Network Science Library in Typescript

Soka University of America

Rodrigo Hiroto Morais

April 2022

Contents

Introduction

This capstone consists of two parts. The first part is a Network Science library, *NeTS*, which was created using TypeScript and 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. Libraries are very important, and make up essential parts of any software project. NeTS, specifically, can be used for research that relates to Network Science, or be imported into other programs or projects to serve as a base for any kind of computation that involves graphs. NumPy is an example of an important Python library widely used by programers. It assists those who use Python for dealing with more complex data structures, and contains built-in math functions that make computations simpler.

Graph Theory is an entire field of mathematics dedicated to studying graphs. Graphs are a very specific kind of mathematical structure that will be formally explained later in this chapter. On a most fundamental level, graph theory is about connections. Network Science is a field that applies the many ideas of Graph Theory to the real world by giving meaning to the connections it studies. Most things that involve conections and links, can be analyzed through the lenses of Network science. From road networks and subway systems, to exchange markets, or even organ donations.

The second part of this capstone 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 and Graph Theory will also be provided in later chapters as they become relevant.

In the Types and Interfaces chapter, we give an overview of all the fundamental data structures and definitions of the library. After that, the chapter for Values and Functions describes simpler algorithms of NeTS, and also provides background on how these algorithms can be applied through Network Science. We will then explain the library's more complex algorithms as well as some of the decisions that went into writing them as they currently are.

Lastly, we list further improvements and extensions that could be done with the library.

NeTS is created following its older version, written in the JavaScript (JS) programming language. That version, Net20, was originally coded for the Spring 2020 Network Science class at Soka University of America. Net20 had many flaws and inefficiencies which are addressed by the NeTS library.

Both NeTS and Net20 are open-source and can be accessed at [5] and [6], respectively.

NeTS does not have a visualization tool as of the writing of this document. All images shown here were created using Net20's visualization tools unless stated otherwise.

Technical Aspects

JavaScript (JS) is a multi-paradigm programming language. It is the most-used language in the web [2]. 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 (also called ES21), and is already implemented in most modern browsers. For more information, see [3].

Typescript (TS) is a strongly-typed programming language that builds on JS [1]. NeTS is made specifically for dealing with networks, which are a special kind of mathematical object with very well defined properties. Thus, TS's type functionality serves the purpose of modeling networks very well.

Basic Graph Theory

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

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

An edge is a two-element set that cointains two elements from V. Formally, we can write that as:

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

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

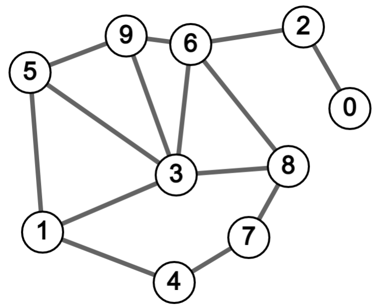

Figure 1: Example of an undirected network.

Graphs are usually defined by the types of edges it can have. Some of the ways in which we can classify graphs are:

1. Directed or undirected: This means an edge can have a direction associated with it. A directed network contains edges that are directed from one node to another. NeTS deals with both directed and undirected networks. A directed network could be used to represent a road network, where the roads are represented by edges, and intersections are the vertices. An undirected network could be used to represent friendships in a college campus, where the edges reflect whether two people are friends or not, and the nodes represent the students. A common use for Network Science is the analysis of social networks. With a social network, each user is usually represented as a node in the network. Edges can be used to express connections: follows, interactions, friendships. A specific example of an undirected network is Facebook. Users can either be friends or not. In contrast to that binary relationship of Facebook friendships, we can observe 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.

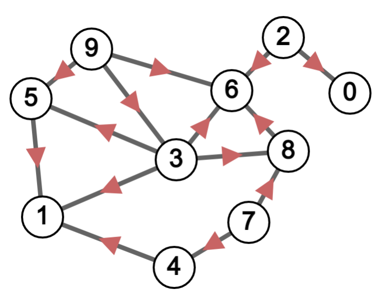

Figure 2: Network in previous example with randomized directed edges.

