CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Matemáticas e Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Betweenness centrality in transport networks. A comparison between static and temporal approaches.

Autor: Jorge Galindo Morata

Director: Aristides Gionis

MADRID, JULIO 2019

1 Resumen

Los grafos, o redes, están compuestos por un conjunto de nodos conectados por aristas, estos nodos pueden representar distintas entidades entre las cuales hay una relación binaria representada por las aristas. Además, estas redes pueden ser usadas para modelar una gran cantidad de procesos que pueden suceder en las entidades que representan.
Las redes normalmente son estáticas, no se considera su evolución respecto al tiempo. Si consideramos dicha evolución tenemos redes temporales, en las cuales las aristas solo existen en ciertos momentos temporales, los cuales pueden ser tiempos discretos o intervalos. Con esto representamos como evoluciona la relación binaria entre las entidades que forman el grafo. Esta información condiciona la definición y métodos para obtener ciertas medidas en las redes, como por ejemplo las medidas de centralidad, que representan cuanto de importante es un nodo en la red; y más especificamente, la centralidad de intermediación en los caminos más cortos "shortest-path betweenness centrality", en la cual un nodo se considera importante si es probable que para dos nodos aleatorios, este primer nodo se encuentre en el camino más corto entre ellos.
En este trabajo comparamos los resultados para esta medida usando su definición en redes estáticas y temporales usando como red proveniente del mundo real las redes de transporte público de Madrid y de Helsinki dando como conclusión en que casos dan mayores ventajas cada modelo.
Además, por último, buscamos proponer nuevos métodos que puedan suponer un compromiso entre ambos modelos o mejorarlos de alguna manera.

Betweenness centrality in transport networks: A comparison between static and a temporal approaches.

Jorge Galindo Morata

July 31, 2019

Abstract

Graphs or networks are a set of nodes connected by edges. They can represent several different entities and pairwise relations between them. Moreover, graphs can be used to model a great number of natural and anthropogenic processes. The most common graph model is a static graph. In this kind of graphs, the nodes and edges are not modified over time. Such representation can be sufficient for networks such as friendship or road networks as these evolve at a low velocity over time compared to the processes that can be studied over these networks, such as transfer of information or traffic respectively. On the other hand, some networks can change fast over time, making the different edges only appear at certain times. For example, public transport or contact networks throughout a period of time. This temporal information is important as, for example, in the first case the times of departure of the different transports are needed when planning routes, and in the second, transfer of information can only happen towards contacts taking place after receiving the information.
This report aims to compare results using both static and temporal approaches when measuring network centralities, i.e., how important is a node or edge in the network. We focus on a particular kind of centrality, namely shortest-path betweenness centrality, and we use for this task real-world networks, in particular, different transport networks. As a conclusion, we use the results obtained applying these two different approaches to indicate which of them is more advantageous to use depending on the context. We also seek to propose several new approaches that could represent a compromise between the two approaches or improve them in different manners.

2 Networks

A graph is a set of elements, called nodes or vertices, which have some pairwise relation. This relation is represented using edges. Edges are pairs of nodes which are connected by this relation. Graphs can be classified depending on

whether the relation is symmetric (i.e., having a node i connected to a node j by this relation implies that j is also connected to i). If this happens, we say the graph is undirected. In the opposite case, we classify the graph as directed. The graphs can have a numerical value assigned to each edge, in which case we call it a weighted graph, otherwise every edge has a value 1 as its weight, and we call it unweighted. If the vertices or edges have some attributes, for instance, a name, then we can call the graph a network[1]. This notation described follows recent papers like [1].

This report focuses on weighted directed networks as these are the most general kind, as unweighted networks can be considered weighted networks in which every edge has the same weight, and undirected networks can be considered directed networks in which every edge exists in both directions. Further in the report, if it is not specified, all networks are directed and weighted.
Some examples of networks are:

- The roads networks between cities. These networks can be represented as undirected or directed if we consider one-way roads, although they are not typical. It can be weighted using, for instance, the length of the roads.
- Social networks of a set of people, having as relation whether they interacted with each other over a period of time or not. This kind of network is undirected and can be weighted using the number of times they interact with each other or the amount of time they do it. It could be considered directed if, instead of interacting face to face, you take into consideration sending emails for example.

There are several ways to represent a graph, for instance:

- Adjacency matrix: a matrix A in which each element a_{ij} indicates, if $a_{ij} \neq 0$, the existence and weight of the edge connecting vertex i to vertex j or its non existence otherwise. If the graph is unweighted each element can only be either 1 or 0, and, if the graph is undirected, the matrix is symmetric.
- Adjacency list: a map from each vertex to a list that indicates the vertices connected to this one, and its weight in the case of weighted graphs.
- Edge list: a list of vertex pairs and weights in the case of weighted graphs. Each pair indicates an edge in the graph.

2.1 Temporal network

Networks, as described in the previous chapter, do not model any changes over time. Thus, they are also called static networks. To overcome this lack of temporal information, we consider a second type of network, namely, temporal networks:

[1] `https://en.wikipedia.org/wiki/Network_theory/`

Definition 1 *A temporal, or time-varying network is a network such that its edges only appear at certain time moments or intervals.*

Definition 2 *A static network is, in opposition to a temporal one, one in which all the edges are permanent.*

When measuring different properties about temporal networks, or their elements, we need to consider the time in which the edges exist. This consideration strongly affects how to measure these properties. The simplest way to obtain these results is by using the aggregated network. The aggregated network of a temporal network is a static network whose set of edges contains all the edges that appear in the temporal network at some point in time with its average weight if it varies over time. We can obtain aggregated networks of either the whole interval in which the temporal network exists or of a snapshot of a smaller interval of time. Taking shorter snapshots can help to approximate temporal measures in the graph. However, regardless of the length of the snapshot, this approach always loses information on how the network evolves while it is traversed. This information can be important for several different measures depending on the structure of the network. Therefore, the measures obtained with this approach are inaccurate and might give essentially-different results to the actual values. In opposition, if we use the temporal network, we cannot compute the same measures as in static network, we need to define a generalisation of such measure for temporal networks.

Some examples of intrinsically temporal networks are:

- The public transport networks, as there are only connections between 2 stops at certain times.
- Networks requests between computers and servers.
- Networks of arrivals to a set of places (considering as node both the items arriving and the places nodes in the network).

We can consider two kinds of temporal networks. First, there are those whose edges are instantaneous, i.e., their duration is negligible with respect to the duration of the network. In opposition, there are those whose edges last a non-negligible interval of time. These are known respectively as contact sequences and interval graphs or networks [4]. If we consider discrete times, we can represent interval graphs as contact sequences. To perform this translation, we need to create a new contact edge per time point covered by each interval edge. Using temporal points instead of intervals simplifies the algorithms over the network. However, depending on the discretisation of the time and how long these intervals are, the number of edges can largely increase, thus, making the algorithms take excessively long times to complete.

Temporal networks can be represented using similar structures to the ones used for static networks. The only change we need to make is to add the

temporal information to the adjacency list or the edge list. In contrast, using adjacency matrices requires more complex generalisations. For instance, one of the simplest generalisations uses a time-dependent adjacency list $A(t)$, where each entry $A(t)$ is the adjacency matrix at time t. Another possible generalization would be considering the set of nodes $V = \cup_t V_t$, with V_t the set of nodes at each time t, we then need to connect each node v_{it} to its time-successor v_{it+1} and we can generate a supra-adjacency matrix, using this connections and the connections of the static graph at each timestep [7].

3 Network centralities

Definition 3 *A measure of centrality describes how important are vertices or edges from the graph by assigning each of them a numerical value, giving higher values to more important vertices or edges.*

Depending on what is considered an important node or edge, as there is no single definition for this, we can consider several distinct measures a centrality measure.
Some interesting centralities are the so-called path-based centralities[1]. To explain these centralities, we need the concept of "path", for which is necessary to define "walk" first:

Definition 4 *A walk is a succession of nodes from the network in the order in which they are visited. To be able to move between two nodes, u and v, there has to be an edge $\{u, v\}$.*

This concept can be used to describe flows in traffic networks, transfer of information or transfer of diseases, between others.

