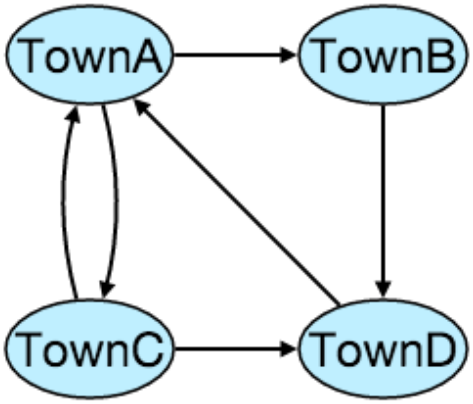
There are two most common ways to traverse a graph given a start node. There are many other ways to traverse a graph depending on the nature of the graph (e.g. Tree is a special graph and we've seen that we use Pre-Order, In-Order and Post-Order traversals for the tree). Here are the most common Graph traversal algorithms.

Depth First Search

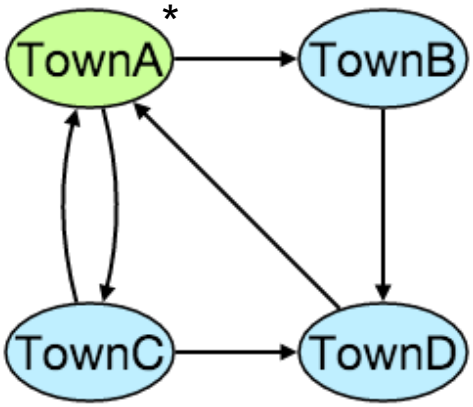
Breadth First Search

Depth First search in the graph

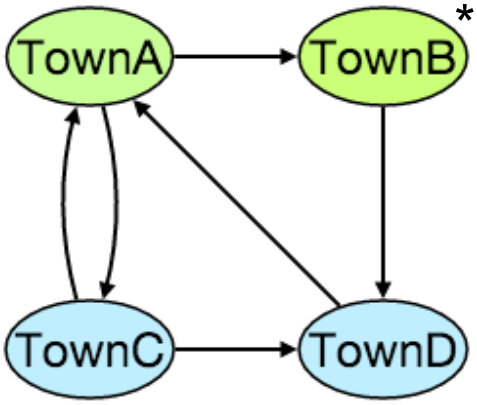
Depth First Search (DFS) is a traversal strategy in which we start from a node and then traverse the graph as far as possible before backtracking and starting traversal from the next node. At each step, we mark the nodes that we've already visited so that we don't visit them again. Here's a visualization of DFS starting at "Town A" in our city graph.



