Network Science Library in Typescript

Soka University of America

Rodrigo Hiroto Morais

January 2022

Contents

Introduction

This capstone consists of two parts. The first part is a network science library created using TypeScript with Deno. A library, here, refers to code written modularly in such a way that it can be imported and used elsewhere. Libraries help programmers avoid doing work that was already done by someone else. This library can be used for research that relates to network science, as well as imported into other programs or projects to serve as a base for any kind of computation that involves graphs.

The scond part is this document. It is organized as follows:

Here, in the introduction, the technologies used in the creation of the library are described. A brief introduction to graph theory is also given. More complex concepts that relate to network science will also be provided in later chapters.

In the types and interfaces chapter of this document, we give an overview of all the fundamental data structures and definitions of the library.

After that, the chapters for values and functions describe simpler algorithms of the library, and also provide background on what they mean for graph theory.

We will then describe explain its more complex algorithms as well as some of the decisions that went into writing them the way they currently are.

The testing chapter shows how testing was done to make sure all the algorithms and values outputted by the library were correct.

Finally, we provide a real-world example of what the library could be used for.

All images shown here were created using an older version of this library unless stated otherwise.

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 this 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 its older version, written in JS. That version was originally coded for the Spring 2020 Network Science class. That original library (Net20) had many flaws and inefficiencies which are addressed with this library.

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)

Formally, we can write that as:

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

Thus, a graph G can be represented as $G = (V, E)$.

The library only deals with directed or undirected graphs with no self-loops and no multi-edges. In other words, a graph cannot have more than one edge between any two vertices, and it also cannot connect a vertex to itself.

A common use for network science is social networks. Each user is usually represented as a node in the network, while their connections are the edges. One example of an undirected network is Facebook. Users can either be friends or not.

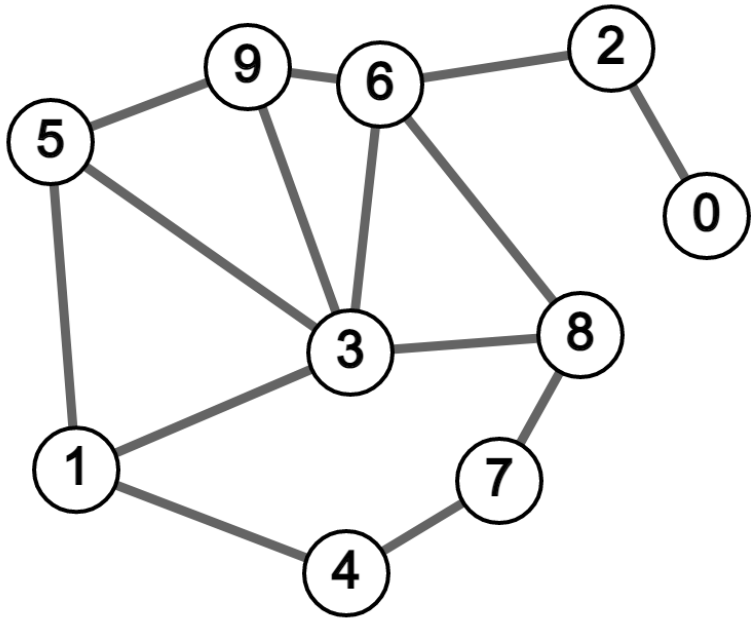

Figure 1: Example of an undirected network.

In contrast to that binary relationship of Facebook friendships we have Twitter's system of followers. Twitter's network can be represented as a directed graph. The connections between users are directional: User A can follow user B without the latter following the former.