Definition 5 *A path is a special case of walk in which each node appears only once.*

In temporal networks we are interested in a special kind of paths and walks, time respecting paths and walks:

Definition 6 *A path or walk in a temporal network is considered time-respecting if each of the edges used in the path or walk is active at some point in time after arriving at its starting node.*

There are several centralities defined using either paths or walks:

- Degree centrality: The number of paths using one edge that contains the vertex; as its name implies it corresponds to the degree of the vertex, i.e., the number of edges that connect this node. If the network is directed we

distinguish between in-degree and out-degree centralities, for edges leading to and coming from each node respectively.

- Eigenvector centrality: The relative growth in the number of walks coming into each vertex when its length goes to infinite, or, in other terms, how likely it is to be in a certain node after this infinite walk. It is computed as the left dominant eigenvector of the adjacency matrix, i.e., a vector v such that $vA = kv$, having A the adjacency matrix, and k the largest scalar in absolute value that keeps this property.

- Shortest-path betweenness centrality: Defined as follows in static graphs:

$$C_B(v) = \sum_{u \neq w \neq v} \frac{\sigma_{u,w}(v)}{\sigma_{u,w}} \tag{1}$$

In this equation, $\sigma_{u,w}$ is the number of shortest-paths between vertices u and w and $\sigma_{u,w}(v)$ the number of shortest-paths between u and w that pass by v. Intuitively, for a vertex v, this measure is the sum of proportions of shortest-paths between each other two vertices that pass by v. A shortest path is a path with the minimum length between the two vertices. The length of a path or a walk is considered to be the sum of the weights of all edges in the path or walk in static networks.
There are several generalisations to compute this measure in temporal networks [6, 5]. We implement the one explained in [5], which is defined as follows:

$$C_{B,[i,j]}(v) = \sum_{i<=t<=j} \sum_{u \neq w \neq v} \frac{\sigma_{u,w,[t,j]}(v)}{\sigma_{u,w,[t,j]}} \tag{2}$$

Here, i and j are the times in which the network starts and ends respectively and $\sigma_{u,w,[t,j]}$ and $\sigma_{u,w,[t,j]}(v)$ the time respecting paths between u and w contained in the interval $[t, j]$. We need to loop over time as, if we only took paths starting at the beginning of the interval, then we can lose information about how the shortest-paths evolve over time. For instance, one node can be highly connected at the beginning of the interval, and have high centrality at that point. However, if it then is poorly connected during the remaining duration of the network, this node should have a low centrality over the complete interval. If we don't loop over time, the algorithm loses this information and will assign the node a high centrality. Thus, we need to consider the evolution of shortest-paths over time[5]. This report also compares this generalisation with another possible generalisation in which we don't restrict the paths to finish in the interval, but only to start in it:

$$C_{B_\infty,[i,j]}(v) = \sum_{i<=t<=j} \sum_{u \neq w \neq v} \frac{\sigma_{u,w,[t,\infty]}(v)}{\sigma_{u,w,[t,\infty]}} \tag{3}$$

4 Scope of the study

This project focuses on betweenness centrality and its use in public transport networks. As this kind of network is intrinsically temporal, we compare the results obtained using the aggregated network over the whole period, the aggregated network over different intervals of the network and the temporal network, again over both the whole period and smaller intervals.
The public transport networks are modelled using as the set of vertices $V = \cup_i \{v_i\}$, where each v_i corresponds to each of the different stops in the transport network. We use $E = \cup_i \{e_i\}$ as the set of edges, where each edge $e_i = \{u, v, t, d\}$, here u and v are the nodes corresponding to two consecutive stops for a route in the transport network, t the time of departure of a vehicle from u, and d the duration of the journey until v.
To compare between these approaches we have several requirements:

- Obtain suitable data from different public transport networks for creating these networks.
- Design or adapt and programs and algorithms suitable for this purpose.
- Generate different graphics to visualise the results obtained.

5 Algorithms

There are several possible algorithms for computing shortest-path betweenness centrality on static graphs:

- Floyd-Warshall algorithm [3] is a well-known algorithm to compute all shortest-paths in a network by computing a shortest-path distances matrix M in which each element m_{ij} corresponds to the distance from node i to node j. To reconstruct the paths, we use the second matrix S in which we store the successors of each node in the path, each position s_{ij} storing the first nodes of each shortest-path from i to j.
  M is initialised with all the positions outside the diagonal to infinity, and the positions in the diagonal to 0, as this is the distance from each node to itself. Then, for each edge $\{i, j, w\}$ we set $m_{ij} = w$. Afterwards, we iterate over $[1, |V|]$, where V is the set of vertices, with 3 iterators: i, j and k, if $M_{ij} > M_{ik} + M_{kj}$ we update the distance from i to j with this second value. At the same time, we update the matrix S, in which all the positions have been initialised to $\{\}$, an empty list. Then for each edge i, j, w the position s_{ij} is updated with $\{j\}$. Furthermore, every time that we update M, we also update S with the rule $s_{ij} = s_{ik}$. Finally, if we find positions i, j, k such that $M_{ij} = M_{ik} + M_{kj}$, then we append s_{ik} to s_{ij}. With this matrix, we can reconstruct any path from i to j by updating i with its successor s_{ij} until we reach j, and using these, compute the centrality using formula (1). Its running time is $O(|V|^3)$, and its space requirement is $O(|V|^2)$.

- Brandes' algorithm [2], although it is originally defined only for unweighted graphs, as this algorithm is based on the breadth-first search algorithm (BFS), which is an algorithm that finds shortest paths starting from a source node on unweighted graphs, it can be generalised for weighted graphs, modifying it to use Dijkstra's algorithm instead of BFS. In Brandes' algorithm, we repeat BFS starting from each vertex s in the set of vertices V. We call this node s the source node. In each iteration of BFS with a different source, we need to store some information for each vertex $v \in V$, namely, a list of its predecessors, its distance from the source and the number of shortest paths that reach this node. Additionally, we need a queue and a stack to control the order of the nodes being visited from the source. These are initialised in each BFS iteration as follows:
  - The stack $S = \{\}$, an empty stack.
  - The predecessor list $pred(v) = \{\}$, an empty list,$v \in V$
  - The distance $dist(v) = \infty, v \in V, v \neq s, dist(s) = 0$
  - The number of paths $numPaths(v) = 0, v \in V, v \neq s, numPaths(s) = 1$
  - The queue $Q = \{s\}$, a queue with only the source in it.

  Likewise, we need to initialise an array to store the computed centralities. This is initialized at the beginning of the algorithm having $cent(v) = 0, \forall v \in V$. When all the necessary data for the BFS iteration has been initialised we iterate until the queue is empty. In each iteration, we dequeue the first element v from the queue and push it to the stack. Then, we iterate over the neighbours of v and, if the neighbour u was not reached yet, i.e. $dist(u) == \infty$, then $dist(u)$ is set to $dist(v) + 1$. Then, if $dist(u) == dist(v)+1$, $numPaths(u)$ is set to $numPaths(u)+numPaths(v)$, and v is appended to $pred(u)$. When the queue is empty, we will have finished the iteration of BFS. However Brandes' algorithm, instead of computing the centrality after having all the paths computed as Floyd-Warshall algorithm does, it computes the contribution of the shortest paths from the source s that has just been iterated. Computing these contributions allows the algorithm to save the space that would be required to store these paths. To achieve this we initialize a dependencies array, $dep(v) = 0, v \in V$. At this stage, the stack will contain all the elements that BFS was able to reach from the source for this iteration in non-increasing distance order. Iteratively, we pop elements from the stack until it is empty, and for each popped element w we iterate over $pred(w)$, the predecessors of w, computing for each predecessor v $dep(v) = dep(v) + \frac{numPaths(v)}{numPaths(w)} * (1 + dep(w))$. After iterating over these predecessors, and, as w will not appear again as the predecessor of another node since they are stored in non-decreasing distance order, we then know the contribution to the centrality of w for this iteration and can compute $cent(w) = cent(w) + dep(w)$. Its running time is $O(|V|^2 log(|V|) + |V||E|)$ when using a weighted generalisation, $O(|V||E|)$ on unweighted graphs, and its space requirements is $O(|V|)$

