Network Science Library in Typescript

Soka University of America

Rodrigo Hiroto Morais

January 2022

Contents

Introduction

Technical Aspects

JavaScript (JS) is a multi-paradigm programming language. It is the most-used language in the web. ECMAScript (ES) is the standardized specification of JS. ES is updated almost every year, and brings many different functionalities to the language, some of which are used in the library. The latest version of ES is ES2021, and is already implemented in most modern browsers.

Typescript is a strongly typed programming language that builds on JavaScript. The library is made specifically for dealing with a special kind of mathematical object with very well defined properties. Thus, TS's type functionality serves it very well.

This library is created following the one I had originally coded for the Spring 2020 Network Science class. That original library (Net20) had many flaws and inefficiencies which are addressed with this library in part due to TS.

Basic Graph Theory

Graph theory is a field of mathematics that studies graphs. A graph, or network, essentially consists of two sets:

1. V, a set of vertices (also called nodes), and
2. E, a set of edges (also called links)

Thus, a graph G can be represented as $G = (V, E)$ such that:

$$E \subseteq \{\{x, y\} \mid x, y \in V, x \neq y\} \forall a, b \in E, a \neq b$$

The library only deals with directed or undirected simple graphs with no self-loops. In other words, a graph cannot have more than one edge between any two vertices, and it also cannot connect a vertex to itself. The library considers all networks to be weighted on a technical level. When created, all edges and vertices have their weight set to one. An unweighted network is thus just a network with all weights set to the default of one.

Types and Interfaces

These type definitions are the foundation of the library. They are stored inside 'enums.ts'. The base_id type is used throughout in the library. It signifies that the identification variable for a vertex can be either a string of characters or a number.

```
export type base_id = string | number;
```

An ID of 'vertex_a' is as valid as an ID of42, for instance. The args interfaces are used by function inputs. For example, when creating an edge, the library will be expecting an object with the format of EdgeArgs.

```
export interface VertexArgs {
  id: base_id;
  weight?: number;
}

export interface EdgeArgs {
  from: base_id;
  to: base_id;
  id?: base_id;
  weight?: number;
  do_force?: boolean;
}

export interface NetworkArgs {
  is_directed?: boolean;
  is_multigraph?: boolean;
  edge_limit?: number;
  vertex_limit?: number;
}
```

The question-mark indicates a property is optional. It is possible to create a network without any parameters. Weights are optional parameters, and (as previously mentioned) set to one by default.

The following is an example of code that creates a network 'net', then adds the vertices '1' and "b" and an edge between them:

```
const net = new Network()

net.addVertex({ id: 1 })
net.addVertex({ id: 'b' })

net.addEdge(1, 'b')
```

The networks are by default undirected. A directed network has to be explicitly declared:

```
const is_directed = true
const directed_net = new Network({ is_directed })
```

'ParsedCSV' and 'ERROR' are types used internally by the library to load CSV files and manage error messages, respectively:

```
export type ParsedCSV = string[][];

export const ERROR = {...};
```

Vertex and Edge Classes

The vertex class receives an object with the interface of 'VertexArgs'. The weight is optional and, if not given, set to one.

```
import { base_id, VertexArgs } from "./enums.ts";

export class Vertex {
  readonly id: base_id;
  weight: number;

  /**
   * Vertex constructor
   * @param  {VertexArgs} args
   */
  constructor(args: VertexArgs) {
    this.id = args.id;
    this.weight = args.weight ?? 1;
  }
}
```

The '??' operator is a "Nullish coalescing operator" introduced in ES2021. If 'args.weight' is undefined, the instruction on the right is chosen. This operator is used instead of the ternary 'a ? b : c' operator because if 'args.weight=0', it would still select 'args.weight', whereas the ternary operator would consider '0' a Falsy value.

A Falsy value is something with the same Boolean value as false. '0', for example, although a number, it is still considered false in Typescript. In other languages, such as Ruby, '0' actually has a Truthy value, meaning that if you feed it into a logical operation, it considers it to be true.