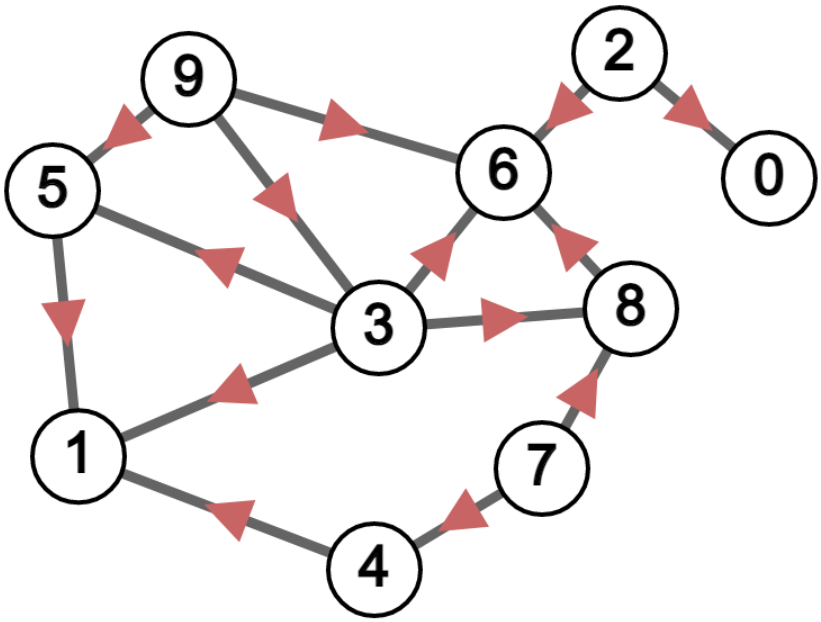

Figure 2: Network in previous example with randomized directed edges.

The library considers all networks to be weighted on a technical level. This means that, when created, any edge or vertex has their weight set to one. An unweighted network is thus just a network with all weights set to the default of one.

Visually, the weight of an edge is usually represented as thickness. A weighted network could be used to represent interactions between people in a day. The more interactions users have, the greater the edge's weight.

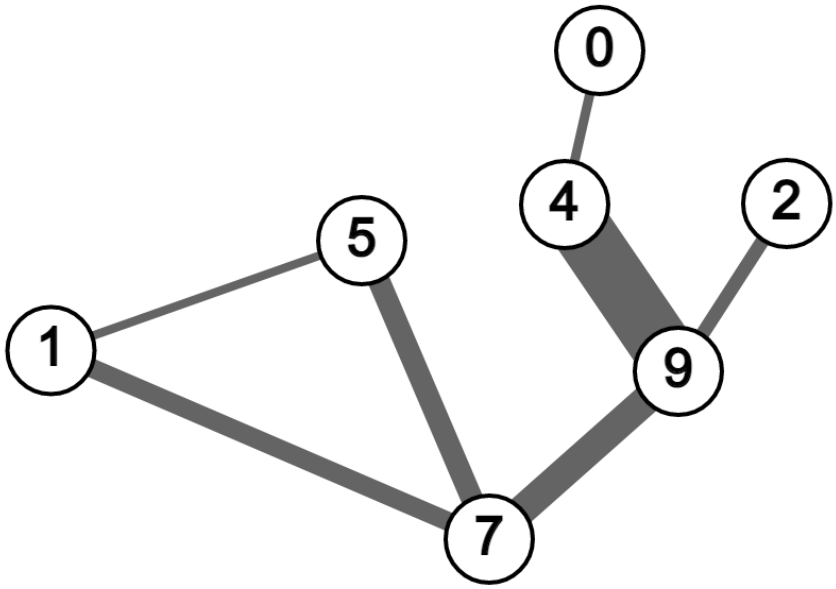

Figure 3: A network with weighted edges.

In Figure 3, individuals 9 and 4 interacted many times during the day, whereas 4 and 0 interacted very little. 5 and 4 didn't interact at all.

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 of 42, 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`.