Between these two options, Brandes' algorithm has less running time and space requirements. For this reason, and, as explained before, to be able to use it on weighted graphs, we use a modified version of this algorithm using Dijkstra's algorithm instead of BFS. To achieve this, the queue needs to be replaced by a priority queue using the distance to the source as the key. Then, when dequeuing elements from the queue, as in Dijkstra's algorithm the shortest path may not be the first path found, we need to compare the distance to the node dequeued and skip it if it is not the most recent one. Due to this same reason, when exploring the neighbours of a node v, we need to update the distance to each neighbour node u, reached by the edge $\{v, u, w\}$ not only if $dist(u) == \infty$, but also if $dist(u) > dist(v) + w$. Furthermore, when this happens, we need to reset the predecessor list of u to an empty list. Additionally, the rule to add a new predecessor to the predecessor list transforms into $dist(u) == dist(v) + w$, only needing to be added if it was not in the predecessor list yet.
As we want to represent public transport networks, and how they evolve over time, we need to represent how the intervals between vehicles change. As this only affects the first stop of the route, we need to introduce a second modification to the algorithm. This modification consists of adding a special kind of node from which all the paths have to start. This modification allows us to create copies of all the stops in the network from which the paths have to start. In these copies, we modify the weight of all its edges summing to it the expected waiting time before the vehicle arrives at the stop, instead of having only the time to complete the journey between the two stops. This approach misses the waiting time when taking transfers, but it still shows better how the network changes over time than without considering these special path start nodes. To accomplish this, the only change required in the algorithm is to control that the source node is a special path start node before starting the BFS iteration. The pseudocode for the modified Brandes' algorithm for static weighted networks is shown in Algorithm 1.

To further generalise this algorithm for its use on temporal networks, we need to repeat the algorithm once per discrete time considered, and to modify Dijkstra's algorithm for it only to accept time-respective paths. To reach the neighbour u of a node v, which has been reached at time t, while maintaining time-respectiveness of the path we need to iterate over all the edges v, u, t', w skipping all edges whose t' is such that $t' < t$. Then, we keep the edge that minimises $t' - t + w$ using this quantity as effective weight of the edge. This is the edge that allows us to reach u from v at the minimum time possible. A pseudocode for the temporal implementation of Brandes' algorithm is shown in Algorithm 2.

Having to iterate over the time allows us to choose a timestep parameter which allows us to exchange accuracy for performance, the lower the timestep, the higher the accuracy of the measure. One issue to consider with this parameter is that it is not possible to compare the absolute measures obtained with different timesteps. This issue appears because a lower timestep implies

Algorithm 1 shortest-path betweenness centrality algorithm for static weighted networks

```
1: cent_B[v] ← 0, v ∈ V
2: UsesSpecialRouteSartNode ← true|false
3: for s ∈ V do
4:     if !UsesSpecialRouteSartNode||s.specialRouteStartNode then
5:         stack ← empty stack
6:         queue ← empty priority queue of pairs [v ∈ V, distance]
7:         predecessors[v] ← empty list, v ∈ V
8:         nPaths[v] ← 0, v ∈ V
9:         dist[v] ← infinity, v ∈ V
10:        nPaths[s] ← 1
11:        dist[s] ← 0
12:        queue.enqueue([s, dist[s])
13:        while queue not empty do
14:            [v, vDist] ← queue.dequeue        ▷ Vertex with smallest distance.
15:            if vDist == dist[v] then      ▷ An element can be inserted more
16:                                  than once, so the queue can have old values.
17:                stack.push(v)
18:                for [w, weight] edge of v do
19:                    if dist[v] + weight < dist[w] then
20:                        dist[w] ← dist[v] + weight
21:                        queue.enqueue([w, dist[w])
22:                        nPaths[w] ← nPaths[v]
23:                        predecessors[w] ← [v]
24:                    else if dist[v] + weight == dist[w] then
25:                        nPaths[w] ← nPaths[v]
26:                        for pred ∈ predeccessors[w] do
27:                            if pred! = v then
28:                                nPaths[w] ← nPaths[w] + nPaths[pred]
29:                        if v ∉ predecessors[w] then
30:                            predecessors[w].push(v)
31:        dependency[v] ← 0, v ∈ V
32:        while stack not empty do
33:            w ← stack.dequeue                          ▷ Furthest element.
34:            for v ∈ predecessors[w] do
35:                dependency[v] ← dependency[v]+(nPaths[v]/nPaths[w])(1+dependency[w])
36:            if  thenw ≠ s
37:                cent_B[w] ← cent_B[w] + dependency[w]
```

more iterations, which causes all the centralities to be larger than when using a larger timestep. Therefore, it is only useful to compare the centralities after normalising them.
A further consideration when obtaining the minimum valid edge is the organisation of the data. Having the edges ordered in increasing-time order allows to skip the edges whose time is already past, i.e., they are not valid anymore, and the valid edges $\{u, v, t1, w1\}$ such that another valid edge $\{u, v, t2, w2\}$ exists such that $t2 + w2 < t1$, so it is clear that the first edge cannot be optimal. The pseudocode of the algorithm to find the minimal valid edge exploitting this ordering is shown in Algorithm 3.

6 Studied networks

The networks used for this project are the transport networks from Madrid[2] and Helsinki[3] obtained in GTFS format[4]. Madrid's network is obtained in several datasets corresponding to each kind of transport, whereas Helsinki's is obtained in a single dataset.
GTFS format defines several files containing tables of comma separated values (CSV) with information about the routes, the trips going over them, the different stops and their locations, the exact times in which the transports from each trip go over these stops, and in the case that the transports don't follow exact times, the frequencies in which they go over the trips.
This format doesn't allow to study the network directly as the information needed for this is spread over several CSV files. Therefore, before studying the networks, we need to transform them into a format which allows computing network measures in a less complex manner. For the purposes of this report, we choose to use an edge list due to its simplicity to create.

7 Implementation

First, we implement a translator between GTFS and edge list formats. The resulting edge list is printed out in CSV format.
This translator[5] is a console application implemented using vc++17. It takes as input the folder containing the GTFS network. Furthermore, it can also accept additional arguments which allow to define whether the output will be a temporal or aggregated network, over which weekdays and hours the output network spans, if the output will have links added between proximate stops which would represent the possibility of walking between them, the output folder, and several possible extra columns that can be printed to the output CSV file aside from

[2] `http://data-crtm.opendata.arcgis.com/`
[3] `https://transitfeeds.com/p/helsinki-regional-transport/735`
[4] `http://gtfs.org/reference`
[5] `https://gitlab.com/jorgegalinmor/gtfstoedgelist`