The edge class has 'from' and 'to' properties that hold the ID of a vertex in a network, and a weight that behaves in the same way as the vertex class. The IDs of the vertices in an edge are private, meaning they cannot be read or overwritten. When an edge is added to a network, only its weight can change. Changing its vertices would fundamentally change what the edge is and is thus not allowed. The vertices can be accessed and read through the 'vertices' getter, which returns the edge's 'from' and 'to' properties:

```
import { base_id, EdgeArgs } from "./enums.ts";

export class Edge {
  private to: base_id;
  private from: base_id;
  weight: number;

  /**
   * Create an edge between 'from' and 'to'.
   *
   * Weight is set to 1 by default (i.e. unweighted).
   * @param  {EdgeArgs} args
   */
  constructor(args: EdgeArgs) {
    this.from = args.from;
    this.to = args.to;
    this.weight = args.weight ?? 1;
  }

  /**
   * Returns an object with the two vertices in the edge.
   * @returns {{ from:base_id, to:base_id }}
   */
  get vertices(): { from: base_id; to: base_id } {
    return { from: this.from, to: this.to };
  }
}
```

Network Constructor

The network class has 4 'readonly' properties. The edges and vertices are stored in Maps that use 'base_id' as their keys and the values store the actual vertex or edge instance. The two other 'readonly' are Booleans that store fundamental graph properties: the directionality and complexity of the network.

The 'private' properties have to do with hidden functionality and performance limitations. There are limits to the number of edges and vertices a network can have, and they can only be set in the creation of a network.

```
class Network {
  readonly edges: Map<base_id, Edge>;
  readonly vertices: Map<base_id, Vertex>;

  readonly is_directed: boolean;
  readonly is_multigraph: boolean;

  private edge_limit: number;
  private vertex_limit: number;
  private free_eid: number;
  private free_vid: number;

  constructor(args: NetworkArgs = {}) {
    this.edges = new Map();
    this.vertices = new Map();
    this.is_directed = args.is_directed ?? false;
    this.edge_limit = args.edge_limit ?? 500;
    this.vertex_limit = args.vertex_limit ?? 500;
    this.free_eid = 0;
    this.free_vid = 0;
    this.is_multigraph = false;
  }
}
```

The 'free_eid' and 'free_vid' properties will be further explained later. A network with a larger number of maximum edges and vertices could be created as such:

```
const edge_limit = 10_000
const vertex_limit = 10_000

const net = new Network({ edge_limit, vertex_limit })
```

Network Values

There are several network properties that, instead of being stored in a variable, have getters to them. These either have to be calculated on the fly or don't really serve an internal purpose that would justify storing them inside a variable property.

Getters have the basic format:

```
get getter_name(): PropertyType {
  return property;
}
```

And, different from functions, can be accessed without the brackets:

```
net.getter_name
```

Functional Getters

These getters exist mostly to provide functionality to the network class. For example, the args getter returns the properties of the network necessary to make a copy of it:

```
  get args(): NetworkArgs {
    return {
      is_directed: this.is_directed,
      is_multigraph: this.is_multigraph,
      edge_limit: this.edge_limit,
      vertex_limit: this.vertex_limit,
    };
  }
```

The list getters return a list with the values inside the vertices and edges Maps. This is particularly useful for efficiency as it makes it possible to use standard Array functions.

```
get vertex_list(): Vertex[] {
  return [...this.vertices.values()];
}

get edge_list(): Edge[] {
  return [...this.edges.values()];
}
```

The '...' destructuring operator was also introduced in ES2021. It takes the iterable return of the 'values()' function and destructures it into its individual elements. The elements are then put inside an array, which is returned by the getter.

Calculations

These getters involve calculations that make use of the network's vertices and edges. That is why they are not permanently stored inside a property, seeing as they could change any time a new edge or vertex is added to the graph.

The calculations are also not turned into their own functions because they do not require more complex algorithms.

Weight

```
get weight(): number {
  return this.vertex_list
    .map((vertex) => vertex.weight)
    .reduce((prev, curr) => prev + curr);
}
```

Say $w(x)$ is the weight of the vertex x in the network $G = (V, E)$. A network's weight is given by:

$$\sum w(x), \forall x \in G$$

The getter uses the 'Array.prototype.map' and 'Array.prototype.reduce' functions. It first maps the 'vertex_list' into a list with just the weights of the vertex, and then reduces it by summing all of the new list's values.