The question-mark indicates a property is optional. Weights are optional parameters, and (as previously mentioned) set to one by default.

```
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;
  edge_limit?: number;
  vertex_limit?: number;
}
```

It is possible to create a network without any parameters. The following is an example of code that creates a network `net`, then adds the vertices '1' and 'b' and an edge between them. It is represented visually in Figure 4.

```
const net = new Network()

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

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

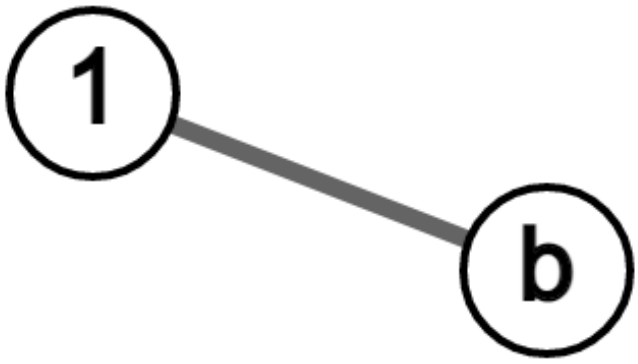

Figure 4: ‘net’ with an edge between ‘1’ and ‘b’.

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

```
const is_directed = true
const directed_net = new Network({ is_directed })
net.addEdge(1, 'b')
```

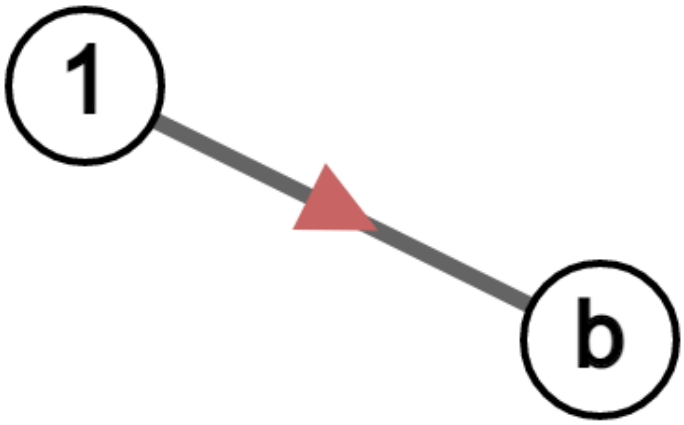

Figure 5: Network now directed from ‘1’ to ‘b’

`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 provided as a parameter, set to one.

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

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

  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. For example, '0', although a number, is still considered Falsy 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 evaluates to 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 be altered from outside the class.

Changing the verticesof an edge would fundamentally change what that 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;

  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. Most of the time when it is necessary to work with a graph data structure, the scale of such graphs is already known. Therefore, to assist with error handling limits to the number of edges and vertices are set. This helps because it prevents users from getting into infinite loops that try to add too many edges or vertices.

```
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 some of 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 finally returned by the getter.

Calculations

These next 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 algorithms that are too elaborate.

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

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

Where $|V|$ represents the size of the vertex set V.

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

Density

A graph's density D is defined by:

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

Where $|E|$ is the size of the edge set $|E|$ 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 remove said elements of the network.

```
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 or not a network is directed. Nevertheless, it is also possible to force an undirected check.

A forced undirected check 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 two functions that serve mostly as a convenience. They are ID functions which help create new IDs that can be assigned to new edges or vertices. On the user level, there is seldom any reason to assign IDs to edges. However, because the network uses `Maps`, every edge needs to have an ID assigned to it. The `newEID()` function generates a valid ID to be used internally.

