Fast Cluster-based Computation of Exact Betweenness Centrality in Large Graphs

Cecile Daniel (✉ cecile.daniel@univ-eiffel.fr)

Gustave Eiffel University: Universite Gustave Eiffel <https://orcid.org/0000-0001-7519-1962>

Angelo Furno

ENTPE: Ecole Nationale des Travaux publics de l'Etat

Lorenzo Goglia

University of Sannio: Universita degli Studi del Sannio

Eugenio Zimeo

University of Sannio: Universita degli Studi del Sannio

Research

Keywords: Complex networks analysis, Betweenness centrality, Distributed computation, Big data processing

Posted Date: March 23rd, 2021

DOI: <https://doi.org/10.21203/rs.3.rs-321493/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.
[Read Full License](#)

RESEARCH

Fast Cluster-based Computation of Exact Betweenness Centrality in Large Graphs

Cecile Daniel^{1*}, Angelo Furno¹, Lorenzo Goglia² and Eugenio Zimeo^{2*}

*Correspondence:

cecile.daniel@univ-eiffel.fr;

zimeo@unisannio.it

¹LICIT UMR T9401, University
Gustave Eiffel, University of Lyon,
ENTPE, Lyon, France

²Dept. of Engineering, University
of Sannio, Benevento 82100, Italy

Full list of author information is
available at the end of the article

Abstract

Nowadays a large amount of data is originated by complex systems, such as social networks, transportation systems, computer and service networks. These systems can be modeled by using graphs and studied by exploiting graph metrics, such as Betweenness Centrality (BC), a popular metric to analyze node centrality of graphs. In spite of its great potential, this metric requires long computation time, especially for large graphs. In this paper, we present a very fast algorithm to compute BC of undirected graphs by exploiting clustering. The algorithm leverages structural properties of graphs to find classes of equivalent nodes: by selecting one representative node for each class, we are able to compute BC by significantly reducing the number of single-source shortest path explorations adopted by Brandes' algorithm. We formally prove the graph properties that we exploit to define the algorithm and present an implementation based on Scala for both sequential and parallel map-reduce executions. The experimental evaluation of both versions, conducted with synthetic and real graphs, reveals that our solution largely outperforms Brandes' algorithm and significantly improves known heuristics.

Keywords: Complex networks analysis; Betweenness centrality; Distributed computation; Big data processing

Introduction

The massive amount of data available today in many domains is often originated by complex systems that can be modeled as graphs (*e.g.*, social networks, transportation networks, computer networks, service networks, *etc.*) where network centrality is exploited for identifying important nodes (or edges) of the modeled systems.

Among centrality metrics, Betweenness Centrality (BC) [1] is receiving an increasing interest. Its popularity is due to the potential in identifying critical nodes (or edges) since this metric measures the extent to which a node (or edge) sustains the information flow between any other pair of nodes in the network.

BC is widely used, with both directed and undirected graphs, to identify opinion leaders or influential people in social network analysis [2], critical intersections in transportation networks [3, 4, 5, 6, 7], vulnerabilities in computer networks [8], threats from terrorist networks [9]. However, in spite of the great potential, the computation time of BC often represents a barrier to the application of this metric in large-scale contexts, especially with dynamic graphs.

In recent years, the Floyd method [10], which requires $O(n^3)$ computation time, has been overcome by the well known Brandes' algorithm [11]. Given a graph $G(V, E)$, it exhibits $O(n + m)$ space complexity, $O(nm)$ time complexity for unweighted graphs and $O(nm + n^2 \log(n))$ for weighted ones, where $n = |V|$ is the number of nodes and $m = |E|$ the number of edges. However, the polynomial complexity of Brandes' algorithm, which is almost quadratic for very sparse graphs, is still an obstacle for analyzing very large networks. Such problem becomes even more evident and limiting if centrality is used for real-time analysis of dynamic networks.

In the last decade, many researchers have therefore worked with the aim of improving the performance of Brandes' algorithm. Many proposed solutions exploit the topological characteristics of some kinds of graphs to partition or compress them in order to reduce one of the two dimensions that characterize the complexity of Brandes' algorithm (m or n).

In this paper, we propose an algorithm based on clustering, inspired by previous work on approximated BC computation [12, 6], which makes possible the exact computation of BC on large, undirected graphs with an impressive speedup when compared to Brandes' algorithm and a significant improvement over recent variants of Brandes' algorithm based on clustering [13].

The algorithm leverages structural properties of graphs to find classes of equivalent nodes: by selecting one representative node for each class, we are able to compute BC by significantly reducing the number of single-source shortest path explorations required by Brandes' algorithm. We formally prove the graph properties that we exploit to define and implement two versions of the algorithm based on Scala for

both sequential and parallel map-reduce executions. The experimental analysis has been conducted by testing both versions of the algorithm on synthetic and real-world graphs. The algorithm we propose is able to work with undirected, weighted or unweighted graphs. However, in this paper, we focus on unweighted graphs while its extension to weighted ones can be easily obtained by substituting the breadth-first search (BFS) with Dijkstra exploration.

Undirected graphs are very common in real-world systems; examples are social networks, communication networks, protein interactions graphs, people interaction graphs, finite element meshes, *etc.* Among these graphs, scale-free and Barabasi-Albert graphs [14] represent an important target of our analysis, especially for synthetic graphs, since they model many real-world systems, such as the World Wide Web, the Internet and other computer networks, citation networks, social networks, airline networks, financial networks, *etc.*

The main contributions of the paper are:

- Introduction of the general concept of equivalence class for reducing BC computation time.
- Formal proof about the existence of topological properties to identify an equivalence class with respect to clusters' border nodes in undirected graphs.
- Two variants of Brandes' back propagation technique to avoid the direct computation of the dependency score on cluster nodes due to pivots.
- Scala-based implementations of the proposed algorithm for both sequential and parallel map-reduce executions.
- Extensive evaluation of the proposed algorithm with both synthetic and real large-scale graphs.

The rest of the paper is organized as follows. Section 1 positions the paper with reference to the main results from the literature. Section 2 introduces the notation we use in the rest of the paper, as well as the background concepts and algorithms that are at the basis of the proposed solution. Section 3 formally proves the main properties we exploit to define the algorithm when clustering is used to identify classes of equivalent nodes. Section 4 illustrates the rationale of the specific implementation of the algorithm and the main steps that characterize it, by presenting also the constituent sub-algorithms exploited for the implementation. Section 5 reports on the experimental results obtained by running the proposed algorithm on

several synthetic and real graphs characterized by different sizes and topological properties. Finally, Section 6 summarizes the results and discusses the limits of the proposed solution by highlighting possible future improvements.

1 Related Work

Brandes' algorithm is a fast and robust solution to compute BC, but it is not adequate for real-time processing of large graphs, since BC computation time increases very rapidly with the graph size, even in sparsely-connected configurations.

Several approaches, either exact or approximated, have been developed to reduce BC computation time by improving Brandes' algorithm. The proposed solutions can be classified into five main categories: (a) exploiting and increasing parallelism; (b) updating the BC of specific nodes in dynamically evolving graphs; (c) estimating BC via a partial exploration of the graph in terms of nodes or edges; (d) exploiting structural properties of some kinds of graphs to compress them; (e) decomposing graphs in clusters.

Exploiting parallelism. Brandes' algorithm is extremely parallelizable due to the possibility of performing n independent breadth first searches (BFS) or Dijkstra explorations on a shared graph structure. In [15], an efficient parallel implementation of BC computation is provided. The solution leverages fine-grained multi-level parallelism by concurrently traversing the neighbors of a given node via a shared data structure with granular locking in order to increase concurrency. The improved version of the previous approach, proposed in [16], removes the need for locking in the dependency accumulation stage of Brandes' algorithm through the adoption of a successor list instead of a predecessor list for each node. Other efforts in this direction try to exploit the large amount of cores available on GPUs to better exploit parallelism [17].

Incremental computation. These (stream-based) approaches try to avoid recomputing the BC values of all the nodes of a graph when they are known for a previous configuration, by performing computation over only a small portion of the graph that is impacted by some changes. Recently, an efficient algorithm for streamed BC computation [18] of evolving graphs has been proposed based on edges addition or removal. However, the algorithm is efficient only when the new graph changes in only one edge if compared with the old one. Continuous BC processing of large

graphs to handle streamed changes of a significant number of edges is therefore inefficient.

Approximated computation. These algorithms aim at achieving low computation time by calculating approximated BC values. Brandes and Pich proposed in [19] an approximated algorithm for faster BC calculation by choosing only $k \ll n$ nodes, called pivots, as sources for the single-source shortest path (SSSP) algorithm through different strategies, showing that random selection of pivots can achieve accuracy levels comparable to other heuristics. The approach has been further improved by other authors [20]. The goal of these algorithms is to calculate the BC only for selected nodes called pivot nodes. The selection of these nodes depends on the problem to solve and may limit the use of BC. ABRA [21] is a suite of algorithms to compute high-quality approximations of the betweenness centrality of all nodes (or edges) of both static and fully dynamic graphs by using progressive random sampling.

Topology manipulation. Some algorithms exploit topological properties of graphs to accelerate BC computation. Puzis et al. in [22] propose two heuristics to simplify BC computation: (a) identification of structural equivalent nodes, *i.e.*, nodes that have the same centrality index and contribute equally to the centrality of other nodes; (b) partitioning a large graph in smaller bi-connected sub-graphs. Computation time on the graph partitions is significantly lower due to the quadratic to cubic complexity of Brandes' algorithm. The authors also combine the two techniques to improve the speedup when compared with Brandes' algorithm. In [23], the authors use both compression and splitting techniques, including the ones developed in [22], to reduce the size of the input graph and its largest connected component since these are the main parameters that affects the computation time. In particular, they split the input graph by using bridges and articulation vertices and compress it by removing degree-1, identical and side vertices. Bridges and articulation vertices are edges and nodes, respectively, whose removal from a graph leads to a new graph with a greater number of connected components; degree-1 vertices are leaf nodes which, considered as source and targets, contribute equally to the computation of BC of crossed nodes; identical vertices are the ones characterized by the same neighbors and, consequently, by the same BC values; side vertices are nodes such that the graphs induced by their neighbors are cliques and they are not crossed

by shortest paths. By using all these techniques, the authors achieve significant speedup with different kinds of graphs. The authors in [24] propose a variant of Brandes' algorithm based on topological characteristics of social networks where nodes belonging to particular tree structures are not considered for Brandes' SSSP explorations; their contribution is simply computed by counting. Topology manipulation and graph compression are very useful techniques with some types of graphs and are complementary to other solutions from the literature, including the one proposed in this paper.