There are three other functions related to vertex weight.

```
/**
 * List of vertices with negative weight.
 * @returns Vertex[]
 */
 get negative_vertices(): Vertex[] {
 const { vertex_list } = this;
 return vertex_list.filter((vertex) => vertex.weight < 0);
```

```
}

/**
 * List of vertices with positive weight.
 * @returns Vertex[]
 */
get positive_vertices(): Vertex[] {
  const { vertex_list } = this;
  return vertex_list.filter((vertex) => vertex.weight > 0);
}

/**
 * List of vertices with zero weight.
 * @returns Vertex[]
 */
get zero_vertices(): Vertex[] {
  const { vertex_list } = this;
  return vertex_list.filter((vertex) => vertex.weight == 0);
}
```

Maximum Number of Edges

The highest number of edges M_E a graph with $|V|$ vertices can have is:

$$M_E = \frac{|V| * (|V| - 1)}{2}$$

```
  get max_edges(): number {
    return (this.vertices.size * (this.vertices.size - 1)) / 2;
  }
```

Density

A graph's density D is defined by:

$$D = \frac{|E|}{M_E}$$

The number of edges a graph has, divided by the maximum number of edges it could have with the number of nodes it currently has.

```
  get density(): number {
    return this.edges.size / this.max_edges;
  }
```

Functions

The functions for removing edges and vertices are self-explanatory in their name.

```
removeEdge(args:
          { from: base_id; to: base_id; id?: base_id }) {
  if (args.id !== undefined) {
    this.removeMultigraphEdge(args.id);
    return;
  } else if (this.is_multigraph) {
    throw { message: ERROR.UNDEFINED_ID, id: args.id };
  }

  this.edges.forEach(({ vertices }, id) => {
    if (this.checkEdgeIsSame(vertices, args)) {
      this.edges.delete(id);
      return;
    }
  });
}
```

The 'removeVertice' function differs itself from 'removeEdge'. When a vertex is removed, all of the edges associated with it also have to be removed.

```
removeVertex(id: base_id) {
  if (!this.vertices.has(id))
    throw { message: ERROR.INEXISTENT_VERTICE, vertex: id };

  this.vertices.delete(id);

  this.edges.forEach(({ vertices }, key) => {
    const { from, to } = vertices;
    if (from === id || to === id)
      this.edges.delete(key)
  });
}
```

An advantage of using Maps to store the network's vertices and edges is that it is easier to get them by ID:

```
hasVertex(id: base_id): boolean {
  return this.vertices.has(id);
}
```

To get an edge between two vertices, 'Array.prototype.find' is used in the 'edge_list' array. The 'find()' function returns the first element in the list that fulfills the given property.

```
/**
 * Returns the edge between two nodes.
 * @param  {base_id} from
 * @param  {base_id} to
 * @returns base_id[]
 */
 edgeBetween(
  from: base_id,
  to: base_id,
  is_directed = this.is_directed
): Edge | undefined {
  return this.edge_list.find(({ vertices }) =>
    this.checkEdgeIsSame(vertices, { from, to }, is_directed)
  );
}
```

The property fed into the function checks if the vertices given to 'edgeBetween()' form an edge that is the same as an edge that actually exists in the network. To check if two edges are the same (if they have the same 'from' and 'to') the private function 'checkEdgeIsSame()' is used.

A private function can only be accessed inside the class declaration.

```
checkEdgeIsSame(
    edge_a: EdgeArgs,
    edge_b: EdgeArgs,
    is_directed = this.is_directed
  ): boolean {
    if (edge_a.from === edge_b.from && edge_a.to === edge_b.to)
      return true;
    else if (
      edge_a.to === edge_b.from &&
      edge_a.from === edge_b.to &&
      !is_directed
    )
      return true;
```

```
    return false;
  }
```

The way an edge check works depends on whether a network is directed or not.

Nevertheless, it is also possible to force an undirected check. This is useful when it is only necessary to know if an edge between A and B exists at all, instead of whether a *directed* edge from A to B exists.

There are also two functions that serve mostly as a convenience.

```
  newVID(): base_id {
   let id = this.free_vid++;
   while (this.vertices.has(id)) {
     id = Math.floor(Math.random() * this.vertex_limit);
   }
   return id;
 }

 newEID() {
   let id = this.free_eid++;
   while (this.edges.has(id)) {
     id = Math.floor(Math.random() * this.edge_limit);
   }
   return id;
 }
```

These ID functions help create new IDs to be assigned to new edges or vertices. On the user level, there rarelly is any reason to assign IDs to edges. However, since the network uses Maps, when an edge is created, it needs to have an ID assigned to it. The 'newEID()' function generates a valid ID to be used internally.