```
  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;
 }
```

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 edges could be added as such:

```
const edge_list = [
  [2,3],
  ['b',3]
]

net.addEdgeList(edge_list)
```

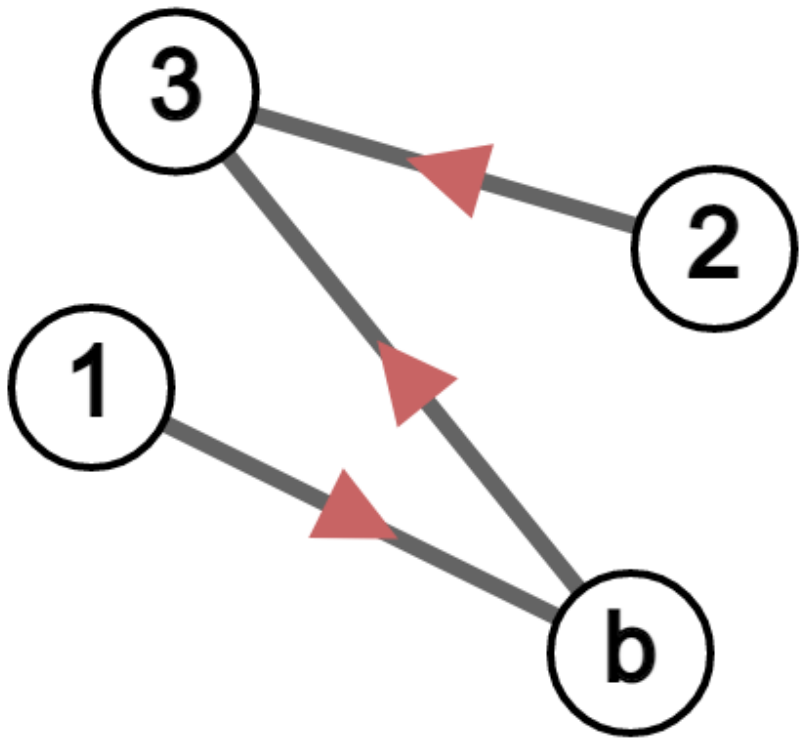

Figure 6: Our previous 'net', now with the list of edges added.

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 x's neighborhood $H(x)$ is defined as the vertices that are connected to that vertex by an edge. Note also that $H_x \subset V$. $H[x]$ is used to define the set with neighbors of x as well as x itself: $H[x] = H(x) + x$. The `Network.neighbors(id)` function returns a list with the IDs of the neighbors of the vertex given.

```
 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 parameter ID.

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

The degree of a vertex 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:

```
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;
}
```

The average degree of a vertex is defined as the sum all its neighbor's degrees over its own degree. For a vertex i, it is usually written as:

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

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

```
averageDegree(id: base_id): number {
  let neighbor_degree_sum = 0;

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

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

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

$$A_x = \frac{\sum D(i)}{D(x)}, i \in H(x)$$

In code:

```
averageDegree(id: base_id): number {
  let neighbor_degree_sum = 0;

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

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

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

Assortativity

Assortativity is a fairly complex algorithm. In this library, an auxiliary function was created to help calculate it:

```
edgeAverageOperationList(
    operations: ((vertices: EdgeArgs) => number)[]
  ) {
  let totals = new Array(operations.length).fill(0);
  this.edges.forEach(
    ({ vertices }) =>
      (totals = totals.map(
        (total, index) => (total += operations[index](vertices))
      ))
  );

  return totals.map((total) => total / this.edges.size);
}
```

This function borrows an important idea of functional programming. The parameter it takes in is an array of functions. The functions have the format:

```
(vertices: EdgeArgs) => number
```

They take in `EdgeArgs` and output a number. For example, an input function could be:

```
const in_function =
  ({ from, to }) =>
    this.vertices.get(from).weight +
    this.vertices.get(to).weight;
```

In essence, this function takes in two vertex IDs that belong to an edge and sums their weights. Although the function tecnically expects `EdgeArgs`, the two properties of this type that we are really interested in are `from` and `to` (the two vertices in an edge).