Algorithm 2 shortest-path betweenness centrality algorithm for temporal weighted networks

```
1: cent_B[v] ← 0, v ∈ V
2: UsesSpecialRouteSartNode ← true|false
3: for s ∈ V do
4:     t ← NetworkStartTime
5:     while t < networkEndTime do
6:         if !UsesSpecialRouteSartNode||s.specialRouteStartNode then
7:             stack ← empty stack
8:             queue ← empty priority queue of pairs [v ∈ V, distance]
9:             predecessors[v] ← empty list, v ∈ V
10:            nPaths[v] ← 0, v ∈ V
11:            dist[v] ← infinity, v ∈ V
12:            nPaths[s] ← 1
13:            dist[s] ← 0
14:            queue.enqueue([s, dist[s])
15:            while queue not empty do
16:                [v, vDist] ← queue.dequeue ▷ Vertext with smallest distance.
17:                if vDist == dist[v] then ▷ An element can be inserted more
18:                                         than once, so the queue can have old values.
19:                    stack.push(v)
20:                    for w neighbour of v do
21:                        e ← minValidEdge(v, w, t)        ▷ Psudocode for this
22:                                                           function in Algorithm 3.
23:                        distW ← e.time − t + e.weight + dist[v])
24:                        if distW < dist[w] then
25:                            dist[w] ← distW
26:                            queue.enqueue([w, dist[w])
27:                            nPaths[w] ← nPaths[v]
28:                            predecessors[w] ← [v]
29:                        else if distW == dist[w] then
30:                            nPaths[w] ← nPaths[v]
31:                            for pred ∈ predeccessors[w] do
32:                                if pred! = v then
33:                                    nPaths[w] ← nPaths[w] + nPaths[pred]
34:                            if v ∉ predecessors[w] then
35:                                predecessors[w].push(v)
36:            d[v] ← 0, v ∈ V                              ▷ Dependency vector.
37:            while stack not empty do
38:                w ← stack.dequeue                        ▷ Furthest element.
39:                for v ∈ predecessors[w] do
40:                    d[v] ← d[v] + (nPaths[v]/nPaths[w])(1 + d[w])
41:                if  thenw ≠ s
42:                    cent_B[w] ← cent_B[w] + d[w]
43:        t ← t + networkStep
```

Algorithm 3 Minimal temporal edge between two vertices

```
1: procedure MinValidEdge(Vertex v, Vertex w, Scalar t)
2:     minEdge ← null
3:     minT ← infinity
4:     if edges[u, w] has permanent egde then
5:         minEdge ← edges[u, w].permanentEdge
6:         minT ← minEdge.weight
7:     for edge ∈ edges[u, w].tempEdges do    ▷ tempEdges has to be ordered
8:                                              regarding to the time of the edges.
9:         if edge.time > tAND(edge.time − t) + edge.weight < minT then
10:            minEdge ← edge
11:            minT ← dge.time + edge.weight
12:        else if edge.time − t > minT  then
13:            break loop
       return minEdge
```

the default ones: the origin and end stop IDs, the duration of the edge and the time in which it is present, in the case of temporal networks. These possible extra columns include the name of the trip, route or stop or the coordinates of the stop.

The second program we implement [6] is also a console application implemented in vc++17. It computes the betweenness centrality on the networks. First, it reads an input network in edge-list format and stores it in memory as an adjacency list, abstracting whether it is a temporal or static network. Then, it computes the shortest-path betweenness centrality using the appropriate algorithm depending on the network being temporal or static. Finally, it prints a list of the nodes and their centrality. This list is ordered in a decreasing centrality fashion. Moreover, it also prints a normalised version of these centralities in a second file. The centralities are normalised so that they follow a Gaussian distribution.
If the input is a temporal network, the program allows defining as a command line argument the timestep the algorithm uses to compute the temporal centralities. Moreover, it is also possible to define, using a command line argument as well, an interval size used to compute the centrality during subintervals, of this given size, of the network. The result of this option is essentially different from obtaining the network for this subinterval and computing its centrality as our algorithm when using this first approach computes the alternate definition of temporal betweenness centrality (3).

Finally, we implement a set of tools to visualise the obtained data[7] using MATLAB. It comprises three functions that read the lists of centralities and prints

[6] `https://gitlab.com/jorgegalinmor/betweennesscentrality`
[7] `https://gitlab.com/jorgegalinmor/matlabcentralityplotting`

them in a map, in a histogram of the biggest centralities, and in histograms of the centralities that change the most between the different approaches and accuracies used. This software makes use of two third-party MATLAB functions[8] [9].

8 Results

In this section, we describe the obtained results using some of the plots obtained with MATLAB[10]. The networks obtained and plotted correspond to the temporal and aggregated networks of the whole week, and for each four hours interval.

8.1 Helsinki

First, we compare the aggregated network over the whole week with the networks for smaller four hour intervals using their maps. It is observed that the normalised centrality barely changes over most of the different intervals when compared to the whole one, shown in Figure 1. It only changes in a noticeable manner in the 0-4h and 20-24h intervals. We show the first of these intervals in Figure 2. In this interval, some of the visible paths of high centrality nodes are missing, and some of them change due to night hours. During the remaining intervals, the network only changes in the waiting intervals from the first stop in each path, which causes their centralities to stay very similar. If we change the way the network is generated, and we model the waiting times when changing transport, this aggregated approach might show further differences between the intervals, although they would, in a very likely way, still remain close similar.

[8] https://se.mathworks.com/matlabcentral/fileexchange/27627-zoharby-plot$_{}$google$_{}$map

[9] https://se.mathworks.com/matlabcentral/fileexchange/23573-csvimport

[10] https://gitlab.com/jorgegalinmor/centralityplots

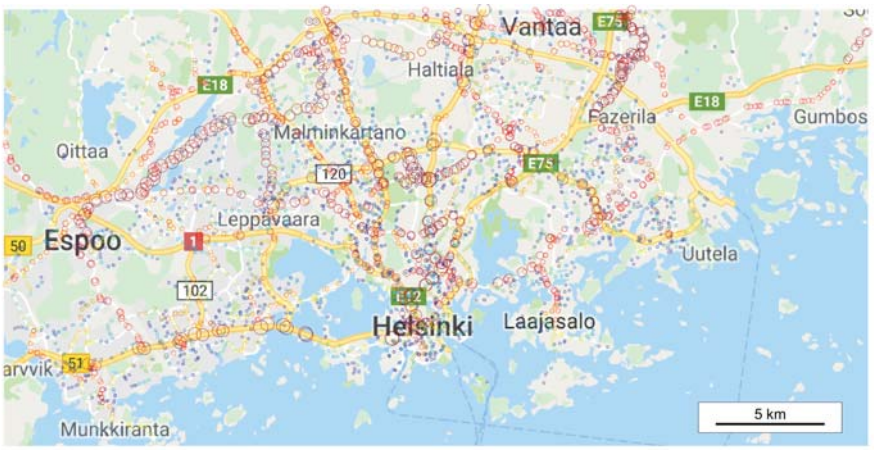

Figure 1: Static centralities over the full interval. Higher radius and warmer colour indicate higher centrality.

If we look at the histograms of these intervals (Figures 3, 4), we can see that the most important node in the aggregated network over the whole interval, Leppävaara, is only the most important in one of the fragments. Moreover, some nodes, for instance, Kamppi (Kaukoliikenneterminali), which is the fourth most important node in the network for the whole interval, does not appear again between the ten most important ones for any of the four hours interval networks. These cases can be explained by the fact that in some of these histograms we can see a second stop at Leppävaara or Kamppi. As they are very proximate, it is possible for them to interchange their centrality charge in an easy manner. Although we show only two of these intervals, the remaining ones can be checked in the referenced repository containing all the obtained plots.

Figure 2: Static centralities over the interval 0-4h.Higher radius and warmer colour indicate higher centrality.

Secondly, we compare the centrality from the aggregated network for the whole interval with the temporal centrality for this same interval, shown in the map in Fig. 5. As we can see by comparing Fig. 5 with Fig. 1, centrality is condensed in smaller zones. This condensation allows us to see more clearly which zones are more important. It is also possible to see a swift of the centrality between some of the biggest roads to some evident paths located about the middle between several pairs of major roads. These paths correspond to the train stations. This is explained by the fact that train is probably the fastest mean of transport in the public transport network, and it has fewer stops for a given distance than buses or tram. Hence, the fastest routes between farther places tend to use some other mean of transport to get to a train station and then does most of the travel by train. This information gets lost when accumulating the network, which disables us from seeing this same high centrality paths. This is possibly caused because the trains are not as frequent as other means of transport, even though they are faster. This can cause the static algorithm to prefer usually the routes using other kinds of transport instead of the ones using trains, even though they would be preferable at many times. There's also another clear path in the bottom part of the map that didn't appear before, although having smaller centrality than the trains, which corresponds to the different metro stations, which would be the second fastest mean of transport, having as well longer distances between stops than buses or trams.