Cluster-based computation. A way to compute BC is to cluster a large graph into smaller sub-graphs, calculate the BC inside these small graphs, and then compute the BC on the remaining part of the graph. A first paper based on this approach was proposed in [12]. This technique exploits a fast clustering method [25] to identify clusters inside a graph. The border nodes of the clusters are then used as reference nodes to discover, for each cluster, classes of nodes that contribute the same way to the dependency score of the nodes outside the clusters. For each class, a pivot node is selected as representative node for the computation of the dependency scores from the class nodes to the other graph nodes by exploiting the well-known SSSP exploration of the graph. Hence, the dependency score is multiplied by the cardinality of the class the source node belongs to and summed up to the local contribution of BC, computed by considering only nodes belonging to the clusters, to obtain the final approximated values of BC. This technique can be also classified among the ones based on pivots for computing approximated BC values, even if the strategy adopted to identify pivots is based on clustering. The authors in [13] propose a technique based on clustering to reduce the complexity of BC computation. They prove that with a decomposition of graphs into hierarchical sub networks (HSNs), time complexity can be reduced to $O(n^2)$ for unweighted graphs under the hypothesis that the number of clusters $c \gg k/2$. In that case, the speedup, compared with Brandes' algorithm, is in the order of one half of the graph's average degree k , since the number of edges $m = k \cdot n/2$. This means that if the considered graph has a number of edges $m \sim n$, then $k \sim 2$ and the speedup is 1, that is the algorithm is not able to improve Brandes' algorithm.

A very similar solution has been proposed in [26]. Differently from [13], the authors propose to build a simplified hierarchical representation of the graph after clustering

(named *Skeleton*) by substituting each cluster with a weighted clique connecting the cluster border nodes. This way, they reduce the number of nodes in the Skeleton but need the more computationally expensive Dijkstra algorithm for computing the shortest paths over the weighted graph. Moreover, the proposed solution computes exact BC values of nodes only with respect to a subset of nodes of a graph, named target set. When the target set includes all the nodes of a given graph, the solution converges towards Brandes' algorithm, but with the additional overhead due to the creation and exploitation of the skeleton graph.

In this paper, we propose a cluster-based technique to reduce the time needed for computing exact values of BC in undirected graphs by: *i*) leveraging pivots (as in approximated approaches) to reduce computation time, and *ii*) considering HSN and the properties of undirected graphs to completely remove errors during computation (which affected the algorithm proposed in [12]). This paper is a substantial extension of our first proposal in [27], since the algorithm and its implementations have been significantly improved and tested also with real graphs. Moreover, we formally prove the correctness of the proposed technique.

2 Background

In this section, we first introduce the notation used throughout the paper, then we briefly describe Brandes' algorithm. Finally, we present the concept of *equivalence class*, which constitutes the basis of our algorithm.

Let $\mathbf{G}(\mathbf{V}, \mathbf{E})$ be an undirected unweighted graph with $\mathbf{V}$ representing the set of n vertices (or nodes) and $\mathbf{E}$ the set of m edges (or links). Let $s, t \in \mathbf{V}$ be two generic nodes of $\mathbf{G}$. We denote by $e_{s,t}$ the edge connecting s and t . The *neighbors* of a vertex s are all vertices u such that $e_{s,u} \in \mathbf{E}$. The *distance* between s and t , denoted by $d_{\mathbf{G}}(s, t)$, is the length of the shortest path(s) connecting them in $\mathbf{G}$. The *number* of shortest paths between s and t is denoted by $\sigma_{s,t}$, whereas the number of shortest paths between s and t that cross a generic node $v \in \mathbf{V}$ is denoted by $\sigma_{s,t}(v)$. It is worth noting that since the graph is undirected, $d_{\mathbf{G}}$ and σ are symmetric functions, thus $d_{\mathbf{G}}(s, t) = d_{\mathbf{G}}(t, s)$, $\sigma_{s,t} = \sigma_{t,s}$ and $\sigma_{s,t}(v) = \sigma_{t,s}(v)$. Given a generic node $w \in \mathbf{V}$, $\mathbf{P}_s(w) = \{u \in \mathbf{V} : e_{u,w} \in \mathbf{E}, d_{\mathbf{G}}(s, w) = d_{\mathbf{G}}(s, u) + 1\}$ is the set of direct *predecessors* of vertex w on shortest paths from s .

The *Betweenness Centrality* (BC) of a vertex $v \in \mathbf{V}$ is defined as follows:

$$BC(v) = \sum_{s \neq v \neq t \in \mathbf{V}} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} \quad (1)$$

Notation	Description
$\mathbf{G}$ $\hat{\mathbf{G}}$	undirected unweighted input graph a connected sub-graph of $\mathbf{G}$
$\mathbf{V}$ $\mathbf{V}_{\hat{\mathbf{G}}}$ $\overline{\mathbf{V}}_{\hat{\mathbf{G}}}$ $\mathbf{V}_{HSN}$	set of vertices of $\mathbf{G}$ ($ \mathbf{V} = n$) set of vertices of $\mathbf{G}$ inducing $\hat{\mathbf{G}}$ (set of vertices of $\hat{\mathbf{G}}$) set of vertices in $\mathbf{V} \setminus \mathbf{V}_{\hat{\mathbf{G}}}$ set of vertices of HSN
$\mathbf{E}$ $e_{s,t}$	set of edges of $\mathbf{G}$ ($ \mathbf{E} = m$) edge connecting vertices s and t
$d_{\mathbf{G}}(s, t)$ $\hat{d}_{\mathbf{G}}(s, t)$	distance between vertices s and t in $\mathbf{G}$ normalized distance between vertices s and t in $\mathbf{G}$
$\sigma_{s,t}$ $\sigma_{s,t}(v)$ $\hat{\sigma}_{s,t}$	number of shortest paths between vertices s and t number of shortest paths between vertices s and t which cross vertex v normalized number of shortest paths between vertices s and t
$\mathbf{P}_s(v)$ $\mathbf{P}_s(\mathbf{V})$	set of direct predecessors of vertex v on shortest paths from vertex s set of direct predecessors of vertices in $\mathbf{V}$ on shortest paths from vertex s
$BC(v)$ $\delta_{s,t}(v)$ $\delta_{s,\bullet}(v)$ $\delta_{s,\mathbf{V}_{\hat{\mathbf{G}}}}(v)$	betweenness centrality of vertex v pair-dependency of pair of vertices (s, t) on the intermediary vertex v dependency score of vertex s on vertex v due to all destination vertices dependency score of vertex s on vertex v due to all destination vertices in $\mathbf{V}_{\hat{\mathbf{G}}}$
$\mathbf{C}$ $\mathbf{C}_i$ $\mathbf{C}(v)$ $\mathbf{C}^*$ $\mathbf{C}_i^*$	set of clusters of $\mathbf{G}$ a generic cluster in $\mathbf{C}$ the cluster vertex v belongs to set of extended clusters in $\mathbf{G}$ a generic extended cluster in $\mathbf{C}^*$
$\mathbf{K}$ $\mathbf{K}_i$ $\mathbf{K}_{\mathbf{C}_i}$	set of all the equivalence classes an equivalence class set of equivalence classes of cluster $\mathbf{C}_i$
$\mathbf{P}$ k_i	set of all the pivots pivot node of the equivalence class $\mathbf{K}_i$
$\mathbf{EN}$ $\mathbf{EN}_{\mathbf{C}_i}$	set of all the external nodes set of external nodes of cluster $\mathbf{C}_i$
$\mathbf{BN}$ $\mathbf{BN}_{\mathbf{C}_i}$ $\mathbf{BN}_{\mathbf{C}_i}(s, t)$ b_i	set of all the border nodes set of border nodes of cluster $\mathbf{C}_i$ set of border nodes of cluster $\mathbf{C}_i$ on shortest paths from $s \in \mathbf{V}_{\mathbf{C}_i}$ to $t \in \overline{\mathbf{V}_{\mathbf{C}_i}}$ a generic border node in $\mathbf{BN}$
$\delta_{s,\bullet}^\gamma(v)$ $\delta_{s,\mathbf{V}_{\mathbf{C}(v)}}^\gamma(v)$ $\delta_{s,\mathbf{V}_{\mathbf{C}(v)}}^\gamma(\mathbf{V}_{\mathbf{C}(v)})$ $\delta^\gamma(v)$ $\delta^\gamma(\mathbf{V})$	global dependency score of s on v due to all $t \in \overline{\mathbf{V}_{\mathbf{C}(s)}}$ (same as $\delta_{s,\overline{\mathbf{V}_{\mathbf{C}(s)}}}^\gamma(v)$) global dependency score of s on v due to all $t \in (\overline{\mathbf{V}_{\mathbf{C}(s)}} \cap \mathbf{V}_{\mathbf{C}(v)})$ global dependency score of s on vertices in $\mathbf{V}_{\mathbf{C}(v)}$ due to all $t \in (\overline{\mathbf{V}_{\mathbf{C}(s)}} \cap \mathbf{V}_{\mathbf{C}(v)})$ sum of all the global dependency scores (global BC) on v sum of all the global dependency scores (global BC) on vertices in $\mathbf{V}$
$\delta_{s,\bullet}^\lambda(v)$ $\delta_{s,\bullet}^\lambda(\mathbf{V})$ $\delta_{s,\bullet}^\lambda(v)$ $\delta^\lambda(\mathbf{V})$	local dependency score of s on v due to all $t \in \mathbf{V}_{\mathbf{C}(s)} = \mathbf{V}_{\mathbf{C}(v)}$ local dependency score of s on vertices in $\mathbf{V}$ due to all $t \in \mathbf{V}_{\mathbf{C}(s)} = \mathbf{V}_{\mathbf{C}(v)}$ sum of all the local dependency scores (local BC) on v sum of all the local dependency scores (local BC) on vertices in $\mathbf{V}$
$\delta_{s,\bullet}^\epsilon(v)$ $\delta_{s,\bullet}^\epsilon(\mathbf{EN})$ $\delta^\epsilon(v)$ $\delta^\epsilon(\mathbf{EN}_{\mathbf{C}(s)})$	dependency score of s on v , as external node, due to all $t \in \mathbf{V}_{\mathbf{C}(s)}$ dependency score of s on external nodes $\mathbf{EN}$ due to all $t \in \mathbf{V}_{\mathbf{C}(s)}$ sum of all the dependency scores on v as external node sum of all the dependency scores on external nodes of cluster $\mathbf{C}(s)$

Table 1: Notation.