Assortativity makes use of `edgeAverageOperationList()`.

```
assortativity(): number {
  const [edge_multi, edge_sum, edge_sqr_sum] =
    this.edgeAverageOperationList([
      ({ from, to }) => this.degree(from) * this.degree(to),
      ({ from, to }) => this.degree(from) + this.degree(to),
      ({ from, to }) => this.degree(from) ** 2 +
                        this.degree(to) ** 2,
    ]);

  return (
    (4 * edge_multi - edge_sum ** 2) /
    (2 * edge_sqr_sum - edge_sum ** 2)
  );
}
```

We adopted the formula:

$$r = \frac{(4\langle k_a k_b\rangle - \langle k_a + k_b\rangle^2)}{2\langle k_a^2 + k_b^2\rangle - \langle k_a + k_b\rangle}$$

$\langle ...\rangle$ represents an operation performed over all edges, where k_a and k_b are its vertices.

There is another function in the Network class called edgeAverageOperation. It works almost exactly like its 'List' version, except it only takes in one operation. The list version makes assortativity $O(c)$, where c is a constant. That is because the operations are done one after the other in a single loop through all edges. If edgeAverageOperation was used instead, it would make an algorithm like assortativity $O(n)$ (linear complexity). n being the number of operations necessary, since each operation would need an entire loop through the network's edges.

Complement

The complement of a network N is a network N_c with the same number of vertices, but with all the edges N doesn't have. That is to say, let $E(N)$ be the set of edges in N, and $E(N_c)$ the set of edges in N_c, the edge e:

$$e \in E(N_c) \iff e \notin E(N)$$

The complement function has no inputs, and returns a Network object.

```
complement(): Network {
  const complement_network =
    new Network({ is_directed: this.is_directed });

  this.vertices.forEach((vertex_a) => {
    const { id: id_a } = vertex_a;
    this.vertices.forEach((vertex_b) => {
      const { id: id_b } = vertex_b;
      if (id_a !== id_b) {
        if (!this.hasEdge(id_a, id_b))
          complement_network.addEdge({
            from: id_a, to: id_b
          });
        if (complement_network.is_directed &&
            !this.hasEdge(id_b, id_a))
          complement_network.addEdge({
            from: id_b, to: id_a
          });
      }
    });
  });

  return complement_network;
}
```

It goes through every vertex and if it finds two vertices that don't have an edge in the network, it adds it to the complement.

Ego

The ego is the subgraph induced in the set $H[x]$.

The ego G_x of a vertex x from a network $N = (V, E)$ is the set $V(G_x)$ (vertices of the G_x) of all vertices that have an edge with it, as well as all the edges with vertices in $V(G_x)$.

Formally:

$$V(G_x) \subset V, E(G_x) \subset E$$

$$E(G_x) = \{e : v_e, u_e \in H_x\}$$

In the library, the algorithm takes in a vertex's ID, and returns a Network instance.

```
ego(id: base_id): Network {
  const ego_network = new Network(this.args);

  this.edges.forEach((edge) => {
    const { from, to } = edge.vertices;
    if (from === id || to === id) {
      ego_network.addEdge({ from, to });
    }
  });

  this.edges.forEach(({ vertices }) => {
    const { from, to } = vertices;
    if (ego_network.vertices.has(from) &&
        ego_network.vertices.has(to))
      ego_network.addEdge({ from, to });
  });

  return ego_network;
}
```