2. Self-loops: Networks can also have self-loops, meaning for an edge $e = \{x, y\}$, $x = y$. Self-loops are used in state-machines.

3. Multigraph: Networks can be classified as multigraphs, meaning for $e_1, e_2 \in E$, e_1 and e_2 have the same nodes, but represent different edges. Back to the road network example, a two-way street that connects two intersections could be represented as two different edges that connect the same two nodes.

4. Simple graph: simple graphs are undirected graphs with no self-loops, and no multiple edges. NeTS only deals with simple graphs. Nevertheless, there are some functions that are setup with compatibility with multigraphs in mind. Thus, although multigraphs are beyond the scope of this capstone, in the future, NeTS could be adapted to work with multigraphs.

5. Weighted or Unweighted: Traditionally, a weighted network means the edges in the network are weighted. However, it is also possible to weight vertices. Continuing with the example of social networks, we can take a look at Facebook's Messenger. An edge between two nodes (users) could represent whether the users have ever sent messages to each other, while the number of messeges exchanged could be represented by the edge's weight. On a different example, the weight of a vertex could represent the GDP of countries in a network of global trade where an edge between two countries represents if they trade. NeTS considers all networks to be weighted on a technical level. This means that, when created, any edge or vertex has their weight set to 1. An unweighted network is thus just a network with all weights set to the default of 1. Visually, the weight of an edge is usually represented as thickness.

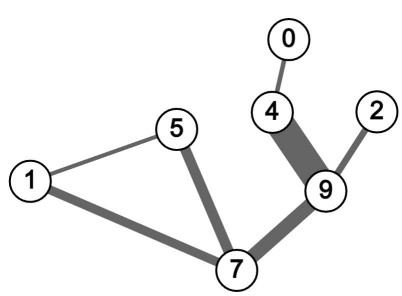

Figure 3: A network with weighted edges. In the example of Messenger, we could say that 5 and 4 have never messaged exchanged messages, while 4 and 9 have exchanged many messages.

For more information on Network Science see [4].

Types and Interfaces

In this chapter, we give an overview of the fundamental data structures and definitions used in the library. These sections contains type definitions. Type definitions are what differentiate TS from JS. With them, we are able to precisely define what properties objects have.

However, before we get into this chapter, it is useful to understand some of the naming conventions that will be used. Class, types and interface definitions follow `PascalCase`. Local variables, instance variables, and getters follow `snake_case`. Functions follow `camelCase`.

In this chapter, we will take a look at the most basic types and interfaces: `base_id`, `VertexArgs`, `EdgeArgs`, `NetworkArgs`. We will also describe the basic structure of the `Vertex`, `Edge`, and `Network` classes.

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;
```

Given a vertex in a graph, it is necessary to assign an ID it. This helps us identify what vertex we are are refering to when we want to do certain operations or calculations with the vertex. The type `string | number` means that the ID of an vertex can be a string such as `"vertex_a"`or a number such as 42.

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

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 set to one by default. This means that, on a technical level, all networks are weighted. The difference between an unweighted network and a weighted network is that the weighted network contains weights other than one. Edges, just like vertices, also have IDs.

Because NeTS doesn't deal with multigraphs, we could have gone without edge IDs. This architecture is an example of how NeTS has been prepared with the possibility of dealing with multigraphs.

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

Notice that all properties of `NetworkArgs` are optional. Therefore, it is possible to create a network without using any parameters.

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

The following is an example of code that creates a network `net`, adds the vertices '1' and 'b' to it, and an edge between them. It is visually represented in Figure 4.

```
const net = new Network()

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

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

The functions `addEdge()` and `addVertex()` are explained in the Functions section.

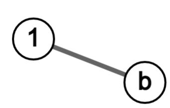

Figure 4: 'net' with an edge between '1' and 'b'.

In NeTS, networks are by default undirected. A directed network has to be explicitly declared. In the following code, we create a directed network, and add an edge that connects '1' to 'b'.

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

Notice that adding an edge automatically adds the nodes associated with it. This behavior is related to the `do_force` property and is further discussed in the Functions section.