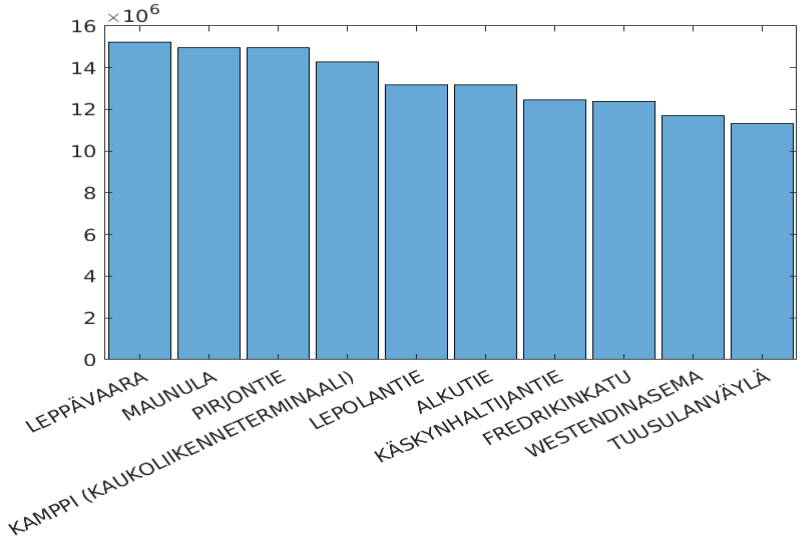

Figure 3: Static centralities over the full interval.

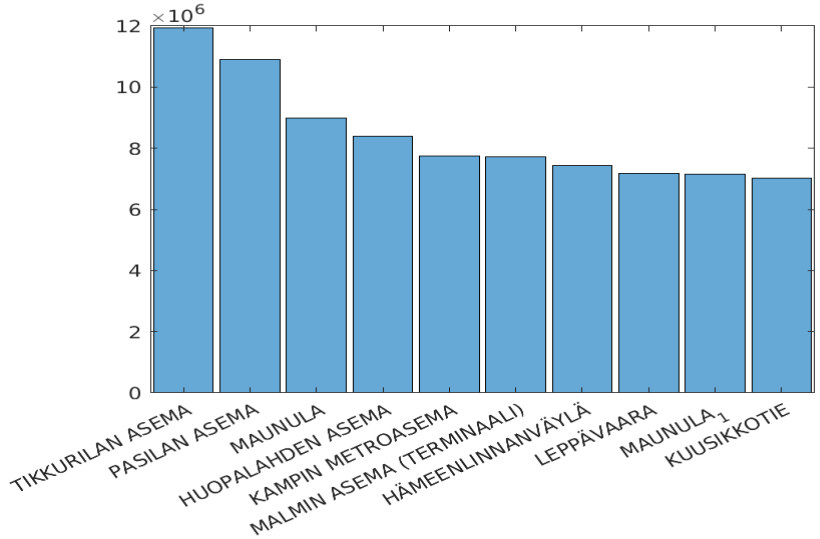

Figure 4: Static centralities over the interval 0-4h.

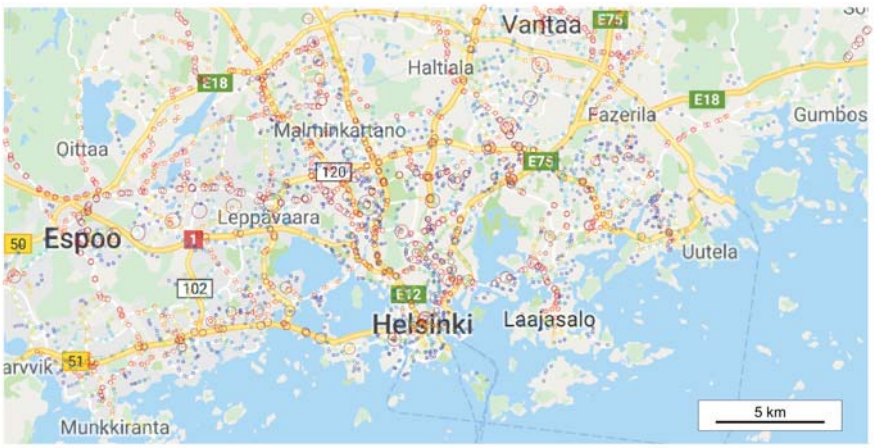

Figure 5: Temporal centralities over the whole interval.Higher radius and warmer colour indicate higher centrality.

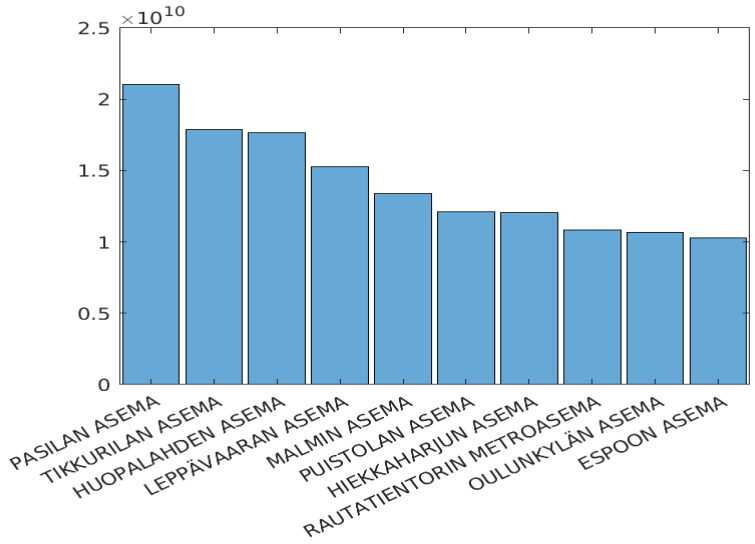

Figure 6: Temporal centralities over the whole interval.

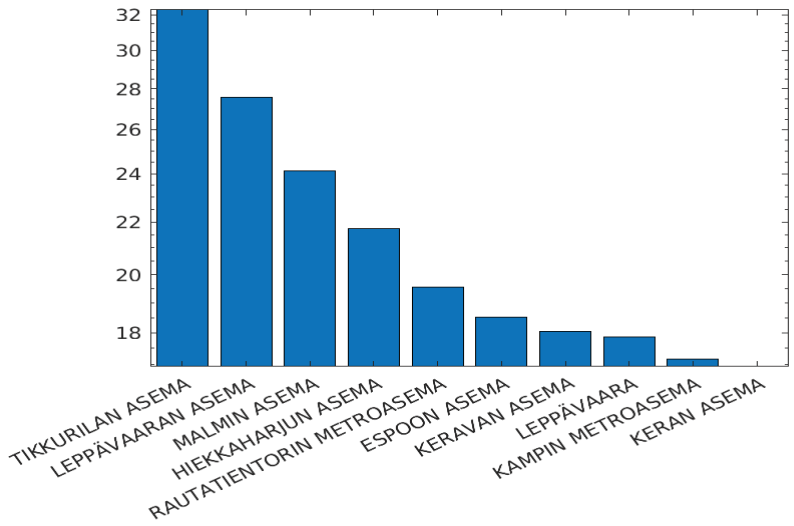

Figure 7: Increase of centralities over the whole interval, when going from static to temporal approach.