$BC(v)$ thus represents the fraction of shortest paths containing v among all the shortest paths in the graph between any generic pair of nodes s and t , summed over all possible pairs s and t with $s \neq v$, $s \neq t$ and $v \neq t$.

We refer to Table 1 for a summary of the notation used in the paper.

2.1 Brandes' algorithm

Brandes' algorithm is the fastest known general-purpose sequential algorithm for computing BC. It is based on the notions of *pair-dependency* and *dependency score*. Let us consider two generic nodes $s, t \in \mathbf{V}$. Given shortest paths counts $\sigma_{s,t}(v)$ and $\sigma_{s,t}$, the pair-dependency $\delta_{s,t}(v)$ of a pair s, t on an intermediary node $v \in \mathbf{V}$ is defined as follows:

$$\delta_{s,t}(v) = \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} \quad (2)$$

The pair-dependency represents the fraction of shortest paths between s and t crossing v . The dependency score $\delta_{s,\bullet}(v)$ of a vertex s on a vertex $v \in \mathbf{V}$ is then defined as follows:

$$\delta_{s,\bullet}(v) = \sum_{t \in \mathbf{V}} \delta_{s,t}(v) \quad (3)$$

BC can thus be redefined in terms of dependency score:

$$BC(v) = \sum_{s \neq v \neq t \in \mathbf{V}} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} = \sum_{s \neq v \neq t \in \mathbf{V}} \delta_{s,t}(v) = \sum_{s \in \mathbf{V}} \delta_{s,\bullet}(v) \quad (4)$$

The key observation of Brandes' algorithm is that the dependency score obeys a recursive formula. In particular, for each $s \in \mathbf{V}$ we have:

$$\delta_{s,\bullet}(v) = \sum_{w: v \in \mathbf{P}_s(w)} \frac{\sigma_{s,v}}{\sigma_{s,w}} \cdot (1 + \delta_{s,\bullet}(w)) \quad (5)$$

Brandes' algorithm runs in two phases, exploiting equation 5. For each (source) node $s \in \mathbf{V}$, in the first phase, a single-source shortest-paths (SSSP) algorithm, based on breadth-first search (BFS), is executed on $\mathbf{G}$ to find all the shortest paths rooted in s . In the second phase, dependency scores are accumulated by backtrack-

ing along the discovered shortest paths using the recursive relation in Eq. 5. In backtracking, nodes are visited in descending order of distance from the source. During these two phases, for each node $v \in \mathbf{V}$ the algorithm builds and exploits the following data structures: the set of direct predecessors $\mathbf{P}_s(v)$ on shortest paths from the source, the distance $d_{\mathbf{G}}(s, v)$ from the source, the number of shortest paths $\sigma_{s,v}$ from the source and the dependency score $\delta_{s,\bullet}(v)$ that accumulates the contribution of the source on node v due to all destinations during the back-propagation step.

2.2 Equivalence class

Computing BC over the whole graph through n SSSP explorations as in Brandes' algorithm still requires an extremely high computation time for large graphs. To reduce the number of explorations and thus lower the computation time, we exploit the concept of *equivalence class*. Formally, given a connected sub-graph $\hat{\mathbf{G}}$ of $\mathbf{G}$ induced by the set of nodes $\mathbf{V}_{\hat{\mathbf{G}}} \subset \mathbf{V}$, we define an equivalence class $\mathbf{K}_i$ as any subset of nodes in $\mathbf{V}_{\hat{\mathbf{G}}}$ that produce the same dependency score on all nodes - and for destinations - outside sub-graph $\hat{\mathbf{G}}$ when used as sources for SSSP explorations.

By choosing only one representative node (called pivot) for each class, the correct dependency scores of nodes can be computed by multiplying the scores computed via the SSSP rooted in the pivot by the cardinality of the class, *i.e.*, let k_i be a pivot of $\mathbf{K}_i$ and $v \notin \mathbf{V}_{\hat{\mathbf{G}}}$, a node outside sub-graph $\hat{\mathbf{G}}$, we have:

$$\sum_{s \in \mathbf{K}_i} \sum_{t \notin \mathbf{V}_{\hat{\mathbf{G}}}} \delta_{s,t}(v) = |\mathbf{K}_i| \cdot \sum_{t \notin \mathbf{V}_{\hat{\mathbf{G}}}} \delta_{k_i,t}(v)$$

which, according to our notation, can be re-written as:

$$\sum_{s \in \mathbf{K}_i} \delta_{s, \overline{\mathbf{V}_{\hat{\mathbf{G}}}}}(v) = |\mathbf{K}_i| \cdot \delta_{k_i, \overline{\mathbf{V}_{\hat{\mathbf{G}}}}}(v) \quad (6)$$

Equation 6 clearly shows that a low number of classes significantly reduces the computation time, by allowing to skip a high number of SSSP explorations.

A possible technique to identify equivalence classes is to consider reference nodes. Given a generic sub-graph $\hat{\mathbf{G}}$, the reference nodes in $\mathbf{V}_{\hat{\mathbf{G}}}$ are those that need to be traversed to reach, via shortest paths from nodes in $\mathbf{V}_{\hat{\mathbf{G}}}$, any other node in $\overline{\mathbf{V}_{\hat{\mathbf{G}}}}$.

To easily identify reference nodes, we use clustering while to increase the chances of identifying a low number of equivalence classes, we consider a clustering technique based on modularity. Such an approach allows reducing the amount of connections among groups of nodes belonging to different clusters, and, consequently, lowers the number of reference nodes to be considered for discovering equivalence classes.

The approach and the details about the clustering technique adopted are reported in the next sections.

3 Clustering and BC computation

In this section, we discuss and formally prove the graph properties exploited to identify equivalence classes within clusters.

Let us assume that a given graph $\mathbf{G}$ is split into a set of clusters $\mathbf{C}$, where a single cluster $\mathbf{C}_i$ is a connected sub-graph of $\mathbf{G}$ induced by a set of nodes $\mathbf{V}_{\mathbf{C}_i} \subset \mathbf{V}$. For each cluster $\mathbf{C}_i \in \mathbf{C}$, it is possible to identify a set of *border nodes* $\mathbf{BN}_{\mathbf{C}_i}$. A border node $b_i \in \mathbf{BN}_{\mathbf{C}_i}$ is a node belonging to $\mathbf{C}_i$ and having at least one neighbor belonging to another cluster, as graphically presented in Fig. 1 (circled nodes are border nodes).

3.1 Equivalence class with clustering: formal proofs

To discover equivalence classes, for each cluster $\mathbf{C}_i$, we group nodes based on their distance and number of shortest paths to the border nodes identified via clustering.

Theorem 3.1 proves that, for any given pair of nodes s, p belonging to the sub-graph induced by nodes in cluster $\mathbf{C}_i$ (*i.e.*, $s, p \in \mathbf{V}_{\mathbf{C}_i}$), the dependency score due to s and p on all nodes $v \in \overline{\mathbf{V}_{\mathbf{C}_i}}$ and for destinations $t \in \overline{\mathbf{V}_{\mathbf{C}_i}}$ is the same if the distance and the number of shortest paths from s and p to every border node of $\mathbf{C}_i$ are the same except for an additive or multiplicative factor, respectively.

Theorem 3.1 *Let $k \in \mathbb{R}^+$ and $l \in \mathbb{R}$. Let $\mathbf{C}_i$ be a generic cluster of graph $\mathbf{G}$ with border nodes $\mathbf{BN}_{\mathbf{C}_i}$. Let $s, p \in \mathbf{V}_{\mathbf{C}_i}$. If $\forall b_j \in \mathbf{BN}_{\mathbf{C}_i} \sigma_{s,b_j} = k \cdot \sigma_{p,b_j}$ and $d_{\mathbf{G}}(s, b_j) = d_{\mathbf{G}}(p, b_j) + l$, then $\delta_{s, \overline{\mathbf{V}_{\mathbf{C}_i}}}(v) = \delta_{p, \overline{\mathbf{V}_{\mathbf{C}_i}}}(v), \forall v \in \overline{\mathbf{V}_{\mathbf{C}_i}}$.*

Proof By rewriting the statement of the theorem as follows:

$$\delta_{s, \overline{\mathbf{V}_{\mathbf{C}_i}}}(v) = \delta_{p, \overline{\mathbf{V}_{\mathbf{C}_i}}}(v) \quad \forall v \in \overline{\mathbf{V}_{\mathbf{C}_i}} \iff \sum_{t \notin \mathbf{V}_{\mathbf{C}_i}} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} = \sum_{t \notin \mathbf{V}_{\mathbf{C}_i}} \frac{\sigma_{p,t}(v)}{\sigma_{p,t}} \quad \forall v \in \overline{\mathbf{V}_{\mathbf{C}_i}} \quad (7)$$

we can prove it by proving that the two following conditions:

$$\sigma_{s,t}(v) = k \cdot \sigma_{p,t}(v) \quad \forall v, t \in \overline{\mathbf{V}_{C_i}} \quad (8)$$

and

$$\sigma_{s,t} = k \cdot \sigma_{p,t} \quad \forall t \in \overline{\mathbf{V}_{C_i}} \quad (9)$$

hold under the hypotheses of the theorem.

Let us first prove the following Lemma 3.2, which permits to express the relationship on the distances to cluster border nodes (*i.e.*, $d_{\mathbf{G}}(s, b_j) = d_{\mathbf{G}}(p, b_j) + l$) as

an equivalence of the sets of border nodes traversed from s and p to reach nodes t outside the given cluster.