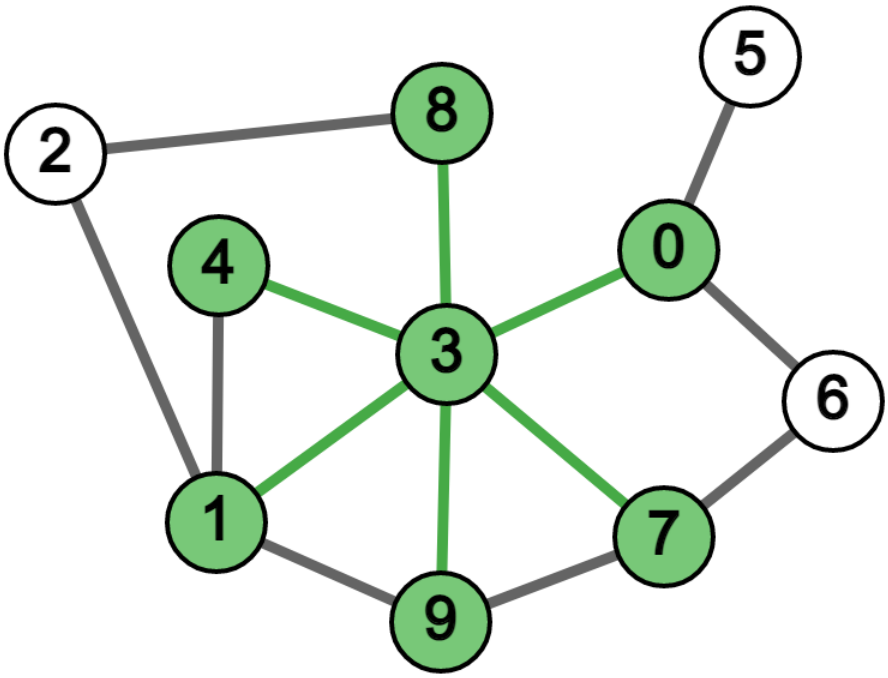

Figure 7: 3's ego includes $0, 1, 3, 4, 7, 8, 9$.

First, the algorithm goes through all edges, and add to the `ego_network` the ones that contain the given vertex. Then, it goes through the edges of the network again, this time adding edges that don't connect to the ego vertex, but connect to vertices already in the ego network.

Copy

Because of the way JS works, making a copy of a class' instance is not as simple as:

```
const net_copy = net;
```

This method works more like a reference to the original object. If a property of net changes, `net_copy` will also change.

Nevertheless, there are many other ways of making copies of objects with JS. Here are som of the most common:

```
// Destructuring:
const copy_destructure = { ...net }

// Assign:
const copy_assign = Object.assign({}, net)

// JSON:
const copy_json = JSON.parse(JSON.stringify(net))
```

However, none of these suit our need for a completely independent copy. For destructuring, although simple properties such as the maximum number of edges of the Network would be properly copied, objects (which are most of the network) would still become references. Assigning has the same issue. The edges property, for instance, would become a reference because it is a Map. So if an edge is added to the original, the copy would also receive it.

The JSON method solves this problem. It transforms the network into a JSON string, and then transforms it back into an object with JSON.parse(). The big problem with this is that the copy is no longer a network instance, just an object with many of the properties a network would have. This is an issue we would want to avoid even more when we consider typing and interfaces are precisely why TS was chosen.

Thus, the copy algorithm works differently, and is specific to the Network class:

```
copy(): Network {
  const network_copy = new Network(this.args);
  network_copy.addEdgeMap(this.edges);
  network_copy.addVertexMap(this.vertices);
```

```
  return network_copy;
}
```

Clustering

The clustering coefficient measures how connected the neighbors of a vertex are to each other. It is the number of edges that exists in between the vertex's neighbors in relation to the maximum number of edges that could exist there.

$$\frac{|e : v, u \in ev, u \in H[x]|}{|H[x]|(|H[x]| - 1)}$$

```
clustering(id: base_id): number {
  const ego_net = this.ego(id);

  if (ego_net.vertices.size <= 1) return 0;

  const centerless_ego = ego_net;

  // Max edges in a network without the given vertex.
  centerless_ego.removeVertex(id);
  const { max_edges } = centerless_ego;
  const existing_edges = centerless_ego.edges.size;

  // If graph is directed, multiply result by 2.
  const directed_const = this.is_directed ? 2 : 1;

  return directed_const * (existing_edges / max_edges);
}
```

The algorithm makes use of the `ego()` function, removing the ID vertex after. It also uses a tenary operator because the only difference between the clustering from a directed network to an undirected one is that the former has its clustering multiplied by 2.