In addition, we want to check how much accuracy we lose using a larger timestep to compute the the temporal centrality. Looking at Figure 8, 9 and 10 which are respectively a map, a histogram of the increase and a histogram of the decrease of the centralities using a timestep of four hours instead of the one minute timestep used for the previous figures. We can see that the normalised centralities are practically the same as before, although some stops get to change in the ranking, but not dramatically. If we try some timestep between these previous ones, for instance, ten minutes, we can observe that, while finishing already in a more reasonable time than when using a one minute timestep, no stop changes in ranking anymore and the variation is a few exponential times lower as depicted by the histograms and maps for this timestep, which can be found in the references. Because of this, for most applications it would be ideal taking a timestep between these two limits, being possible to vary it, depending on the required accuracy.

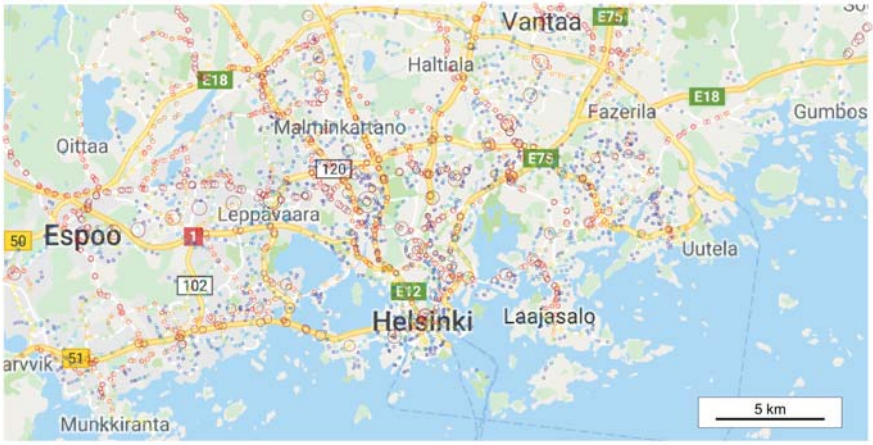

Figure 8: Temporal centralities over the whole interval with 4h timestep.Higher radius and warmer colour indicate higher centrality.

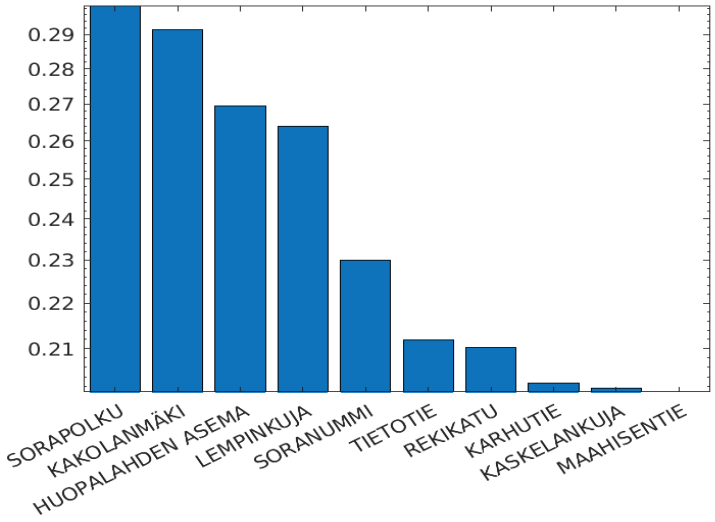

Figure 9: Temporal centralities over the whole interval with 4h timestep.

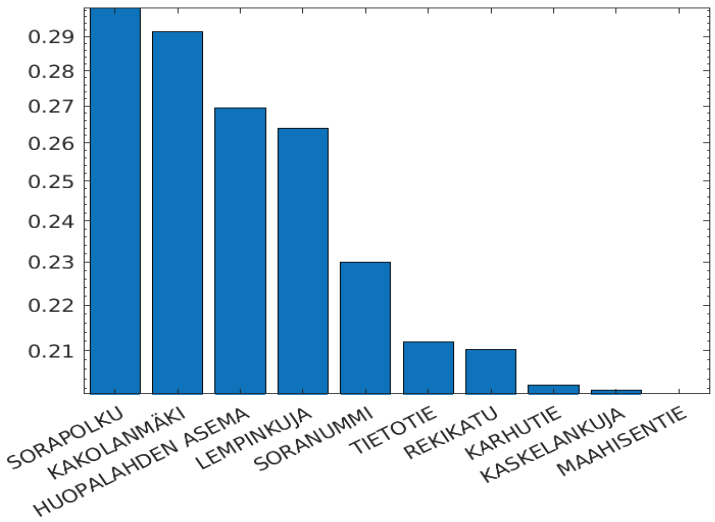

Figure 10: Variation of temporal centralities over the whole interval using a 4h timestep.

Another interesting comparison is between the temporal centralities for the whole network and the ones obtained over different smaller intervals. Most of the intervals have very similar centralities to the ones obtained with the whole interval like in the aggregated approach. An exception to this is, as happened in the aggregated networks, the centralities of the interval 0-4h. As we can see in Figure 11, the path corresponding to the metro lines does not appear anymore, and a number of bus stops either cease being used or are used very infrequently, which makes the fastest routes to be by foot in many cases, distributing more evenly the centralities around the whole map. However, the train stays as the most important mean of transport. In the rest of the intervals, everything stays mostly invariant, although there are small changes in some stops, they are not significant. In this case, the 20-24h interval also stays very similar to the whole network, unlike in the aggregated networks. That means that the night routes appearing in this interval only appear at the end of it, so they do not largely affect the centralities of the whole interval, even though they start making them change at the end of the interval. This change could be observed by taking smaller divisions in this interval.

Figure 11: Temporal centralities over the interval 0-4h. Higher radius and warmer colour indicate higher centrality.

Finally, we compare these last temporal centralities over four hours intervals, which have been computed using definition (2), with the centralities for these same intervals computed using the definition (3), which also uses paths that don't finish inside of the interval. As we can see comparing Fig. 12 and Fig. 13, which show the centralities for the interval 8-12h with the definitions (2) and (3) respectively, there are a few zones that get an increased centrality in this approach. These zones appear because there are some stops that either during the night, like in this case, or during the day, cannot be reached. Hence, all the paths that are being taken by the algorithm to reach these destinations, which cannot be reached during the interval, go to the closest reachable stop to the destination, wait there for several hours until a transport towards this destination departs, enabling then the route to reach its destination. This kind of routes are found once per timestep, and, in addition, there are a high number of shortest paths taking this kind of route to these unreachable destinations, instead of considering them unreachable. This problem gets bigger when comparing Figure 11 with Figure 14, which shows the centralities for the interval 0-4h using definition (3), because, in this case, the number of unreachable stops is higher than in any other interval, which causes it to change most of the centralities.
This problem could be solved by reducing the contribution of the path depending on how far from the end of the interval it finishes. This reduction in the contribution would avoid this kind of path from contributing too much to the centralities. However, it would still capture the centrality contribution from new paths that may be appearing at the end of the interval which wouldn't appear if we use definition (2). Capturing these paths would be especially useful if we're computing centralities for a small interval of time as this would hide all the routes that are bigger than the interval, and a high number of smaller routes,

even though they can be important shortest paths, losing all their contribution to the stops in the path.

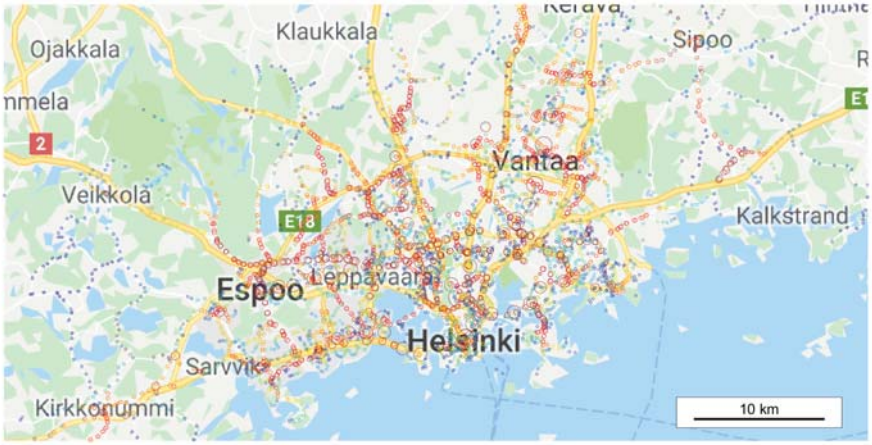

Figure 12: Temporal centralities over the interval 8-12h. Higher radius and warmer colour indicate higher centrality.