Lemma 3.2 *Let $\mathbf{BN}_{\mathbf{C}_i}(u, t) \subseteq \mathbf{BN}_{\mathbf{C}_i}$ denote the set of border nodes of cluster $\mathbf{C}_i$ on the shortest paths from $u \in \mathbf{V}_{\mathbf{C}_i}$ to $t \in \overline{\mathbf{V}_{\mathbf{C}_i}}$. Given a constant $l \in \mathbb{R}$ and two nodes $s, p \in \mathbf{V}_{\mathbf{C}_i}$, if $d_{\mathbf{G}}(s, b_j) = d_{\mathbf{G}}(p, b_j) + l \quad \forall b_j \in \mathbf{BN}_{\mathbf{C}_i}$ then $\mathbf{BN}_{\mathbf{C}_i}(s, t) = \mathbf{BN}_{\mathbf{C}_i}(p, t)$.*

Proof Let us consider two border nodes $b_j, b_k \in \mathbf{BN}_{\mathbf{C}_i}$ with $b_j \in \mathbf{BN}_{\mathbf{C}_i}(s, t)$ and $b_k \notin \mathbf{BN}_{\mathbf{C}_i}(s, t)$. By definition of shortest path between two nodes, we have:

$$d_{\mathbf{G}}(s, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(s, t) \quad (10)$$

Given that $b_j \in \mathbf{BN}_{\mathbf{C}_i}(s, t)$ by hypothesis, Eq. 10 can be easily re-written as follows:

$$\begin{aligned} d_{\mathbf{G}}(s, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(s, t) &\iff d_{\mathbf{G}}(s, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(s, b_j) \\ &\quad + d_{\mathbf{G}}(b_j, t) \end{aligned} \quad (11)$$

Now, by relying on the hypothesis of the lemma, we exploit the relationships holding between the distances of generic nodes s and p to each border node in $\mathbf{BN}_{\mathbf{C}_i}$, thus obtaining:

$$\begin{aligned} d_{\mathbf{G}}(s, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(s, t) &\iff d_{\mathbf{G}}(p, b_k) + l + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(p, b_j) \\ &\quad + l + d_{\mathbf{G}}(b_j, t) \\ &\iff d_{\mathbf{G}}(p, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(p, b_j) \\ &\quad + d_{\mathbf{G}}(b_j, t) \end{aligned} \quad (12)$$

From Eq. 12, we can derive that b_k does not belong to any shortest path between p and t , i.e.:

$$d_{\mathbf{G}}(s, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(s, t) \iff b_k \notin \mathbf{BN}_{\mathbf{C}_i}(p, t)$$

As the relation above holds for any node $b_k \in \mathbf{BN}_{\mathbf{C}_i}$ which does not belong to any shortest path between s and t , we can conclude that:

$$\mathbf{BN}_{\mathbf{C}_i}(p, t) \subseteq \mathbf{BN}_{\mathbf{C}_i}(s, t) \quad (13)$$

Likewise, it is possible to prove that if $b_j \in \mathbf{BN}_{\mathbf{C}_i}(p, t)$ and $b_k \notin \mathbf{BN}_{\mathbf{C}_i}(p, t)$, we have:

$$d_{\mathbf{G}}(p, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(p, t) \iff \mathbf{BN}_{\mathbf{C}_i}(s, t) \subseteq \mathbf{BN}_{\mathbf{C}_i}(p, t) \quad (14)$$

Therefore, from Eq. 13 and Eq. 14, we can conclude that the following relationship holds:

$$\begin{aligned} \mathbf{BN}_{\mathbf{C}_i}(s, t) \subseteq \mathbf{BN}_{\mathbf{C}_i}(p, t) \text{ AND } \mathbf{BN}_{\mathbf{C}_i}(p, t) \subseteq \mathbf{BN}_{\mathbf{C}_i}(s, t) \\ \iff \\ \mathbf{BN}_{\mathbf{C}_i}(s, t) = \mathbf{BN}_{\mathbf{C}_i}(p, t) \end{aligned} \quad (15)$$

which proves the lemma. $\square$

To complete the proof of Theorem 3.1, we need now to prove Eq. 8 and Eq. 9. To that purpose, we consider the following lemma.

Lemma 3.3 *Let s be a node of cluster $\mathbf{C}_i$, and t any node in $\overline{\mathbf{V}_{\mathbf{C}_i}}$. $\mathbf{BN}_{\mathbf{C}_i}(s, t)$ is the set of border nodes of cluster $\mathbf{C}_i$ that belong to the shortest paths between s and t . If $\mathbf{BN}_{\mathbf{C}_i}(s, t) = \mathbf{BN}_{\mathbf{C}_i}(p, t)$, then, $\forall t \in \overline{\mathbf{V}_{\mathbf{C}_i}}$, $\sigma_{s,t} = k \cdot \sigma_{p,t}$ and $\sigma_{s,t}(v) = k \cdot \sigma_{p,t}(v)$.*

Proof By leveraging Bellman's criterion:

$$\sigma_{s,t} = \sum_{b_j \in \mathbf{BN}_{\mathbf{C}_i}(s,t)} \sigma_{s,b_j} \cdot \sigma_{b_j,t}. \quad (16)$$

From the hypothesis of Theorem 3.1, we know that $\sigma_{s,b_j} = k \cdot \sigma_{p,b_j} \forall b_j \in \mathbf{BN}_{\mathbf{C}_i}$ and equivalently $\forall b_j \in \mathbf{BN}(s, t)$, as $\mathbf{BN}(s, t) \subseteq \mathbf{BN}_{\mathbf{C}_i}$. Therefore, Eq. 16 becomes:

$$\sigma_{s,t} = \sum_{b_j \in \mathbf{BN}_{\mathbf{C}_i}(s,t)} k \cdot \sigma_{p,b_j} \cdot \sigma_{b_j,t} \quad (17)$$

By the hypotheses of this lemma, we also know that $\mathbf{BN}_{\mathbf{C}_i}(s,t) = \mathbf{BN}_{\mathbf{C}_i}(p,t)$.

Thus, we have:

$$\begin{aligned} \sum_{b_j \in \mathbf{BN}_{\mathbf{C}_i}(s,t)} k \cdot \sigma_{p,b_j} \cdot \sigma_{b_j,t} &= k \cdot \sum_{b_j \in \mathbf{BN}_{\mathbf{C}_i}(p,t)} \sigma_{p,b_j} \cdot \sigma_{b_j,t} \\ &= k \cdot \sigma_{p,t} \end{aligned} \quad (18)$$

With the same reasoning, it is also evident to prove the following:

$$\sigma_{s,t}(v) = k \cdot \sigma_{p,t}(v) \quad (19)$$

□

Eq. 19 and Eq. 18 from Lemma 3.3 prove, via Lemma 3.2, Eq. 8 and Eq. 9 respectively. Therefore Theorem 3.1 is proved.

□

In the following, we use normalized distance, noted as $\hat{d}_{\mathbf{G}}(s, b_j)$, and normalized number of shortest paths, noted as $\hat{\sigma}_{s,b_j}$, to identify equivalence classes. Specifically, normalized distance is defined as follows:

$$\hat{d}_{\mathbf{G}}(s, b_j) = d_{\mathbf{G}}(s, b_j) - \min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} d_{\mathbf{G}}(s, b_k)$$

while the normalized number of shortest paths is defined as:

$$\hat{\sigma}_{s,b_j} = \sigma_{s,b_j} / \min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} \sigma_{s,b_k}$$

Given the definitions above, we can prove the following corollary.

Corollary 3.4 *Let $k \in \mathbb{R}^+$ and $l \in \mathbb{R}$. Let $\mathbf{C}_i$ be a cluster of graph $\mathbf{G}$ with border nodes $\mathbf{BN}_{\mathbf{C}_i}$ and $s, p \in \mathbf{V}_{\mathbf{C}_i}$. If, $\forall b_j \in \mathbf{BN}_{\mathbf{C}_i}$, $\hat{\sigma}_{s,b_j} = \hat{\sigma}_{p,b_j}$ and $\hat{d}_{\mathbf{G}}(s, b_j) = \hat{d}_{\mathbf{G}}(p, b_j)$, then $\delta_{s, \overline{\mathbf{V}_{\mathbf{C}_i}}}(v) = \delta_{p, \overline{\mathbf{V}_{\mathbf{C}_i}}}(v)$, for any node $v \in \overline{\mathbf{V}_{\mathbf{C}_i}}$.*

Proof To prove the corollary, we only need to prove that the following two equations hold:

$$\hat{d}_{\mathbf{G}}(s, b_j) = \hat{d}_{\mathbf{G}}(p, b_j) \quad \forall b_j \in \mathbf{BN}_{\mathbf{C}_i} \implies d_{\mathbf{G}}(s, b_j) = d_{\mathbf{G}}(p, b_j) + l \quad \forall b_j \in \mathbf{BN}_{\mathbf{C}_i} \quad (20)$$

and:

$$\hat{\sigma}_{s,b_j} = \hat{\sigma}_{p,b_j} \quad \forall b_j \in \mathbf{BN}_{\mathbf{C}_i} \implies \sigma_{s,b_j} = k \cdot \sigma_{p,b_j} \quad \forall b_j \in \mathbf{BN}_{\mathbf{C}_i} \quad (21)$$

Let us consider any two generic pair of nodes s and p belonging to cluster $\mathbf{C}_i$ such that:

$$\begin{aligned} & \forall b_j \in \mathbf{BN}_{\mathbf{C}_i}, \\ & d_{\mathbf{G}}(s, b_j) - \min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} d_{\mathbf{G}}(s, b_k) = d_{\mathbf{G}}(p, b_j) - \min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} d_{\mathbf{G}}(p, b_k) \quad (22) \\ & \text{AND} \\ & \frac{\sigma_{s,b_j}}{\min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} \sigma_{s,b_k}} = \frac{\sigma_{p,b_j}}{\min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} \sigma_{p,b_k}} \quad (23) \end{aligned}$$

By definition, Eq. 22 can be easily re-written as follows:

$$\begin{aligned} & \forall b_j \in \mathbf{BN}_{\mathbf{C}_i}, \\ & d_{\mathbf{G}}(s, b_j) - \min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} d_{\mathbf{G}}(s, b_k) = d_{\mathbf{G}}(p, b_j) - \min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} d_{\mathbf{G}}(p, b_k) \\ & \iff d_{\mathbf{G}}(s, b_j) = d_{\mathbf{G}}(p, b_j) + \underbrace{\min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} d_{\mathbf{G}}(p, b_k) - \min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} d_{\mathbf{G}}(s, b_k)}_{\text{constant value}} \\ & \iff d_{\mathbf{G}}(s, b_j) = d_{\mathbf{G}}(p, b_j) + l \quad \text{with } l \in \mathbb{R} \end{aligned}$$

which corresponds to Eq. 20.

