**Directed Graphs**

6 min

So far we have only built out support for undirected

Preview: Docs Loading link description

graphs

. Next, we will focus on expanding our Graph class to be directed, where there does not necessarily have to be edges going in both directions between the vertices, as we have done with undirected graphs.

The main difference between the undirected graph and directed graph is that our undirected graph uses two edges going in opposite directions to indicate that there is a connection between two vertices.

**Instructions**

1. Checkpoint 1 Passed

**1.**

When a Graph is first created, we need a way to identify if it will be directed or not. In the constructor, add a parameter. This argument will be a boolean of whether or not the graph is directed. Store the isDirected argument in the Graph's isDirected property. By default, isDirected should be set to false.

Hint

To check that the isDirected property is set correctly, create two Graph instances. In the first instance, set the argument to true and log out the Graph's isDirected property. It should be true.

In the second instance, leave the argument blank and log out the Graph's isDirected property. It should be false.

2. Checkpoint 2 Passed

**2.**

We only want to create one edge that points in one direction between two vertices for directed graphs.

Modify the Graph's .addEdge() method to create the edge from vertexTwo to vertexOne only if this isDirected property is false.

3. Checkpoint 3 Passed

**3.**

Just as we only want to create one edge between vertices in a directed graph, we also want to remove only one edge between vertices.

Modify the Graph's .removeEdge() method to remove the edge from vertexTwo to vertexOne if this isDirected property is false.

4. Checkpoint 4 Passed

**4.**

Finally, let's modify our train network to only travel in one direction: from New York to Atlanta. Modify the trainNetwork to be unweighted and directed. Pass false for the first argument and true for the second.

We should see only one edge connection going from Atlanta to New York. New York should have no edges going out.

5. Checkpoint 5 Passed

**5.**

Last, but not least, we should test out our edge removal. Call .removeEdge() on the trainNetwork graph to remove the edge between atlantaStation and newYorkStation.

When we print out the resulting graph, Atlanta and New York should have no connections.

**Graph.js**

```
const Edge = require('./Edge.js');

const Vertex = require('./Vertex.js');


class Graph {
  constructor(isWeighted = false, isDirected = false) {
    this.vertices = [];
    this.isWeighted = isWeighted;
    this.isDirected = isDirected;
  }


  addVertex(data) {
    const newVertex = new Vertex(data);
    this.vertices.push(newVertex);
```

```
  return newVertex;
}

removeVertex(vertex) {
  this.vertices = this.vertices.filter(v => v !== vertex);
}

addEdge(vertexOne, vertexTwo, weight) {
  const edgeWeight = this.isWeighted ? weight : null;

  if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {
     vertexOne.addEdge(vertexTwo, edgeWeight);
     if (!this.isDirected) {
       vertexTwo.addEdge(vertexOne, edgeWeight);
     }
  } else {
    throw new Error('Expected Vertex arguments.');
  }
}

removeEdge(vertexOne, vertexTwo) {
  if (vertexOne instanceof Vertex && vertexTwo instanceof Vertex) {
    vertexOne.removeEdge(vertexTwo);
    if (!this.isDirected) {
      vertexTwo.removeEdge(vertexOne);
    }
  } else {
    throw new Error('Expected Vertex arguments.');
```

```
    }
  }

  print() {
    this.vertices.forEach(vertex => vertex.print());
  }
}

const trainNetwork = new Graph(false, true);
const atlantaStation = trainNetwork.addVertex('Atlanta');
const newYorkStation = trainNetwork.addVertex('New York');
trainNetwork.addEdge(atlantaStation, newYorkStation);
trainNetwork.removeEdge(atlantaStation, newYorkStation);

trainNetwork.print();

module.exports = Graph;
```

**Vertex.js**

```
const Edge = require('./Edge.js');

class Vertex {
  constructor(data) {
    this.data = data;
    this.edges = [];
  }

  addEdge(vertex, weight) {
    if (vertex instanceof Vertex) {
```

```
      this.edges.push(new Edge(this, vertex, weight));
    } else {
      throw new Error('Edge start and end must both be Vertex');
    }
  }

  removeEdge(vertex) {
    this.edges = this.edges.filter(edge => edge.end !== vertex);
  }

  print() {
    const edgeList = this.edges.map(edge =>
        edge.weight !== undefined ? `${edge.end.data} (${edge.weight})` : edge.end.data) || [];

    const output = `${this.data} --> ${edgeList.join(', ')}`;
    console.log(output);
  }
}

module.exports = Vertex;
```

**Edge.js**

```
class Edge {
  constructor(start, end, weight = null) {
    this.start = start;
    this.end = end;
    this.weight = weight;
  }
}
```

```
module.exports = Edge;
```