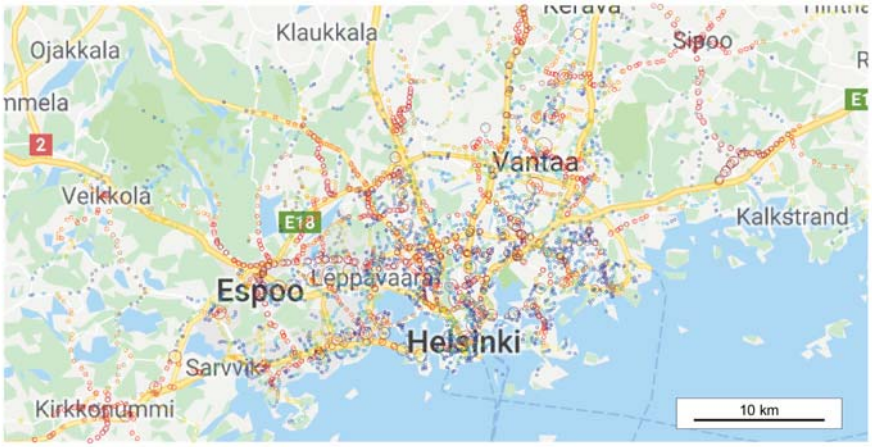

Figure 13: Temporal centralities over the interval 8-12h with the alternative definition. Higher radius and warmer colour indicate higher centrality.

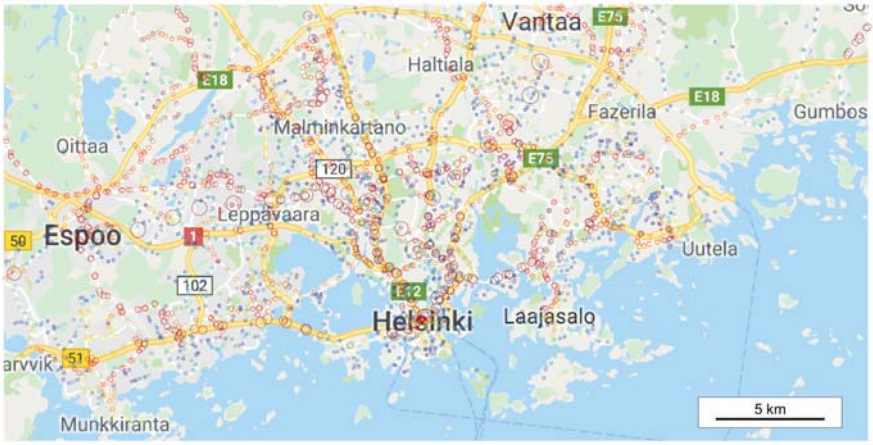

Figure 14: Temporal centralities over the interval 0-4h with the alternative definition. Higher radius and warmer colour indicate higher centrality.

8.2 Madrid

In Madrid's network, we can observe similar results to the ones obtained with Helsinki's network. One difference is that when comparing the different intervals taken we can see the centralities in the interval 4-8h are midway between the centralities in the interval 0-24h and the ones in the interval 0-4h as the night routes start between the 0 and 2 am, and stay running until about 6 am, so they are only partially reflected the interval 4-8h. In Helsinki's network, this also happened in the interval 20-24h as the night routes start already at the end of this interval, but this was only reflected in the aggregated centralities, as the contribution was too small in the temporal ones. In opposition to that case, this change in the interval 4-8h is reflected in both types of centralities as the contribution from the night routes covers approximately half of the interval, making their contribution as important as the contribution from the day routes in the temporal centralities. We show this for the temporal network in Figure 15, 16 and 17. In Madrid's network, it could additionally be interesting to check smaller intervals. This is because, unlike in Helsinki, not only the metro and tram stop during night, but also the trains, and the change between night and day routes is more progressive: buses have night routes between 11 pm and 6 am, trains don't circulate between midnight and 5 am, and metro stops at 2 am, not starting again until 6 am. Therefore, it could be interesting to measure how the centralities change in these smaller intervals between those occurrences.

As a further observation, given that the data on Madrid's network was obtained as a set of networks, one per type of transport, instead of only one network containing every type as the data from Helsinki, we can analyse each type of network individually. By doing this, one important observation that can be made is that in the case of the metro, train and tram networks, the centralities

barely change between any of the intervals and the whole network nor between the aggregated and temporal approaches. This similarity is a consequence of the network maintaining its structure unmodified, and only changing in its intervals during the day. Furthermore, these intervals are modified in a very similar way in all the different trips. If this happens, the aggregated approach for the whole network will give a good result in most cases, although it can still change the result if the network is complex enough. However, when considering the combined network including all types of transport, using aggregated or temporal approaches, and the intervals chosen does make a difference on the centralities of the stops for these, mostly static, kinds of transport because of how they are interconnected, and, most importantly, how they are connected to the bus networks, as these change their structure over time.

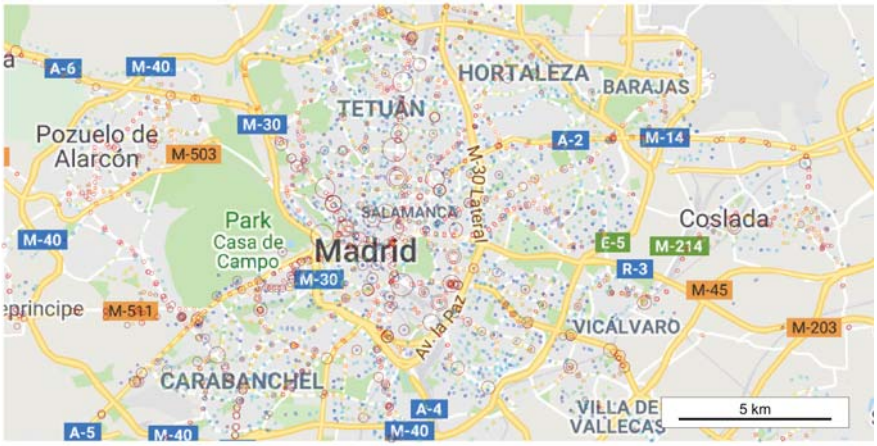

Figure 15: Temporal centralities over the whole network. Higher radius and warmer colour indicate higher centrality.

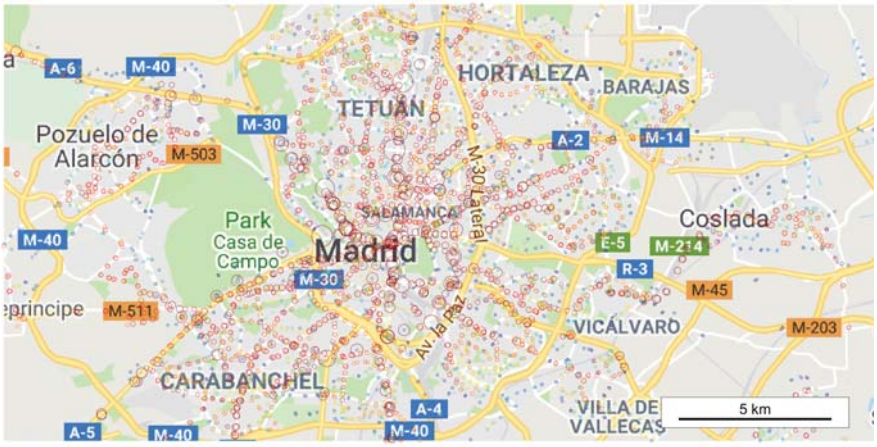

Figure 16: Temporal centralities over the interval 0-4h. Higher radius and warmer colour indicate higher centrality.