Likewise, Eq. 23 can be re-written as:

$$\begin{aligned}
 & \forall b_j \in \mathbf{BN}_{\mathbf{C}_i}, \\
 & \frac{\sigma_{s,b_j}}{\min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} \sigma_{s,b_k}} = \frac{\sigma_{p,b_j}}{\min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} \sigma_{p,b_k}} \\
 & \iff \sigma_{s,b_j} = \sigma_{p,b_j} \cdot \underbrace{\frac{\min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} \sigma_{s,b_k}}{\min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} \sigma_{p,b_k}}}_{\text{constant ratio}} \\
 & \iff \sigma_{s,b_j} = \sigma_{p,b_j} \cdot k \quad \text{with } k \in \mathbb{R}^+
 \end{aligned}$$

which corresponds to Eq. 21. As the two equations (Eq. 20 and Eq. 21) are jointly satisfied, the corollary is proved from Theorem 3.1. $\square$

In conclusion, the topological properties considered in our approach to group nodes into equivalence classes are the *normalized number of shortest paths* and the *normalized distances* from nodes in $\mathbf{V}_{\mathbf{G}}$ to the reference nodes identified through clustering, i.e. *border nodes*.

For the sake of clarity, we propose in the following an example illustrating the concepts of normalized number of shortest paths and normalized distance, as well as the technique exploited to group equivalent nodes. Let $\mathbf{G}$ be the simple graph reported in Fig. 1a, decomposed in three clusters, each separately shown in Fig. 1b. We focus on the blue cluster, referred as $\mathbf{C}_1$, in order to illustrate the concept of equivalence class (see Table 2 and Fig. 2). In $\mathbf{C}_1$, nodes **1** and **2** are border nodes (*i.e.*, b_1 and b_2 in Table 2), while the remaining nodes of $\mathbf{C}_1$ are related to b_1 and b_2 according to the properties detailed in Table 2: for each node the normalized distances and normalized number of shortest paths to the border nodes are reported. According to our previous definitions, nodes **3**, **4**, **6**, **14** and **5**, **1** can be grouped in two classes respectively, whereas node **2** is assigned to a singleton class. Nodes **1**, **2** and **14** are the pivots^[1].

3.2 Cluster-based exact BC computation

The equivalence classes allow us to compute the dependency score on nodes - and for destinations - that do not belong to the same cluster of the source node, which

^[1]In our previous version of the algorithm, the pivots were chosen to minimize the error. Here, as we will explain later, any node in a class can be a pivot.

node v	$\hat{d}_{C_1}(v, b_1)$	$\hat{d}_{C_1}(v, b_2)$	$\hat{\sigma}_{vb_1}$	$\hat{\sigma}_{vb_2}$
1	0	2	1	2
2	2	0	2	1
3	0	0	1	1
4	0	0	1	1
5	0	2	1	2
6	0	0	1	1
14	0	0	1	1

Table 2: Normalized distances and normalized number of shortest paths for the blue cluster C_1

Figure 2: Classes of equivalent nodes in the blue cluster C_1

means that the contributions computed via this approach are only *partial*. To obtain the total BC, we rewrite Eq. 4 as follows:

$$\begin{aligned}
 BC(v) &= \sum_{s \in \mathbf{V}} \delta_{s, \bullet}(v) \\
 &= \sum_{s \in \mathbf{V}} \sum_{t \in \mathbf{V}_{C(s)}} \delta_{s,t}(v) + \sum_{s \in \mathbf{V}} \sum_{t \notin \mathbf{V}_{C(s)}} \delta_{s,t}(v) \\
 &= \underbrace{\sum_{s \in \mathbf{V}_{C(v)}} \sum_{t \in \mathbf{V}_{C(v)}} \delta_{s,t}(v)}_{\text{sum of local dependency scores} = \delta^\lambda(v)} + \underbrace{\sum_{s \in \mathbf{V}} \sum_{t \notin \mathbf{V}_{C(s)}} \delta_{s,t}(v)}_{\text{sum of global dependency scores} = \delta^\gamma(v)} \\
 &\quad + \underbrace{\sum_{s \notin \mathbf{V}_{C(v)}} \sum_{t \in \mathbf{V}_{C(s)}} \delta_{s,t}(v)}_{\text{sum of dependency scores on external nodes} = \delta^\epsilon(v)}
 \end{aligned} \tag{24}$$

As a result, we can distinguish two main terms, *local* and *global* dependency scores. The additional term is necessary to properly take into account the possible

existence of shortest paths connecting nodes of the same cluster via nodes belonging to one or more different clusters, *i.e.*, *external nodes*.

We define the *local* dependency score of a node s on a node v , $\delta_{s,\bullet}^\lambda(v)$, as the sum of pair dependency scores for which source s , the destinations and node v belong all to the same cluster. We define the *local BC* of a node v , $\delta^\lambda(v)$, as the BC of v computed on the sub-graph $\mathbf{C}(v)$. Local BC is computed using Brandes' algorithm inside each cluster^[2], which generates, as a by-product, additional information (*i.e.*, the number of shortest paths and distances to border nodes). This information is later used to group nodes into equivalence classes and to fasten the computation of global dependency scores, as further discussed (see Subsec. 4.2).

The *global* dependency score of a node s on a node v , $\delta_{s,\bullet}^\gamma(v)$, is the sum of all the pair dependency scores for which destinations do not belong to the same cluster of source node s . The *global BC* of the generic node v , $\delta^\gamma(v)$, is thus the sum of the global dependency scores for source node s ranging over the whole set of nodes $\mathbf{V}$.

The dependency score of a node s on an external node v , *i.e.* $\mathbf{C}(v) \neq \mathbf{C}(s)$, noted as $\delta_{s,\bullet}^\epsilon(v)$, is the sum of all the pair dependency scores for which destinations belong to the same cluster of the source node s . We denote by $\delta^\epsilon(v)$ the sum of all the dependency scores on v , when v is an external node and the sources and destinations are in the same cluster, different from $\mathbf{C}(v)$.

This last term $\delta^\epsilon(v)$ is equal to zero when the clustering is ideal, *i.e.* when all the shortest paths between any pair of nodes of a cluster only contain nodes from that same cluster. When this condition is not fulfilled, multiple side effects due to the presence of external nodes have to be taken into account, as discussed below.

3.2.1 External nodes/shortest paths

Given a cluster $\mathbf{C}_i$, two nodes $s, t \in \mathbf{C}_i$ and two border nodes $b_1, b_2 \in \mathbf{C}_i$, there may exist shortest paths between s and t which exit $\mathbf{C}_i$ through b_1 , cross a certain number of nodes belonging to other clusters and then re-enter $\mathbf{C}_i$ through b_2 . We call these shortest paths *external* shortest paths and the nodes lying on them which do not belong to $\mathbf{C}_i$, $\mathbf{EN}_{\mathbf{C}_i}$, *external* nodes of $\mathbf{C}_i$. If the existence of such external shortest paths is neglected, BC computation will be affected by an error due to incorrect values of the distances and the counts of shortest paths between pairs of

^[2]As explained later, in the special case where there are external shortest paths in the cluster, the local BC is actually computed inside the *extended* cluster.

nodes inside the same cluster. Consequently, an error in the computation of the local BC, δ^λ , and in the identification of equivalence classes will be introduced. This was one of the approximation errors affected the previous version of our algorithm [12]. To remove this intra-cluster error, we join the idea proposed by the authors in [13]. After clustering, we build a Hierarchical Sub-Network (HSN), *i.e.*, a sub-graph of $\mathbf{G}$ induced by the border nodes of all the clusters and nodes lying on the intra-cluster shortest paths between pairs of border nodes of the same cluster.

By retrieving all the shortest paths between pairs of border nodes of the same cluster via the HSN, we are able to identify possible external nodes for that cluster. Afterwards, we can extend each cluster with the related external nodes and use the extended clusters as sub-graphs to identify equivalence classes and pivots. Thus, local BC δ^λ can be correctly computed inside these extended clusters instead of the initial ones. Formally, an extended cluster $\mathbf{C}_i^*$ of a cluster $\mathbf{C}_i \in \mathbf{C}$ is defined as a connected sub-graph induced by nodes $\mathbf{V}_{\mathbf{C}_i^*} = \mathbf{V}_{\mathbf{C}_i} \cup \mathbf{EN}_{\mathbf{C}_i}$.

To better understand how the HSN is built and how it is used to form the extended clusters, we provide an illustrative example. Let us consider again the clustered graph from Fig. 1. In cluster $\mathbf{C}_1$, nodes 1 and 2 are border nodes, while node 4 lies on the only intra-cluster shortest path between them. In cluster $\mathbf{C}_2$, nodes 17 and 20 are border nodes and nodes 15, 21, 19 and 16 lie on the intra-cluster shortest paths between them. Finally, in cluster $\mathbf{C}_3$, there is only border node 8. All the aforementioned nodes build up the HSN (see Fig. 3a). If we now consider the shortest paths between border nodes 1 and 2 via the HSN, we notice that node 17

lies on a shortest path connecting the two former nodes. Consequently, it represents an external node of $\mathbf{C}_1$ (see Fig. 3b).