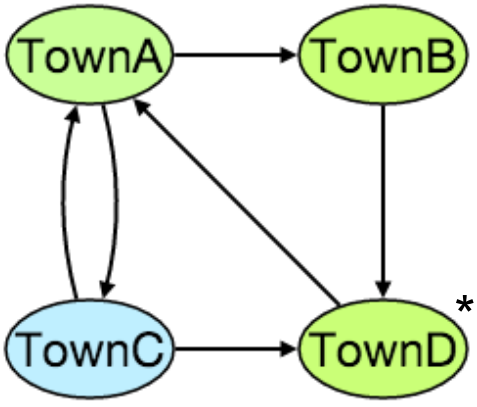
DFS - start at Town A



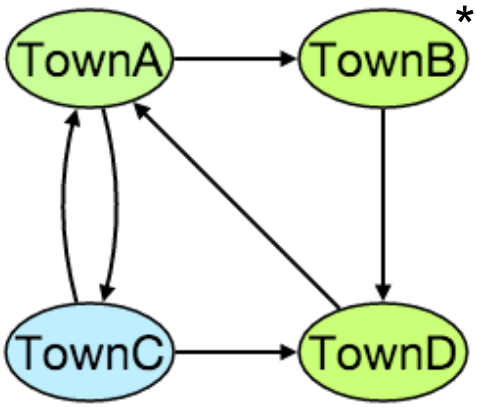
TownA is marked as visited.
Visit TownB



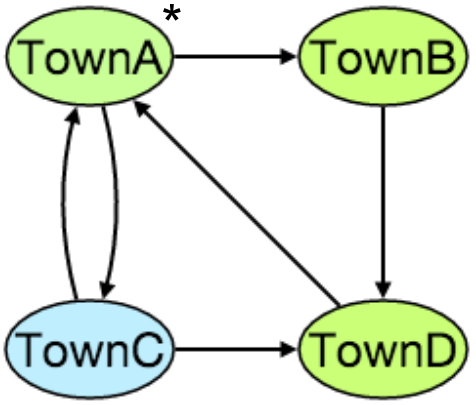
TownB is marked as visited.
Visit TownD.



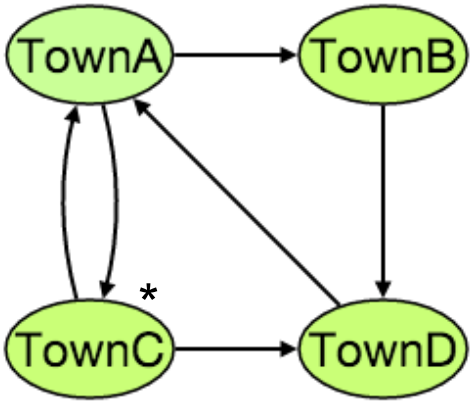
TownD is marked as visited.
Visit TownA but A is already visited.



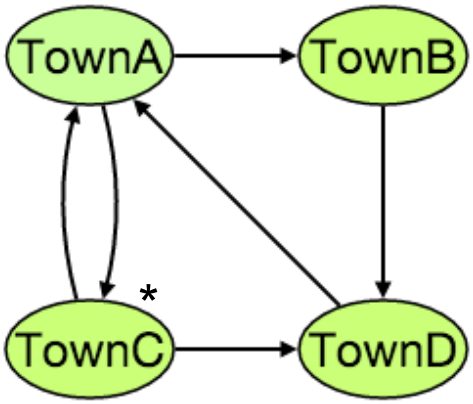
We backtrack to B. B has no more edges so we backtrack to A.



We are at A and we figure that it has another edge so Visit TownC

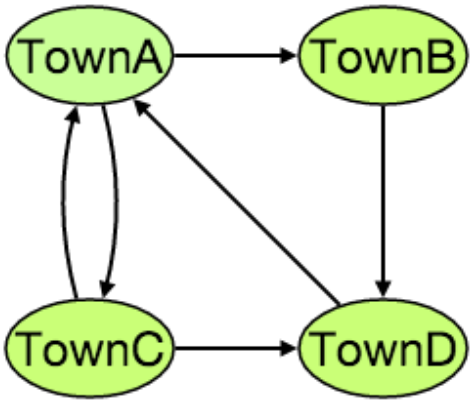


TownC is marked visited.



We now try going to A from C
but A is already visited.

8 of 9



Similarly, all other nodes are
already visited.So we are done.

9 of 9

Here's the code for Depth First Search

```
var DFS = function(graph) {
  var vertices = graph.getVertices();
  if (vertices.length === 0) {
    return;
  }

  // Mark all vertices as NOT VISITED at start
  var visited = {};
  for (var i = 0; i < vertices.length; i++) {
    visited[vertices[i]] = false;
  }
```

```
  // Define our DFS impl method
  function DFSImpl(v) {
    visited[v] = true;
    console.log('Visiting Vertex: ' + v);

    var edges = graph.getEdges(v);
    for (var j = 0; j < edges.length; j++) {
      var edge = edges[j];
      if (!visited[edge]) {
       DFSImpl(edge);
      }
    }
  }

  // Start DFS
  for (var i = 0; i < vertices.length; i++) {
    var vertex = vertices[i];
    if (!visited[vertex]) {
      DFSImpl(vertex);
    }
  }
};
```

> *Run the following code to see DFS in action*

JavaScript

HTML

CSS (SCSS)

```
var graph = new Graph();
graph.addVertex('Anna');
graph.addVertex('Bob');
graph.addVertex('Frank');
graph.addVertex('Jane');

graph.addEdges('Anna', ['Bob', 'Frank']);
graph.addEdges('Bob', ['Anna', 'Frank']);
graph.addEdges('Frank', ['Anna', 'Bob', 'Jane']);
graph.addEdges('Jane', ['Frank']);

DFS(graph);
```