Figure 17: Temporal centralities over the interval 4-8h. Higher radius and warmer colour indicate higher centrality.

9 Conclusions

With the results provided in this work, we can observe that, if the routes in the transport network change over time and we aggregate the network, there is a great quantity of information lost. Furthermore, even if the network does not change greatly during the interval taken (for instance Helsinki 8-12h), it can still lose much information if, for instance, the shortest path between two nodes

changes rapidly. This can happen, for instance, if there is a fast route departing less frequently than a slower route. If this is the case, the static algorithm may exclusively treat the second route as the shortest path, while the first one will periodically be actually better. Missing this kind of route causes the trains not to be as important as they should in Helsinki's aggregated network. The only case in which it is not as important is if the routes don't change over time (i.e. there are no "night routes") and the intervals between transports are, to some extent, even between all the routes. This simplicity usually implies, when considering a transport network, that there is only one kind of transport or several similar ones, i.e., trams and buses are usually similar in speed and intervals.
Although the static approach results in inaccurate results when treating networks which are not simple enough, it is many times faster since we do not need to loop over time, nor search for the optimal edge between two nodes. Hence, if we are treating complex networks and there is a need for fast results even if they are rough, using the aggregated centralities is the recommended approach. On the other hand, if some more accurate results are needed, the temporal network will return considerably better results if the network is complex enough. In the case of uncomplicated networks, as explained before, both approaches will obtain similar results, and,furthermore, the temporal approach will not be significantly slower than the static approach, hence the approach taken is usually indifferent.

10 Future work

There can still be an improvement on the running times when computing the temporal centrality. Moreover, we could compare the results with different variations of both the temporal and static shortest-path betweenness centralities. Examples of these possible variations are:

- Instead of using the absolute shortest-path, we can use a number or percentage of "almost-shortest-paths" or "short-paths" This could be particularly useful for the static centralities, as this can solve the problem exposed before in which it can miss the shortest-path if it is not as frequent as one a bit slower. We show a possible pseudocode for this in algorithm 4.
- When we have some real-time data of the transport network or the data of some social network, we may want to to know the centralities around the present time or even in the near future. To accomplish this objective, the dependencies for several time points can be stored and, whenever a node becomes unreachable near the end of the known data, we use the previous dependencies to predict the centrality in that timepoint using either a regression or interpolation algorithm. The best one would be dependant on the application and the network measured. Hence, this approach would require a high amount of case-dependant testing in each

possible application. We show a possible pseudocode for this variant in algorithm 5.

References

[1] Paolo Boldi and Sebastiano Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014.

[2] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[3] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[4] Petter Holme and Jari Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.

[5] Hyoungshick Kim and Ross Anderson. Temporal node centrality in complex networks. *Physical Review E*, 85(2):026107, 2012.

[6] Vincenzo Nicosia, John Tang, Cecilia Mascolo, Mirco Musolesi, Giovanni Russo, and Vito Latora. Graph metrics for temporal networks. In *Temporal networks*, pages 15–40. Springer, 2013.

[7] Dane Taylor, Sean A Myers, Aaron Clauset, Mason A Porter, and Peter J Mucha. Eigenvector-based centrality measures for temporal networks. *Multiscale Modeling & Simulation*, 15(1):537–574, 2017.

Algorithm 4 Short-path betweenness centrality algorithm for static weighted networks

```
1: cent_B[v] ← 0, v ∈ V
2: UsesSpecialRouteSartNode ← true|false
3: for s ∈ V do
4:     if !UsesSpecialRouteSartNode||s.specialRouteStartNode then
5:         ...                                          ▷ Structures innitialization
6:         while queue not empty do
7:             [v, vDist] ← queue.dequeue        ▷ Vertex with smallest distance.
8:             if vDist == dist[v] then      ▷ An element can be inserted more
9:                                          than once, so the queue can have old values.
10:                stack.push(v)
11:                for [w, weight] edge of v do
12:                    if dist[v] + weight < dist[w] then
13:                        dist[w] ← dist[v] + weight
14:                        queue.enqueue([w, dist[w])
15:                        nPaths[w] ← nPaths[v]
16:                        predecessors[w] ← [[v, dist[v] + weight]]
17:                        for pred ∈ predeccessors[w] do
18:                            limit ← dist[w] ∗ (1 + tolerance)
19:                            if pred.first ≠ v  &  pred.second < limit then
20:                                factor ← (1 − (dist[pred]+weight−dist[w]) / dist[w])
21:                                contrib ← nPaths[pred] ∗ factor
22:                                nPaths[w] ← nPaths[w] + contrib
23:                    else if dist[v] + weight < dist[w] ∗ (1 + tolerance) then
24:                        facetor ← (1 − (dist[v]+weight−dist[w]) / dist[w])
25:                        nPaths[w] ← nPaths[v] ∗ factor
26:                        for pred ∈ predeccessors[w] do
27:                            if pred ≠ v then
28:                                factor ← (1 − (dist[pred]+weight−dist[w]) / dist[w])
29:                                contrib ← nPaths[pred] ∗ factor
30:                                nPaths[w] ← nPaths[w] + contrib
31:                        if v ∉ predecessors[w] then
32:                            predecessors[w].push([v, dist[pred] + weight])
33:        dependency[v] ← 0, v ∈ V
34:        ...                              ▷ Dependency and centrality accumulation
35:        ▷ Alternatively the tolerance condition (lines 19 and 23) could be
   dist[v] + weight < dist[w] + tolerance.(tolerance > 0)
36:        ▷ Also, the scaling factor (lines 20 and 24) could be a different func-
   tion that decreases as the distance grows
```

Algorithm 5 Predictive shortest-path betweenness centrality algorithm for static weighted networks

```
1: cent_B[v] ← 0, v ∈ V
2: UsesSpecialRouteSartNode ← true|false
3: for s ∈ V do
4:     t ← NetworkStartTime
5:     while t < networkEnd do
6:         if !UsesSpecialRouteSartNode||s.specialRouteStartNode then
7:             stack ← empty stack
8:             queue ← empty priority queue of pairs [v ∈ V, distance]
9:             predecessors[v] ← empty list, v ∈ V
10:            nPaths[v] ← 0, v ∈ V
11:            dist[v] ← infinity, v ∈ V
12:            nPaths[s] ← 1
13:            dist[s] ← 0
14:            queue.enqueue([s, dist[s])
15:            sucessors[v] ← empty list, v ∈ V
16:            timestamps ← empty queue
17:            while queue not empty do
18:                ...
19:            d[v] ← 0, v ∈ V                          ▷ Dependency vector.
20:            while stack not empty do
21:                w ← stack.dequeue                    ▷ Furthest element.
22:                for v ∈ predecessors[w] do
23:                    d[v] ← d[v] + (nPaths[v]/nPaths[w])(1 + d[w])
24:                if  thenw ≠ s
25:                    cent_B[w] ← cent_B[w] + d[w]
26:            for v ∈ V do
27:                if dist[V] == infinity then
28:                    d[v] = regression(timestamps[0..timestamps.size].first[v])
29:                    recalculate ← empty queue
30:                    for successor ∈ timestamps.last.second[v] do
31:                        queue.enqueue(successor)
32:                    while recalculate not empty do
33:                        r ← recalculate.first
34:                        reg ← regression(timestamps[0..timestamps.size].first[r])
35:                        d[r] ← 0.5 ∗ d[r] + 0.5 ∗ reg
36:                        ▷ The weights could be different, and also dependant
   on in how many timestamps the stop actually appears
37:                        for successor ∈ timestamps.last.second[r] do
38:                            queue.enqueue(succ)
39:            timestaps.enqueue([d, successors])
40:            if timestamps.size > maxTimestamps then
41:                timestamps.dequeue
42:        t ← t + networkStep
```