3.2.2 Dependency score of pivots

From the equivalence class relationship described in Subsec. 2.2, a pivot of such a class is representative only for the dependency scores on nodes v - and destinations t - which do not belong to its own cluster. In fact, given a cluster $\mathbf{C}_i \in \mathbf{C}$ and all its equivalence classes $\mathbf{K}_{\mathbf{C}_i}$, from Eq. 6, we have :

$$\sum_{s \in \mathbf{K}_i} \delta_{s, \overline{\mathbf{V}_{\mathbf{C}_i}}}(v) = |\mathbf{K}_i| \cdot \delta_{k_i, \overline{\mathbf{V}_{\mathbf{C}_i}}}(v) \quad \forall v \in \overline{\mathbf{V}_{\mathbf{C}_i}}, \mathbf{K}_i \in \mathbf{K}_{\mathbf{C}_i}. \quad (25)$$

This equation can be exploited to speed up computation of BC building on Brandes' algorithm and SSSP explorations, but only holds if $v \in \overline{\mathbf{V}_{\mathbf{C}_i}}$. Thus, it cannot be directly applied to correctly compute values of global BC when v is in the same cluster of the source. Therefore, the algorithm requires a more elaborated approach to properly and efficiently calculate the contribution from the pivot of $\mathbf{K}_{\mathbf{C}_i}$ to the BC of nodes $v \in \mathbf{V}_{\mathbf{C}_i}$. In our previous version of the algorithm, we used Eq. 25 without taking into account the cluster of v . The pivot was chosen to minimize the error. We can avoid such error during the computation of global dependency scores by exploiting the properties of undirected graphs. In the following, we explain in details this technique.

First of all, let us decompose the global dependency scores from Eq. 24 based on the cluster of node v as follows:

$$\delta^\gamma(v) = \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \notin (\mathbf{V}_{\mathbf{C}(v)} \cup \mathbf{V}_{\mathbf{C}(s)})} \delta_{s,t}(v) + \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \in \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) + \sum_{s \in \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \notin \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) \quad (26)$$

The previous equation can be further simplified by considering the following claim.

Claim 1 In undirected graphs:

$$\sum_{s \in \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \notin \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) = \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \in \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) \quad (27)$$

Proof Thanks to the undirected nature of the graph, we have:

$$\begin{aligned}
 \sum_{s \in \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \notin \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) &= \sum_{s \in \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \notin \mathbf{V}_{\mathbf{C}(v)}} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} \\
 &= \sum_{s \in \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \notin \mathbf{V}_{\mathbf{C}(v)}} \frac{\sigma_{t,s}(v)}{\sigma_{t,s}} \\
 &= \sum_{t \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{s \in \mathbf{V}_{\mathbf{C}(v)}} \frac{\sigma_{t,s}(v)}{\sigma_{t,s}}
 \end{aligned}$$

Now, by changing the name of variables s and t :

$$\begin{aligned}
 \sum_{s \in \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \notin \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) &= \sum_{t \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{s \in \mathbf{V}_{\mathbf{C}(v)}} \frac{\sigma_{t,s}(v)}{\sigma_{t,s}} \\
 &= \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \in \mathbf{V}_{\mathbf{C}(v)}} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} \\
 &= \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \in \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v)
 \end{aligned}$$

□

By relying on Claim 1, it becomes possible to replace with zero the sum of the pair-dependencies $\delta_{s,t}(v)$ for which $s \in \mathbf{V}_{\mathbf{C}(v)}$ and $t \in \overline{\mathbf{V}_{\mathbf{C}(v)}}$ in Eq. 26 and compensate later the lack of this term by doubling the sum of the pair-dependencies $\delta_{s,t}(v)$ for which $s \in \overline{\mathbf{V}_{\mathbf{C}(v)}}$ and $t \in \mathbf{V}_{\mathbf{C}(v)}$.

Global dependency scores in Eq. 26 are therefore redefined as follows:

$$\delta^\gamma(v) = \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \notin (\mathbf{V}_{\mathbf{C}(v)} \cup \mathbf{V}_{\mathbf{C}(s)})} \delta_{s,t}(v) + 2 \cdot \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \in \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) \quad (28)$$

With this further step, we can now use pivots to efficiently compute the exact global BC. In particular, let $\delta_{s, \mathbf{V}_{\mathbf{C}(v)}}^\gamma(v)$ and $\delta_{s, \overline{\mathbf{V}_{\mathbf{C}(v)}}}^\gamma(v)$ be the global dependency scores from node s on node v for destinations not belonging to $\mathbf{C}(s)$, but belonging to $\mathbf{C}(v)$, and the global dependency score from node s on node v for destinations not belonging to $\mathbf{C}(s)$ and $\mathbf{C}(v)$. Eq. 28 can be rewritten as follows:

$$\delta^\gamma(v) = \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} [2 \cdot \delta_{s, \mathbf{V}_{\mathbf{C}(v)}}^\gamma(v) + \delta_{s, \overline{\mathbf{V}_{\mathbf{C}(v)}}}^\gamma(v)] \quad (29)$$

Therefore, given a cluster $\mathbf{C}_i \in \mathbf{C}$ and all its equivalence classes $\mathbf{K}_{\mathbf{C}_i}$, we have:

$$\begin{aligned} \forall v \notin \mathbf{V}_{\mathbf{C}_i}, \mathbf{K}_i \in \mathbf{K}_{\mathbf{C}_i}, \\ \sum_{s \in \mathbf{K}_i} (2 \cdot \delta_{s, \mathbf{V}_{\mathbf{C}(v)}}^\gamma(v) + \delta_{s, \overline{\mathbf{V}_{\mathbf{C}(v)}}}^\gamma(v)) = |\mathbf{K}_i| \cdot (2 \cdot \delta_{k_i, \mathbf{V}_{\mathbf{C}(v)}}^\gamma(v) + \delta_{k_i, \overline{\mathbf{V}_{\mathbf{C}(v)}}}^\gamma(v)) \end{aligned} \quad (30)$$

Eq. 30 means that, during the back propagation phase, we should distinguish between contributions due to destinations inside the same cluster of v and contributions due to destinations outside the cluster of v .

Figure 4: Global SSSP explorations from pivots

For a better understanding of the formulas above, let us consider an illustrative example by leveraging again the clustered graph from Fig. 1 and the equivalence classes of cluster $\mathbf{C}_1$ from Fig. 2. The pivot node of the equivalence class composed of nodes $\{3, 4, 6, 14\}$ is node **14**. According to the proposed approach, we calculate the dependency scores from node **14** on all nodes of clusters $\mathbf{C}_2$ and $\mathbf{C}_3$ and multiply them by 4, avoiding to calculate the dependencies scores from nodes **3, 4** and **6**. This way, the computation time is divided by 4. However, while it is correct to multiply by 4 the dependency scores for nodes in $\mathbf{C}_2$ and $\mathbf{C}_3$, it is not for nodes belonging to the same cluster of the pivot (see Fig. 4a) since nodes **14, 3, 4, 6** of the class are equivalent only with reference to border nodes of cluster $\mathbf{C}_1$ (nodes **1,**

2). Therefore, we can not multiply by 4 the dependency scores on nodes **1, 2, 3, 4, 5, 6** since these scores are not the same when computed, for instance, from node **14** or node **4**. To avoid the problem, we put these dependency scores to 0 and we later compensate during SSSP explorations from a pivot node in $\mathbf{C}_2$ and $\mathbf{C}_3$ (see Fig. 4b).

Back-propagation. Differently from Brandes' algorithm, it is not possible to directly express the global dependency score^[3] of a node v , $\delta_{s,\bullet}^\gamma(v)$, in terms of the global dependency scores of w , $\delta_{s,\bullet}^\gamma(w)$, where $v \in \mathbf{P}_s(w)$. Indeed, when $\mathbf{C}(v) \neq \mathbf{C}(w)$ (i.e., when crossing a cluster), the set of destinations of w which do not belong to $\mathbf{C}(w)$ can be composed of both destinations belonging to $\mathbf{C}(v)$ and destinations not belonging to $\mathbf{C}(v)$: for the former, the pair-dependencies have to be multiplied by 2, whereas for the latter no further operation is needed (see Eq. 30).

To overcome this problem, we apply the classic recursive formula of Brandes' algorithm (Eq. 5) on a *vector* of contributions, propagating the global dependency scores $\delta_{s,\bullet}^\gamma(v)$. The dimensions of this vector of contributions correspond to the number of clusters, so that the contribution due to a destination t is assigned to $\delta_{s,\mathbf{V}_{\mathbf{C}(t)}}^\gamma(v)$. Formally, we have the following recursive formula:

$$\forall \mathbf{C}_i \in \mathbf{C} \setminus \mathbf{C}(s) : \delta_{s,\mathbf{V}_{\mathbf{C}_i}}^\gamma(v) = \sum_{w:v \in \mathbf{P}_s(w)} \frac{\sigma_{s,v}}{\sigma_{s,w}} * (\mathbb{1}_{w \in \mathbf{C}_i} + \delta_{s,\mathbf{V}_{\mathbf{C}_i}}^\gamma(w)), \quad (31)$$

where $\mathbb{1}_{w \in \mathbf{C}_i}$ represents a boolean variable equal to 1 if $w \in \mathbf{C}_i$, 0 otherwise^[4]. At the end of the back-propagation phase, we put the dependency scores of nodes v belonging to the same cluster of the (pivot) source node to 0, whereas the dependency scores of nodes belonging to the other clusters are computed using the following formula:

$$\delta_{s,\overline{\mathbf{V}_{\mathbf{C}(s)}}}^\gamma(v) = 2 \cdot \delta_{s,\mathbf{V}_{\mathbf{C}(s)}}^\gamma(v) + \sum_{\mathbf{C}_i \neq \mathbf{C}(v)} \delta_{s,\mathbf{V}_{\mathbf{C}_i}}^\gamma(v) \quad (32)$$

Finally, according to Eq. 30, $\delta_{s,\bullet}^\gamma(v)$ is multiplied by the cardinality of the equivalence class s belongs to.