There are four special ways to add edges and vertices to a network.

```
 addEdgeMap(edge_map: Map<base_id, Edge>) {
  edge_map.forEach((edge, id) => this.edges.set(id, edge));
}

addEdgeList(edge_list: EdgeArgs[]) {
  edge_list.forEach((edge_args, id) =>
    this.edges.set(id, new Edge(edge_args))
  );
}

 addVertexMap(vertex_map: Map<base_id, Vertex>) {
  vertex_map.forEach((vertex, id) =>
    this.vertices.set(id, vertex));
```

```
}

addVertexList(vertex_list: VertexArgs[]) {
  vertex_list.forEach((vertex_args, id) =>
    this.vertices.set(id, new Vertex(vertex_args))
  );
}
```

An array of vertex information could be added as such:

```
const vertex_list = [
  {id: 'node_1', weight:3},
  {id: 'node_2', weight:1.2}
]

net.addVertexList(vertex_list)
```

Other utility functions are:

```
hasEdge(from: base_id,
        to: base_id,
        is_directed = false): boolean {
  return this.edge_list.some(({ vertices }) =>
    this.checkEdgeIsSame(vertices, { from, to }, is_directed)
  );
}

getEdgesBetween(
  from: base_id,
  to: base_id,
  is_directed = this.is_directed
): base_id[] | base_id {
  let edge_list: base_id[] = [];

  this.edges.forEach(({ vertices }, id) => {
    if (this.checkEdgeIsSame(
                              vertices,
                              { from, to },
                              is_directed
                            )) {
      edge_list.push(id);
    }
  });

  return this.is_multigraph ? edge_list : edge_list[0];
}
```

```
addVertex(args: VertexArgs) {
  if (this.vertices.size >= this.vertex_limit)
    throw { message: ERROR.VERTICE_LIMIT };
  if (args.id !== undefined && this.vertices.has(args.id))
    throw { message: ERROR.EXISTING_VERTICE };

  this.vertices.set(args.id, new Vertex(args));
}
```

The 'addEdge' function looks different from the 'addVertex' function because it has to deal with many exceptions.

```
addEdge(args: EdgeArgs) {
  args.do_force ??= true;
  args.weight ??= 1;

  args.id ??= this.newEID();

  if (this.edges.has(args.id))
    throw { message: ERROR.EXISTING_EDGE };

  if (this.edges.size >= this.edge_limit)
    throw { message: ERROR.EDGE_LIMIT };

  if (!args.do_force) {
    if (!this.vertices.has(args.from))
      throw {
        message: ERROR.INEXISTENT_VERTICE,
        vertex: args.from
      };
    if (!this.vertices.has(args.to))
      throw {
        message: ERROR.INEXISTENT_VERTICE,
        vertex: args.to
      };
  } else {
    if (!this.vertices.has(args.from))
      this.addVertex({ id: args.from });
    if (!this.vertices.has(args.to))
      this.addVertex({ id: args.to });
  }

  if (!this.is_multigraph && this.hasEdge(args.from, args.to))
    return;
```

```
  this.edges.set(args.id, new Edge(args));
}
```

The main addition to 'addEdge' is the 'do_force' argument. It is set to true by default. When 'addEdge' is called, it first checks if the vertices you are trying to connect exists. If they don't and 'do_force=true', the function will add the vertices to the network automatically.

In the following code, vertices '1' and '2' are created by 'addEdge' before it adds an edge to net.

```
const net = new Network()
net.addEdge({ from: 1, to: 2 })
```

If the edge should only be added if the network already has the given vertices, do_force can be set to false:

```
const net = new Network()
net.addEdge({ from: 1, to: 2, do_force: false })
```

The previous code will throw an error, since 'addEdge' will not try to force the creation of the edge by adding vertices '1' and '2' to net.