Average Clustering

This function calculates the average clustering of the network. If calculates the clustering of all vertices, inserting its values into an array. It then reduces the array, summing all of its values. Finally, it returns the average, which devides the sum by the number of vertices in the network.

```
  averageClustering(): number {
  let average_clustering = 0;

  if (this.vertices.size <= 1) return average_clustering;

  const clustering_sum = this.vertex_list
    .map((vertex) => this.clustering(vertex.id))
    .reduce((prev, curr) => prev + curr);

  average_clustering = clustering_sum / this.vertices.size;

  return average_clustering;
}
```

Core

A k-core decomposition of a network N is a subgraph C_k with any $v \in V(N)$ with $D_v < k$ removed.

$$C_k = v, e : v \in V(N), D_v \geq k, e \in E(N)$$

It is a function that only ends after all vertices with degree less than k are removed.

```
  core(k: number): Network {
  const k_decomposition = this.copy();

  while (k > 0 && k_decomposition.vertices.size > 0) {
    let { vertex_list } = k_decomposition;
    let vertex_counter;
    for (
      vertex_counter = 0;
      vertex_counter < vertex_list.length;
      vertex_counter++
    ) {
      const current_vertex =
        k_decomposition.vertex_list[vertex_counter];
      if (k_decomposition.degree(current_vertex.id) < k) {
        k_decomposition.removeVertex(current_vertex.id);
        vertex_list = k_decomposition.vertex_list;
        vertex_counter = 0;
      }
    }
    k--;
```

```
  }

  return k_decomposition;
}
```

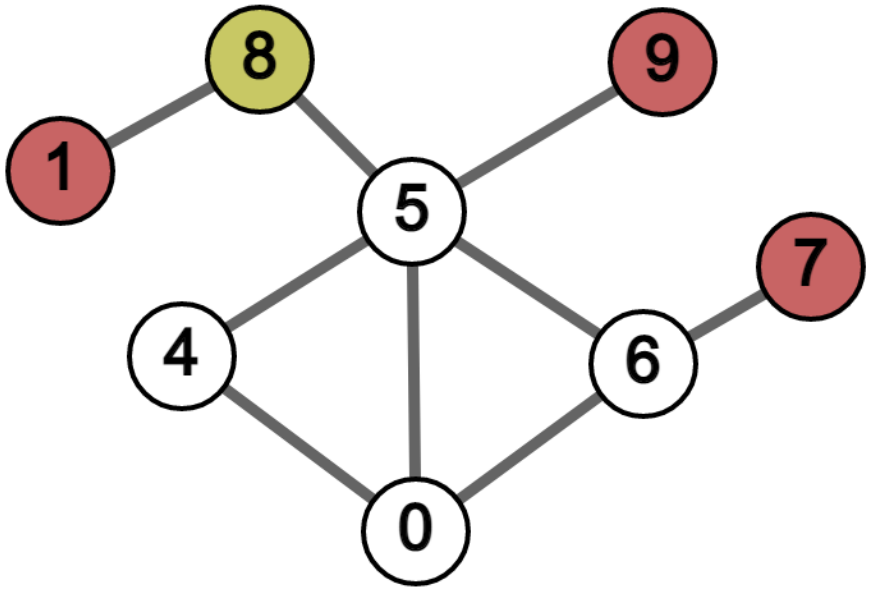

Figure 8: A 2-core decomposition would remove nodes $1, 9, 7, 8$. 8 would be removed in the second iteration of the function.

Triplets

A triplet is a set of 3 vertices and 3 edges connecting said vertices.