Console

Clear

```
Visiting Vertex: Anna
```

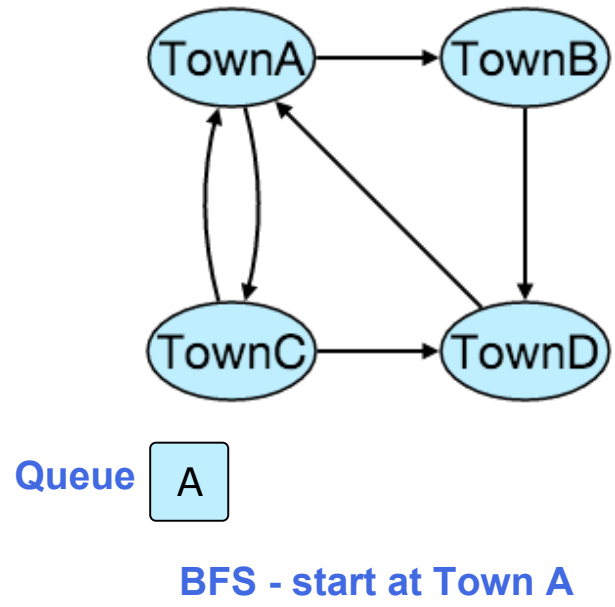
```
Visiting Vertex: Bob
```

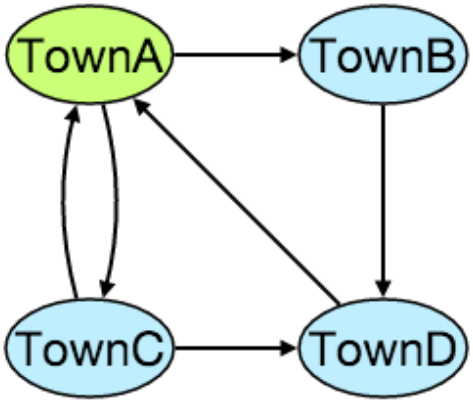
```
Visiting Vertex: Frank
```

```
```

Breadth First search

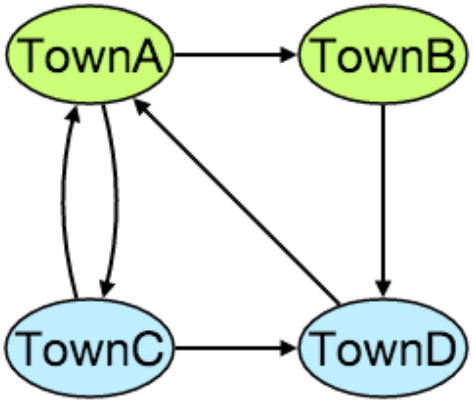
A breadth-first search (BFS) starts at a first vertex and tries to visit vertices as close to the first vertex as possible. In essence, this search moves through a graph layer by layer, first examining layers closer to the first vertex and then moving down to the layers farthest away from the starting vertex. We achieve this by maintaining a queue of the edges connected to the node that we are visiting. We then pickup the next vertex to visit from the queue. Here's a quick visualization





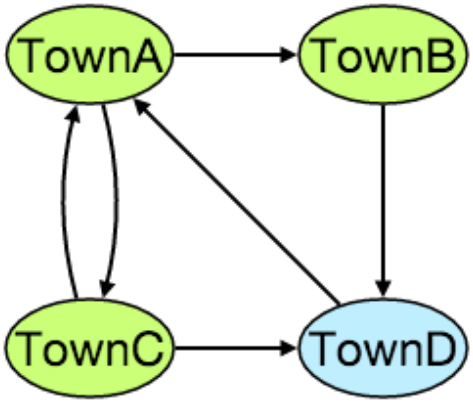
Queue B C

Visit A. Put B, C in Queue.