^[3] $\delta_{s,\bullet}^\gamma(v)$ is equivalent to $\delta_{s,\overline{\mathbf{V}_{\mathbf{C}(s)}}}^\gamma(v)$

^[4] This is the part of contribution due to w as a destination

Alternative solution for back-propagation

We can consider a vector of two elements instead of $|\mathbf{C}|$ elements, so that the contribution due to a destination t is assigned to $\delta_{s, \mathbf{V}_{\mathbf{C}(v)}}^\gamma(v)$ or $\delta_{s, \overline{\mathbf{V}_{\mathbf{C}(v)}}}^\gamma(v)$ depending on whether t is in the same cluster as v , $\mathbf{C}(v)$, or not. During the back-propagation, the two contributions are propagated independently as follows:

$$\delta_{s, \mathbf{V}_{\mathbf{C}(v)}}^\gamma(v) = \begin{cases} \frac{\sigma_{s,v}}{\sigma_{s,w}} \cdot (1 + \delta_{s, \mathbf{V}_{\mathbf{C}(w)}}^\gamma(w)) & \text{if } \mathbf{C}(v) = \mathbf{C}(w), \\ 0 & \text{otherwise} \end{cases}$$

$$\delta_{s, \overline{\mathbf{V}_{\mathbf{C}(v)}}}^\gamma(v) = \begin{cases} \frac{\sigma_{s,v}}{\sigma_{s,w}} \cdot \delta_{s, \overline{\mathbf{V}_{\mathbf{C}(w)}}}^\gamma(w) & \text{if } \mathbf{C}(v) = \mathbf{C}(w), \\ \frac{\sigma_{s,v}}{\sigma_{s,w}} \cdot (1 + \delta_{s, \mathbf{V}_{\mathbf{C}(w)}}^\gamma(w) + \delta_{s, \overline{\mathbf{V}_{\mathbf{C}(w)}}}^\gamma(w)) & \text{otherwise} \end{cases} \quad (33)$$

At the end of back-propagation, we put the dependency scores of nodes v belonging to the same cluster of the (pivot) source node to 0, whereas the dependency scores of nodes not belonging to the same cluster of the source node are computed using the following formula:

$$\delta_{s, \overline{\mathbf{V}_{\mathbf{C}(s)}}}^\gamma(v) = 2 \cdot \delta_{s, \mathbf{V}_{\mathbf{C}(v)}}^\gamma(v) + \delta_{s, \overline{\mathbf{V}_{\mathbf{C}(v)}}}^\gamma(v) \quad (34)$$

Finally, according to Eq. 30, $\delta_{s, \bullet}^\gamma(v)$ is multiplied by the cardinality of the equivalence class s belongs to. However, this solution introduces an error. In fact, in Eq. 33, $\delta_{s, \overline{\mathbf{V}_{\mathbf{C}(w)}}}^\gamma(w)$ may contain contributions of destination nodes belonging to $\mathbf{V}_{\mathbf{C}(v)}$, which should be moved from $\delta_{s, \overline{\mathbf{V}_{\mathbf{C}(v)}}}^\gamma(v)$ to $\delta_{s, \mathbf{V}_{\mathbf{C}(v)}}^\gamma(v)$, so that they can be correctly multiplied by 2 as required by Eq. 34. This situation is a consequence of the presence of external shortest paths that leave and then re-enter the clusters.

The correction above has to be performed during the back-propagation phase for each border node such that *i*) there is at least one external shortest path starting from that border node, and *ii*) all the contributions of the border node have been computed, *i.e.*, when the border node is the current w . Consequently, for each pair of border nodes b_1, b_2 belonging to the same cluster $\mathbf{C}_i$, we need to compute the distance $d_{\mathbf{G}}(b_1, b_2)$ and the number of external shortest paths σ_{b_1, b_2}^{ext} between them. We perform this operation when searching for external nodes.

4 E1C-FastBC algorithm

In this section, we describe the *E1C-FastBC* algorithm, the implementation of the cluster-based solution introduced in the previous section. We also discuss a parallel version which takes advantage of a map/reduce-based approach.

4.1 Louvain clustering

To group nodes in clusters and minimize the number of border nodes, $|\mathbf{BN}|$, and consequently $|\mathbf{BN}_{C_i}|$ for each cluster $C_i \in \mathbf{C}$, we exploit a modularity-based clustering algorithm. *Modularity* is a scalar metric, defined in the range -1 and 1, which measures the density of links inside clusters as compared to links between them: the higher its value, the lower the number of inter-clusters links. Consequently, maximizing the modularity score reduces the number of border nodes in the clusters. This allows not only to keep low the complexity of the algorithm by reducing the size of the HSN and the number of nodes against which topological properties (normalized distances and normalized number of shortest paths) have to be computed, but also to maximise the chances of having few equivalence classes, each with many nodes, since smaller vectors (those storing the topological properties) increase the probability of having linear dependency among them and consequently a smaller number of classes. This is highly beneficial from the perspective of reducing SSSP explorations.

The Louvain method [25] is an example of modularity-based clustering technique. Its time complexity of $O(n \log^2 n)$ is very good compared to that of Brandes' algorithm. The Louvain algorithm runs in two phases which are iteratively repeated. In the first phase, each node is initially assigned to its own cluster and moved in the cluster of the neighbor which ensures the maximum increase of modularity, with respect to the previous configuration. This phase terminates when all nodes have been explored and no further modularity improvement is possible. In the second phase, a new graph is generated by considering the identified clusters as nodes, and the loops inside them as self-loops. Phase one is then repeated using the graph generated by the second phase. The two phases are iterated until a maximum of modularity is reached and a hierarchical structure of clusters has been formed. The output of the algorithm, and consequently the modularity of the identified clusters, may be affected by the order nodes are evaluated with. This order can also influence

the computation time. To improve solutions that are sub-optimal in terms of modularity, multiple runs of the algorithm can be performed over the same network, each associated to a different order for the analysis of the nodes.

4.2 Algorithm implementation

Alg. 1 reports the pseudo-code of the *E1C-FastBC* algorithm taking as input an undirected unweighted graph $\mathbf{G}$ and producing in output the exact values of BC for every node in $\mathbf{V}$.

The algorithm is composed of several phases. We provide a detailed description for all the intermediate phases, while the associated pseudo-code is provided for the most relevant ones in Alg. 1.

Algorithm 1 Pseudo-code of the E1C-FastBC algorithm

```

1: function E1CFastBC( $\mathbf{G}$ )
2:    $\mathbf{C} \leftarrow \text{modularityBasedClustering}(\mathbf{G})$ 
3:    $\mathbf{BN} \leftarrow \text{findBorderNodes}(\mathbf{G}, \mathbf{C})$ 
4:    $\mathbf{V}_{HSN} \leftarrow \text{buildHSN}(\mathbf{BN}, \mathbf{C}, \mathbf{G})$ 
5:    $\mathbf{EN} \leftarrow \text{findExternalNodes}(\mathbf{V}_{HSN}, \mathbf{C}, \mathbf{BN}, \mathbf{G})$ 
6:    $\mathbf{C}^* \leftarrow \text{updateClusters}(\mathbf{C}, \mathbf{EN})$ 
7:    $\delta^\lambda, \delta^\epsilon, \hat{\sigma}, \hat{d} \leftarrow \text{computeLocal}\delta(\mathbf{C}^*, \mathbf{C}, \mathbf{EN}, \mathbf{BN}, \mathbf{G})$ 
8:    $\mathbf{K} \leftarrow \text{findClasses}(\hat{\sigma}, \hat{d})$ 
9:    $\delta^\gamma \leftarrow \text{computeGlobal}\delta(\mathbf{K}, \mathbf{G}, \mathbf{C})$ 
10:  for  $v \leftarrow 1, \mathbf{V}$  do ▷ For all nodes
11:     $BC(v) \leftarrow \delta^\lambda(v) + \delta^\gamma(v) + \delta^\epsilon(v)$ 
12:  end for
13:  return BC
14: end function

```

At line 2, the Louvain, modularity-based clustering algorithm is exploited for splitting graph $\mathbf{G}$ into a set of clusters $\mathbf{C}$. 4.1. These clusters do not need to be explicitly stored in a dedicated data structure as they represent a view of the starting graph, filtered through a membership information stored in every node.

At line 3, we identify the set of border nodes $\mathbf{BN}$ by checking, for each node $v \in \mathbf{V}$, the existence of at least one neighbor belonging to a different cluster.

At line 4, the nodes building up the HSN, referred as $\mathbf{V}_{HSN}$, are retrieved. As detailed in the pseudo-code of Alg. 2, to build the HSN we first execute $|\mathbf{BN}|$ local BFS, each rooted in a border node used as source, *i.e.*, $s \in \mathbf{BN}$. The term local here refers to the fact that only nodes belonging to the same cluster of s , *i.e.*, $\mathbf{V}_{\mathbf{C}(s)}$, are crossed during the explorations. Each BFS returns the set of direct predecessors $\mathbf{P}_s(\mathbf{V}_{\mathbf{C}(s)})$ of every node in $\mathbf{V}_{\mathbf{C}(s)}$ on shortest paths from s . These sets are later

used at line 6 of Alg. 2 to cross the discovered shortest paths backwards starting from destinations t .

Each traversal returns the set of nodes lying on the shortest paths between a pair of border nodes of the same cluster: these nodes, together with the source and destination border nodes themselves, belong to the HSN and are therefore added to the set of all its nodes, *i.e.*, $\mathbf{V}_{HSN}$.

Algorithm 2 Pseudo-code of building HSN algorithm

```

1: function BUILDHSN(BN, C, G)
2:   for  $s \leftarrow \mathbf{BN}$  do
3:      $\mathbf{P}_s(\mathbf{V}_{\mathbf{C}(s)}) \leftarrow \text{BFS}(s, \mathbf{G}, \mathbf{V}_{\mathbf{C}(s)})$ 
4:   end for
5:   for  $s, t \leftarrow \mathbf{BN}$  where  $\mathbf{C}(s) == \mathbf{C}(t)$  do
6:      $\mathbf{V}_{HSN} \leftarrow \mathbf{V}_{HSN} \cup \text{crossSPBackwards}(t, \mathbf{P}_s(\mathbf{V}_{\mathbf{C}(s)}))$ 
7:   end for
8:   return  $\mathbf{V}_{HSN}$ 
9: end function

```

At line 5, we identify external nodes **EN** as detailed in Alg. 3. First, a BFS is executed from each source $s \in \mathbf{BN}$. In these explorations, only nodes of the HSN, $\mathbf{V}_{HSN}$, are considered. Each BFS returns the set of direct predecessors of every node in $\mathbf{V}_{HSN}$ on shortest paths from s , *i.e.*, $\mathbf{P}_s(\mathbf{V}_{HSN})$. Similarly to the previous step, shortest paths are crossed backwards from destination t to source s using the sets of predecessors and every crossed node not belonging to the cluster of s and t is added to the set of external node **EN**.