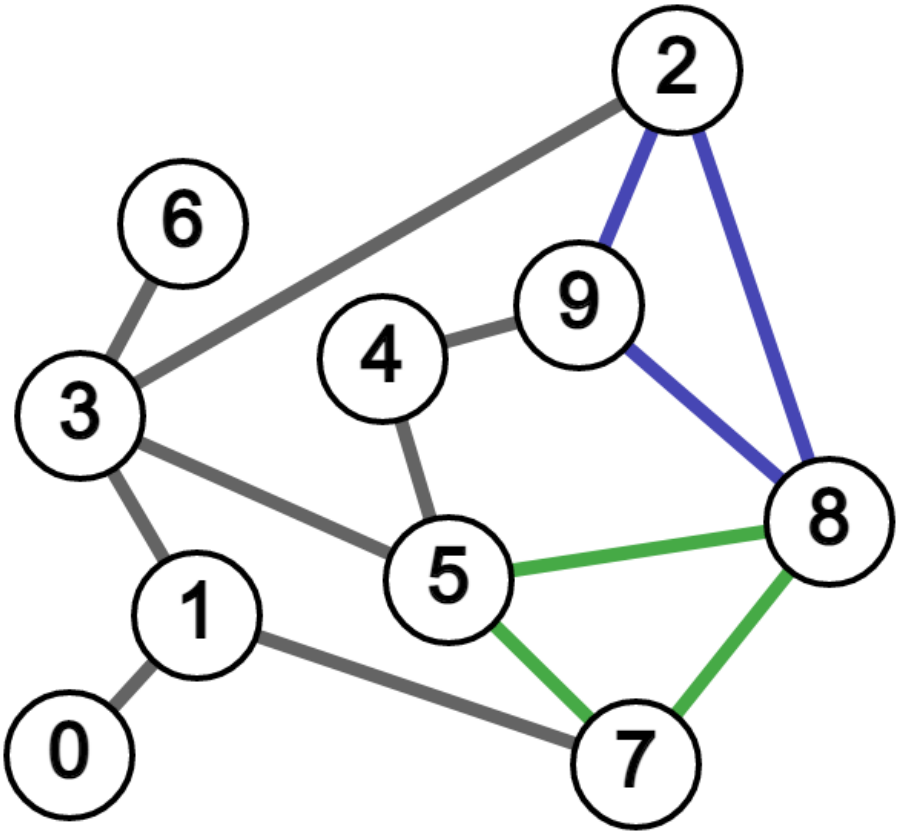

Figure 9: This network has two triplets: $\{5, 7, 8\}$ and $\{2, 8, 9\}$.

This algorithm in the network went through many iterations that improved its performance. The first algorithm looked through all edges, and, while in each edge, looked through all nodes and checked for any that had edges with the two nodes in the edge. The most glaring problem with this is that it ends up each triplet three times: once for each edge.

The solution was to order the triplets so that, internally, the algorithm sees $5, 8, 7$ as different from $5, 7, 8$, for example. This was initially done by comparing the triplet that could be created with the sorted version of that triplet. The current improved version only needs to check if the current vertices are already sorted:

```
k2.isSameTriplet(triplet, triplet.sort())
```

The current iteration of this algorithm also does not go through all vertices in the network. It only looks through the neighbors of one of the vertices in the

edge being analysed. This also saves us from having to check if both vertices have edges to the vertice beeing looked at.

```
triplets(): Triplet[] {
  const triplet_list: Triplet[] = [];

  const k2 = this.core(2);

  const { vertices, edges } = k2;

  edges.forEach((edge) => {
    const { from, to } = edge.vertices;
    k2.neighbors(from).forEach((id) => {
      if (edge.hasVertex(id)) return;
      const triplet: Triplet = [id, ...[from, to].sort()];
      if (k2.isSameTriplet(triplet, triplet.sort()))
        if (k2.hasEdge(id, to, true))
          triplet_list.push(triplet);
    });
  });

  return triplet_list;
}
```

Overall, there was a 27% reduction in the algorithm's by only using sorted triplets; then we saw around 55% impovement due to only looking through the neighbors; finally, there was a 10% improvement in performance by not having to edge-check one of the vertices. All of these were cummulative.