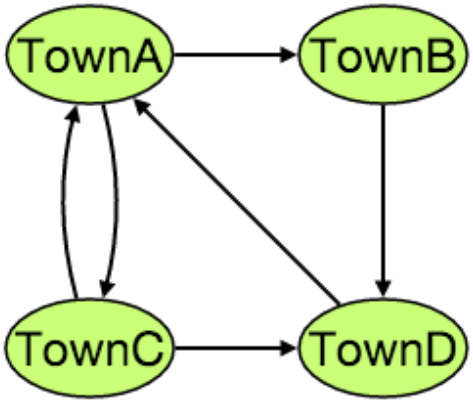
Queue C D

Visit B. Put D in Queue.



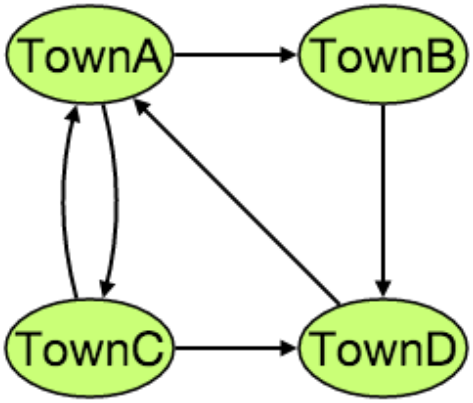
Queue C D

Visit C. Don't put A or D in queue as they are already visisted.



Queue D

Visit D from the queue.



Queue *<empty>*

Queue is empty so we are done.

6 of 6

Here's the code for BFS.

```
var BFS = function(graph) {
  var vertices = graph.getVertices();
  if (vertices.length === 0) {
    return;
  }

  // Mark all vertices as NOT VISITED at start
  var visited = {};
  for (var i = 0; i < vertices.length; i++) {
    visited[vertices[i]] = false;
  }

  var queue = [];
  var startV = vertices[0];
  queue.push(startV);
  visited[startV] = true;

  while (queue.length > 0) {
    // Inefficient. use the Wueue that
    // we created in Queue tutorial
    var v = queue.shift();
    console.log('Visited: ' + v);
    var edges = graph.getEdges(v);

    for (var i = 0; i < edges.length; i++) {
      var e = edges[i];
      if (!visited[e]) {
        queue.push(e);
```

```
        visited[e] = true;
      }
    }
  }
};
```

> Run following code to see BFS in action

JavaScript

HTML

CSS (SCSS)

```
var graph = new Graph();
graph.addVertex('Anna');
graph.addVertex('Bob');
graph.addVertex('Frank');
graph.addVertex('Jane');

graph.addEdges('Anna', ['Bob', 'Frank']);
graph.addEdges('Bob', ['Anna', 'Frank']);
graph.addEdges('Frank', ['Anna', 'Bob', 'Jane']);
graph.addEdges('Jane', ['Frank']);

BFS(graph);
```

Console

Clear

```
Visited: Anna
Visited: Bob
Visited: Frank
Visited: Jane
```

Exercise 1

In our first exercise, we will implement a function to count the total number of edges in a graph.

> Some Tests are failing as the countTotalEdges method is not implemented. Implement it to fix the test cases.

JavaScript

HTML

CSS (SCSS)

```
Graph.prototype.countTotalEdges = function() {
  // Implement this
  return undefined;
}

// This is the method that runs test cases.
// Don't remove this call.
runEvaluation();
```

Console

Clear

```
*** There is some bug lurking there. See failed test cases ***
Here are the tests that ran:
Test case FAILED for Empty graph edges count. Result: undefined. Expected: 0
Test case FAILED for Graph with just one vertex. Result: undefined. Expected: 0
Test case FAILED for Graph with two edges. Result: undefined. Expected: 2
Test case FAILED for Graph with 8 edges. Result: undefined. Expected: 8
```

Summary

A graph is a set of vertices and edges.

Graphs can be represented using either adjacency matrices or adjacency list.

Lookups are fast in adjacency matrices but adjacency lists take up less space in memory.

Two most common algorithms to traverse the graph are called Depth First Search (DFS) and Breadth First Search (BFS).