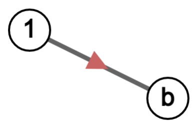

Figure 5: A directed edge from '1' to 'b'

`ParsedCSV` and `ERROR` are types used internally by the library to load CSV files and manage error messages, respectively. NeTS has features for reading and writing CSV files that are very useful for analyzing network data. A very common way of storing the information of a network is through an adjacency matrix. Adjacency matricies are most often written into a CSV file. Because NeTS has the ability to read adjacency matrices, it is possible to import a network that was created elsewhere and use it with the library.

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

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

Vertex and Edge Classes

In this chapter, we will see the two base classes used by NeTS: `Vertex` and `Edge`.

The vertex class receives an object with the interface of `VertexArgs`. The weight is optional and set to 1 if not provided as a parameter.

```
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`, `??` 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 TS and in JS. 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 one vertex each, and a weight. 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. Its two vertices and ID cannot be changed.

Changing the vertices of 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` properties 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 in NeTS they can only be set in the creation of a network. NeTS sets those limits to be `edge_limit=500` and `vertex_limit=500` by default. 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, we set limits to the number of edges and vertices. This helps because it prevents users from getting into infinite loops that try to add too many edges or vertices.

Note the `is_multigraph` property. Recall that in the introduction we mentioned that although multigraphs are beyond the scope of the capstone, the library was setup so as to allow for multigraph support in the future. The `is_multigraph` is such an example of "future-proofing".

```
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;
  }
}
```

Note that, although there are many properties being set in the `Network` class constructor, they are all optional and have defaults in case the user does not provide them. 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
```

The values and properties discussed in this section are: getters for arguments, vertices and edges; weight getters; negative and positive weight getters; maximum number of edges; and network density.

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. The `copy` function is discussed in detail in the Algorithms chapter.

```
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, which are quite efficient.

```
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

The weight of a network is the sum of the weight of its edges.

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

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

$$\sum w(e)$$

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

Similarly, we can calculate the `vertex_weight` of a network by summing the weights of its vertices.

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

There are five other functions related to vertex and edge weight. For vertices, we have `negative_vertices`, `positive_vertices`, and `zero_vertices`. They return a list with all the vertices in the network that have negative, positive, and zero edge weights, respectivelly.

For edges, `negative_edges` and `positive_edges` return a list with all edges that have negative and positive weights, respectivelly.

Edges don't have an equivalent to `zero_vertices` because by convetion it can be said that two vertices not having an edge between them is the same as saying the edge between them has a weight of zero.

This can be illustrated with the example of a network that represents global trade. Instead of GDP, we could say the weights of each vertex represents their GDP growth. A 0-GDP growth (thus, 0-weight) country could theoretically exist, and should still be represented as it could have trades with other countries. However, a 0-edge between two countries would mean that they don't trade between each other at all. Therefore, it is the same as the edge not being represented.

We don't keep track of zero-weight edges (edges that don't exist) because that could mean heavy storage usage complexity for larger networks. Nevertheless, one of the most common ways to store networks is the adjacency matrix. In it, even edges that don't exist end up being represented.

```
get negative_vertices(): Vertex[] {
  const { vertex_list } = this;
  return vertex_list.filter((vertex) => vertex.weight < 0);
}

get positive_vertices(): Vertex[] {
  const { vertex_list } = this;
  return vertex_list.filter((vertex) => vertex.weight > 0);
}

get zero_vertices(): Vertex[] {
  const { vertex_list } = this;
  return vertex_list.filter((vertex) => vertex.weight == 0);
}

get negative_edges(): Edge[] {
  const { edge_list } = this;
  return edge_list.filter((edge) => edge.weight < 0);
}
```

```
get positive_edges(): Edge[] {
  const { edge_list } = this;
  return edge_list.filter((edge) => edge.weight > 0);
}
```

Maximum Number of Edges

The largest number of edges M_n a graph N with $n = |V|$ vertices can have is:

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

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

A complete network K_n is a network with n vertices where all edges that could exist, do exist. Thus, the maximum number of edges for a network with n vertices is $|V(K_n)|$.