There are also some private utility functions used in algorithms that will be explained in the next chapter.

```
  listHasTriplet(triplet_arr: Triplet[],
                 triplet: Triplet): boolean {
    return !!triplet_arr.find((trip) =>
      this.isSameTriplet(triplet, trip));
  }

  isSameTriplet(arr1: Triplet, arr2: Triplet): boolean {
    if (arr1.length !== arr2.length) return false;
    return arr1.every((element, index) =>
      element === arr2[index]);
  }
```

Algorithms

Neighborhood

A vertex's neighborhood is defined as the vertices that are connected to that vertex by an edge. The 'Network.neighbors(id)' function returns a list with the IDs of the given vertex's neighbors.

```
neighbors(id: base_id): base_id[] {
  const neighborhood: base_id[] = [];

  this.edges.forEach(({ vertices }) => {
    const { from, to } = vertices;
    if (from === id) neighborhood.push(to);
    else if (to === id) neighborhood.push(from);
  });

  return neighborhood;
}
```

It goes through the Map of edges, and finds any edge that has one of its vertices match the ID given.

When a network is directed, an edge can have two distinct types of neighbors. In-neighbors are vertices that connect to a vertex a with an edge that ends on a. In other words, for a vertex a in an edge from b to a, b is an in-neighbor of a. Out-neighbors are the opposite.

```
inNeighbors(id: base_id): base_id[] {
  const in_neighbors: base_id[] = [];
  if (!this.is_directed) return in_neighbors;

  this.edges.forEach(({ vertices }) => {
    const { from, to } = vertices;
    if (to === id) in_neighbors.push(from);
  });

  return in_neighbors;
```

```
}

outNeighbors(id: base_id): base_id[] {
  const out_neighbors: base_id[] = [];
  if (!this.is_directed) return out_neighbors;

  this.edges.forEach(({ vertices }) => {
    const { from, to } = vertices;
    if (from === id) out_neighbors.push(to);
  });

  return out_neighbors;
}
```

Degree

A vertex's degree can be defined as the number of edges that contain said vertex.

$$E_x = \{k \mid k \in E, x \in k\}$$
$$D_x = |E_x|$$

In the library:

```
/**
 * Return the degree of a vertex with the given ID.
 * @param  {base_id} id
 * @returns number
 */
degree(id: base_id): number {
  let vertex_degree = 0;

  this.edges.forEach(({ vertices }) => {
    const { from, to } = vertices;
    if (from === id || to === id) vertex_degree++;
  });

  return vertex_degree;
}
```

The two other degree functions are related to in and out-neighbors, and are only defined when a network is directed. 'inDegree' returns the number of edges that end on the given edge. 'outDegree' does the opposite, returning the number of edges that start on the given vertex.

```
inDegree(id: base_id): number {
  let in_degree = 0;
  if (!this.is_directed) return in_degree;

  this.edges.forEach(({ vertices }) => {
    const { to } = vertices;
    if (to === id) in_degree++;
  });

  return in_degree;
}

outDegree(id: base_id): number {
  let out_degree = 0;
  if (!this.is_directed) return out_degree;

  this.edges.forEach(({ vertices }) => {
    const { from } = vertices;
    if (from === id) out_degree++;
  });

  return out_degree;
}
```

Assortativity

Assortativity relates to a vertex's propensity to connect to other vertices that have similar properties. In this algorithm, the property used as reference is a vertex's degree.

Assortativiry of a vertex i is usually written as:

$$k_{nn}(i) = \frac{\sum a_{ij} k_j}{j_i}$$

Where a_{ij} is positive if there is and edge between vertices i and j, and k_j is the degree of vertex j. And this is what this would look like when literally transcribed into code:

```
assortativity(id: base_id): number {
  let vertex_assortativity = 0;

  this.edges.forEach(({ vertices }) => {
    const { from, to } = vertices;
    if (from === id) vertex_assortativity += this.degree(to);
    else if (to === id) vertex_assortativity += this.degree(from);
```

```
  });

  return vertex_assortativity / this.degree(id);
}
```

However, as previously mentioned, we already have a function for getting the neighbors of a specific vertex. Thus, we could write our assortativity A_x of vertex x as:

$$A_x = \frac{\sum D_n(x)}{D_x}$$

In code:

```
assortativity(id: base_id): number {
  let vertex_assortativity = 0;

  this.neighbors(id).forEach((neighbor_id) => {
    vertex_assortativity += this.degree(neighbor_id)
  });

  return vertex_assortativity / this.degree(id);
}
```

This helps us avoid repetition and makes things more clear. If someone who doesn't know what assortativity is were to look at this they would have an easier time figuring out what it means.

Saved 10 seconds from triplets by only using directed edges

Testing Process

For testing, the file 'tester.ts' is used.

```
deno run --allow-write --allow-read tester.ts
```

Test Output

changing from forEach to some cut the time in half essentially