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

Density

The density D of a graph with n nodes and edge set E is defined by:

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

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;
}
```

A graph with low density is called a sparse graph. It essentially tells many links exist as compared to how many there could be at most.

Functions

This section deals with functions. Functions serve to package a set of instructions. For example, in this chapter we will see functions that serve to remove or add edges and vertices. In the case of edge removal, the function `removeEdge` packages the instructions necessary to remove an edge from the network.

Networks are often used to represent real world phenomena that are constantly changing; thus, it is important to be able to dynamically remove edges and nodes from a network. In the aforementioned example of a social network, this could mean someone deleting their account.

The functions for removing edges and vertices remove said elements of the network. The function `removeEdge`, in particular, is quite straightforward. It finds an edge with the given `from` and `to` nodes, and then deletes them from the edge Map.

```
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 `removeVertex()` function differs itself from `removeEdge()`. When a vertex is removed, all of the edges associated with it also have to be removed. Returning to the social network example, we can say that, when a user removes their account, their relationships to the remaining users are also 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. Maps are data structures that consist of key-element pairs. Keys are unique, while elements can be repeated. Both keys and elements can consist of any data (strings, numbers, objects, arrays, etc). In respect to the key-elemtn pairs, this data structe resemble a Python dictionary.

The function `hasVertex()` takes advantage of a vertex ID to quickly find whether a network contains a vertex with the specified 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 a list that fulfills the given property. Here, we can see how the conversion from an edge map to a list of edges can be useful.

The `edgeBetween()` function will return the edge with the given nodes if it exists, otherwise it is undefined.

```
/**
 * 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)
```

```
  );
}
```

To check if two edges are the same (if they have the same `from` and `to` properties), the private function `checkEdgeIsSame()` is used.

A private function can only be accessed inside the class declaration, and is not meant to be used outside of it.

The `checkEdgeIsSame()` function will take in two edges and compare their nodes (`from` and `to`). If a network is directed, `from` is compared with `from`, and `to` is compared with `to`. If it is undirected, the comparison between `to` and `from` also needs to be performed. Thus, the function also receives an `is_directed` property.

```
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 two special ways to add edges and vertices to a network. The first one is with `addEdgeMap`, which uses an edge Map to add edges to the network. This is useful in the case where we want to use the edge map of a different network to add edges. The second special way is with `addEdgeList`, which takes in a list of `EdgeArgs`, and uses the arguments to create new edges. These exist as a shorthand to add vertices and edges. It is useful to be able to add multiple edges to a network at the same time.

```
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))
  );
}
```

As an example, an array of edges could be added as such:

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

```
net.addEdgeList(edge_list)
```

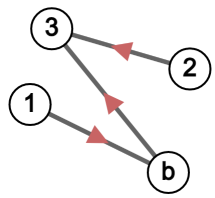

Figure 6: Our previous 'net', now with the list of edges added.

This abstracts the complexity of adding multiple edges at the same time from the user.

There are three other functions that serve utilitarian purposes and are used throughout NeTS. The function `hasEdge` returns whether an edge with the given nodes exists or not in the network.

The function `getEdgesBetween` serves a similar purpose, but instead of returning whether it exists or not, it returns a list with the edges between the two given nodes. This function is yet another example of the NeTS being prepared with the possibility of multigraphs in mind. For simple graphs (non-multigraphs), two vertices will always have only one edge between them. Nevertheless, `getEdgesBetween` can return a list with all the edges that exist, something that would only happen in a multigraph.

The function `addVertex` adds a vertex to the network using the given arguments.

```
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. For example, it handles what happens when an edge that connects a node to a node that doesn't exist in the network. It does so with using the `do_force` argument.

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. Otherwise, if `do_force===false`, it throws an error.

```
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));
}
```

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 two private utility functions used in algorithms. They have a similar purpose to `hasEdge` in the sense that it also checks to see if a particular structure exists in the network. They will be further 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

An algorithm here refers to a process or set of processes and calculations or other problem-solving operations that are performed by a computer. Algorithms are the most important part of NeTS, and any other network science library. They are useful for demonstrating and understanding fundamental aspects of networks. As we will see, they can express how connected a social network is, or even show a profitable exchange path in a given market. In this chapter, we will look at functions that perform algorithms to calculate various values or to find particular structures in the network.

Neighborhood

A vertex x's neighborhood $H(x)$ is defined as the set of vertices that are connected to that vertex by an edge. In a social network, the neighborhood of a vertex represents the immediate connections of a user. In a network that represents the global exchange market for commodities, the neighborhood can be used to represent the trade partners of a given country. 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;
}
```

This function 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.

The `inNeighbors` and `outNeighbors` functions work in a similar fashion to the `neighbors` function. The difference being that they return a list with neighbors that fulfill particular properties.

```
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. Let E_x denote the set of edges containing a vertex x, and D_x denote the degree of x. Then,

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

The degree of a vertex is, in other words, the number of neighbors for said vertex. In the example of a social network, the degree of a user would tells us how many connections that user has in the network.

In this section, we explain the functions related to degree and average degree for undirected and directed networks.

For an undirected network:

```
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 in NeTS are related to in and out-neighbors, and are only defined when a network is directed. The function `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.

For a directed network that represents global debt, let vertices represent countries or international organizations and edges be directed from borrower to lender. We could predict that the IMF, for example, has a high out-degree, as it is a money lender for many countries. Countries with high in-degree would be those with many debts to pay.

```
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 x, it is usually written as:

$$k_{nn}(x) = \frac{\sum a_{ix} D(i)}{D(x)}$$

Where i represents a vertex in the network and the sum is performed over all vertices in the network. a_{ix} is 1 if there is and edge between vertices i and x and 0 oterwise. And this is what this would look like when literally transcribed into code:

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

  this.vertices.forEach(({ id:check_v }) => {
    if (this.hasEdge(id,check_v)) neighbor_degree_sum += this.degree(to);
  });

  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 average degree A_x of vertex x as:

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

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 makes it clearer what the `averageDegree` function is supposed to do. If someone who doesn't know what an average degree is were to look at the assortativity function, they would have an easier time figuring out what it means. This is also faster to compute, as we don't need to iterate over all edges, we only need to go through the neighbors of a vertex.

Assortativity

Assortativity is a fairly complex algorithm. It can be defined in broad terms as the tendency of vertices in a network to connect to other vertices that are similar in some way. Here, the similarity we are considering is in respect to the degree of the vertices in a network.

In this library, auxiliary functions were 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 from 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).

We adopted the formula:

$$r = \frac{(4\langle k_a k_b \rangle - \langle k_a + k_b \rangle^2)}{2\langle k_a^2 + k_b^2 \rangle - \langle k_a + k_b \rangle}; k_a, k_b \in e, \forall e \in N(E)$$

The expressions inside $\langle\rangle$ represent an operation performed over all edges that is then averaged, where k_a and k_b are the endpoints of each 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)
  );
}
```

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 $O(n)$, where n is the number of edges in the network. 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 $O(nk)$ (linear complexity), where k is 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 only with 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 then for any 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;
}
```

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

Ego

The ego is the subgraph induced in the set $H[x]$. $H[x]$ is used to define the set with neighbors of x as well as x itself: $H[x] = H(x) + x$. 4 illustrate this. The ego network of node 3 is highlighted in green.

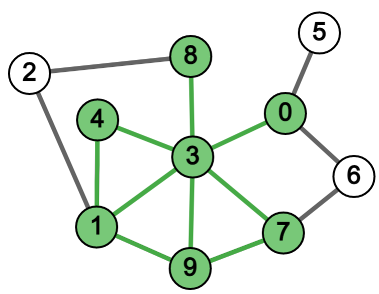

Figure 7: 3's ego includes $0, 1, 3, 4, 7, 8, 9$ and all edges between them.

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, for $V(G_x) \subset V$ and $E(G_x) \subset E$:

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

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;
}
```

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

It is very useful to have a copy of a network when we want another network that derives from it. For example, a core-decomposition (as we will see next) makes use of the copy function.

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 some 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;
}
```

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 recursively. This means, we remove all vertices with degree less than k, then in the resulting network, we remove all vertices with degree less than k, and so on until there are no vertices where $D_v < k$.

$$C_k = (V_c, E_c)$$

$$V_c = \{v : v \in V(N), D_v \geq k\}, E_c = \{e : e \in E(N)\}$$

The core function only ends after all vertices with degree less than k are removed. It returns a network class instance.

```
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;
}
```

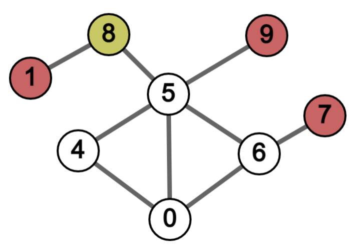

Figure 8: A 2-core decomposition would remove nodes $1, 9, 7, 8$. 8 would be removed in the second iteration of the function.

Clustering

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

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

```
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 coefficient 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;
}
```

Triplets

A triplet is a set of 3 vertices and 3 edges connecting said vertices. Triplets, also called triangles, triples or K_3s, are the foundation of design theory, an entrire field of mathematics. Design theory is concerned with decomposing graphs into triplets.

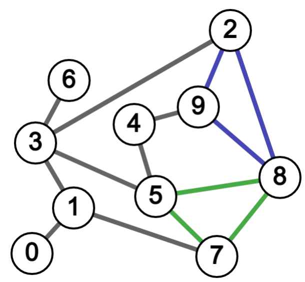

Figure 9: This network has two triplets: $\{5, 7, 8\}$ and $\{2, 8, 9\}$.

The triplets algorithm in NeTS 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 counts 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 improved version only needs to check if the current vertices are already sorted:

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

The current iteration of the triplets 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 with 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;
}
```

Further Work

Currently there are three more obvious ways to improve or extend NeTS functionality: adding multigraph compatibility; adding IO functionality to the library; and adding more algorithms to NeTS.

The one which will probably require most work is adapting the library to handle multigraphs. Although NeTS does not have such capability, some of its functions and error handling were created with the possibility of addressing multigraphs in a future update in mind. As an example, we can look at the way edges are handled. They are stored in a Map data structure, meaning each edge has a unique ID. This was done so that in the case multigraphs are addressed, their property of multiple edges between two nodes can be handled by the already existing edge system.

To handle multigraphs, more work would also have to be done in regards to the already existing algorithms detailed in the Algorithms chapter. While we were creating the functions for triplets, for instance, we realized that addressing this algorithm for multigraphs could mean not just exceptions, but the creation of an entirely new function that is exclusive to multigraphs.

Adding IO options to the library would be an important update to NeTS. Although only mentioned in passing before, NeTS already has the capability to import and export networks to CSV files that contain an adjacency matrix. Further IO options would include the creation and customization of network images. These images could be similar to the images taken from Net20 that were used in this document. Because of this precedent, adapting code and ideas from Net20's image processing functions could help with NeTS' update.

Finally, we can extend NeTS' functionality by adding new algorithms. We already have a list of algorithms to implement in a future release of the ilbrary. Two of interest here are optimized quadruplets (4-cycle) algorithm, and breadth-first-search (BFS). BFS is an extremely important algorithm for applied graph theory. It is used for path-finding. When a GPS tries to find the shortest path from one place to another, it is most likely using BFS or a variation of it. BFS ended up not making into NeTS' first release, but it was present in Net20. Thus, the code from Net20 could be adapted and improved to fit into NeTS. The quadruplets algorithm refers to a 4-cycle search algorithm. Much like the triplets algorithm present here, it searches for vertex cycles. However, instead of looking for cycles of 3 nodes, it finds cycles of length 4. We are interested in 4-cycle specifically, because it is possible, just like we did for triplets, to create algorithms for cycle-finding that are optimized for quadruplets specifically. We could also add a general cycle-finding algorithm, but it wouldn't be as fast as what we could create by targeting 4-cycles with an algorithm.

Bibliography

[1] Handbook - the TypeScript handbook.

[2] JavaScript | MDN.

[3] ECMA International. ECMAScript® 2023 language specification.

[4] F. Menczer, S. Fortunato, and C.A. Davis. *A First Course in Network Science*. Cambridge University Press, 2020.

[5] Rodrigo Morais. rodigu/NeTS. original-date: 2022-01-29T22:27:49Z.

[6] Rodrigo Morais. rodigu/network2020.