Algorithm 3 Pseudo-code of external nodes identification algorithm

```

1: function FINDEXTERNALNODES( $\mathbf{V}_{HSN}$ , C, BN, G)
2:   for  $s \leftarrow \mathbf{BN}$  do
3:      $\mathbf{P}_s(\mathbf{V}_{HSN}) \leftarrow \text{BFS}(s, \mathbf{G}, \mathbf{V}_{HSN})$ 
4:   end for
5:   for  $s, t \leftarrow \mathbf{BN}$  where  $\mathbf{C}(s) == \mathbf{C}(t)$  do
6:      $\mathbf{EN} \leftarrow \mathbf{EN} \cup \text{crossSPBackwards}(t, \mathbf{P}_s(\mathbf{V}_{HSN}))$ 
7:   end for
8:   return  $\mathbf{EN}$ 
9: end function

```

The extended clusters $\mathbf{C}^*$ are generated at line 6 of Alg. 1 from the original clusters, updated with the external nodes.

At line 7, we compute in each (extended) cluster, *i*) the local BC on every node, *i.e.*, δ^λ , *ii*) BC contributions δ^ϵ on external nodes, and *iii*) the topological properties of every node, *i.e.*, the normalized distances $\hat{d}$ and the normalized numbers of

shortest paths $\hat{\sigma}$, computed with respect to the set of border nodes belonging to the cluster of the node. These topological properties are subsequently used at line 8 to find equivalence classes (according to Corollary 3.4). A modified version of Brandes' algorithm enables the computation of all these metrics, as described in Alg. 4. The only difference compared to the canonical implementation of Brandes' algorithm is related to the back-propagation phase: in our case, the contributions due to the external nodes (as destinations) are not propagated, since they represent non-local destinations.

Algorithm 4 Pseudo-code of local BC computation

```

1: function COMPUTELOCAL $\delta(\mathbf{C}^*, \mathbf{C}, \mathbf{EN}, \mathbf{BN}, \mathbf{G})$ 
2:   for  $s \leftarrow \mathbf{V}$  do
3:      $\delta_{s\bullet}^\lambda(\mathbf{V}_{\mathbf{C}(s)}), \delta_{s\bullet}^\epsilon(\mathbf{EN}_{\mathbf{C}(s)}), \hat{\sigma}_s, \mathbf{BN}_{\mathbf{C}(s)}, \hat{d}_{\mathbf{G}}(s, \mathbf{BN}_{\mathbf{C}(s)}) \leftarrow \text{BrandesModifiedV1}($ 
                                                 $s, \mathbf{C}^*(s), \mathbf{C}(s), \mathbf{BN}, \mathbf{G})$ 
4:      $\delta^\lambda(\mathbf{V}_{\mathbf{C}(s)}) \leftarrow \delta^\lambda(\mathbf{V}_{\mathbf{C}(s)}) + \delta_{s\bullet}^\lambda(\mathbf{V}_{\mathbf{C}(s)})$ 
5:      $\delta^\epsilon(\mathbf{EN}_{\mathbf{C}(s)}) \leftarrow \delta^\epsilon(\mathbf{EN}_{\mathbf{C}(s)}) + \delta_{s\bullet}^\epsilon(\mathbf{EN}_{\mathbf{C}(s)})$ 
6:   end for
7:   return  $\delta^\lambda, \delta^\epsilon, \hat{\sigma}, \hat{d}$ 
8: end function

```

At line 9, we compute the global BC, δ^γ . As shown in Alg. 5, a second modified version of Brandes' algorithm, which exploits Eq. 30, Eq. 31 and Eq. 32, is run from one pivot k_i , randomly selected from each class $K_i \in \mathbf{K}$. From the explorations rooted at the class pivots, the global dependency scores, $\delta_{k_i, \overline{\mathbf{V}_{\mathbf{C}(k_i)}}}^\gamma(\mathbf{V})$ are computed on every other node v of the graph. The global BC of every node v is then obtained by summing the global dependency scores deriving from all the pivots.

Finally, at lines 10-12 of Alg. 1, the computed partial terms are summed up to obtain the exact BC values for each node.

Algorithm 5 Pseudo-code of global BC computation

```

1: function COMPUTEGLOBAL $\delta(\mathbf{K}, \mathbf{G}, \mathbf{C})$ 
2:   for  $K_i \leftarrow \mathbf{K}$  do ▷ For all classes
3:      $k_i \leftarrow \text{random}(K_i)$ 
4:      $\delta_{k_i, \overline{\mathbf{V}_{\mathbf{C}(k_i)}}}^\gamma(\mathbf{V}) \leftarrow \text{BrandesModifiedV2}(k_i, \mathbf{G}, \mathbf{C})$ 
5:      $\delta^\gamma(\overline{\mathbf{V}_{\mathbf{C}(k_i)}}) \leftarrow \delta^\gamma(\overline{\mathbf{V}_{\mathbf{C}(k_i)}}) + \delta_{k_i, \overline{\mathbf{V}_{\mathbf{C}(k_i)}}}^\gamma(\overline{\mathbf{V}_{\mathbf{C}(k_i)}}) \cdot |K_i|$ 
6:   end for
7:   return  $\delta^\gamma$ 
8: end function

```

4.3 Parallel implementation with map/reduce

The proposed *E1C-FastBC* algorithm can be parallelized since the execution of its sub-algorithms is highly parallelizable, with the only exception of the selected clustering algorithm. We can exploit *data parallelism* by performing the same operations on different partitions of a given graph leveraging the MapReduce paradigm since most of the computations are applied to each node of the graph.

Figure 5 reports the representation of the parallel version of *E1C-FastBC*, built using some key concepts introduced in Apache Spark, a popular big data processing engine that we use to run the tests reported in the experimental evaluation. Spark applications are generally defined in terms of *transformations* and *actions* that are applied to *Resilient Distributed Datasets* (RDDs). RDDs are immutable collections of data partitioned across the worker nodes of a cluster. Transformations and actions can be processed in parallel on such partitions. In particular, transformations are functions that produce new RDDs starting from the ones they are invoked on, whereas actions are functions that return the result of a computation performed over an RDD.

A Spark application is a collection of *jobs*, each created to perform an action, and executed by one or more *executors* deployed on the worker nodes of the cluster by running, in parallel, *tasks* over the partitions of an RDD. The tasks of the jobs

encapsulate all the transformations that have to be applied to RDDs. The latter are then collected at the end of the jobs by the master node of a cluster: such node hosts the *driver*, which is the process responsible for running the application. To process RDDs, jobs may also require other inputs that can possibly be shared across executors through the so-called *broadcast variables*: they are lazily copied between worker nodes during execution.

The parallel version of *E1C-FastBC* is a sequence of jobs, each implementing one or more sub-algorithms of the algorithm, as detailed in Fig. 5. The main execution flow is represented with solid arrows, whereas, with dashed arrows, we present data that are copied via broadcast among all Spark workers and needed to carry out the jobs. Each job executes a specific type of task, as illustrated in Fig. 5, and receives two classes of inputs: *i*) RDDs, which are used to guide parallelism, *i.e.*, the number of tasks, and *ii*) broadcast variables, which are used by every single task to process its own partition. In the following, we describe each job in terms tasks behaviors and needed inputs.

- *Job 1* organizes the graph into clusters (Alg. 1, line 2) by performing parallel executions of the Louvain method, using different configurations with the aim of selecting the one that produces the clustering with the best modularity score. The job takes as input the graph, passed as broadcast variable, and outputs the clusters. The starting RDD, which does not contain data, only enables parallel executions of multiple runs of the Louvain method.
- *Job 2* identifies border nodes (Alg. 1, line 3), by checking for each node the existence of at least one neighbor belonging to a different cluster. It requires as input the set of clusters and the graph, both passed as broadcast variables. The starting RDD contains all the nodes to analyze, and it is built from the whole set of graph vertices
- *Job 3* retrieves the HSN nodes (Alg. 1, line 4), by performing, for each border node, a constrained (intra-cluster) BFS. It needs the border nodes, the clusters and the graph as its inputs. A broadcast variable is used for all of them, but the set of border nodes is also used to build the starting RDD. Nevertheless, each execution requires the availability of the whole set of border nodes *i*) to avoid leaving clusters while performing BFSs and *ii*) to check whether a destination is a border node.

- *Job 4* discovers the external nodes (Alg. 1, lines 5-6) through BFSs bound to nodes belonging to the HSN. Consequently, compared to *Job 3*, it requires HSN nodes as additional input, passed in the form of broadcast variable, while the starting RDD is the same as that of *Job 3*. At the end, the job outputs the clusters extended with external nodes.
- *Job 5* computes the local BC, the BC on external nodes, the normalized distances and normalized numbers of shortest paths (Alg. 1, line 7). The job receives the graph, clusters, the extended clusters and the border nodes as inputs, all transferred as broadcast variables^[5]. The starting RDD of this job contains all the nodes of the graph.
- *Job 6* identifies the equivalence classes and their pivots (Alg. 1, line 8). The starting RDD contains the topological properties (normalized distances and normalized number of shortest paths) per node, while the inputs passed as broadcast variables are the same as the previous job.
- *Job 7* computes the global BC (Alg. 1, line 9) by using a starting RDD containing pairs composed of a pivot and the cardinality of its equivalence class. The only inputs passed via broadcast variables are the graph and the clusters.

Final BC values are obtained by aggregating all the previously calculated values. This step is performed entirely on the driver in a sequential manner. In all cases, except for *Job 1*, we use a node-level grain: all functions encapsulated in the various tasks are defined to work starting from a single node (simple node, border node or pivot).

5 Experimental evaluation

In this section, we report the main results of the experimental evaluation we conducted by testing the algorithm with both sequential and parallel executions on different types of graphs, and with different graphs and sizes.

We compare the execution times obtained with our algorithm to those obtained with other algorithms by using the *Algorithmic Speedup* (*AS*). Given two algorithms, a_1 and a_2 , the algorithmic speedup of a_1 over a_2 with p cores, noted as

^[5]We do not explicitly pass the set of external nodes as it is trivial to recognize them by leveraging the extended clusters.

$AS_p^{a_1/a_2}$, is defined as $T_p^{a_2}/T_p^{a_1}$, where $T_p^{a_2}$ and $T_p^{a_1}$ are the computation times obtained with p cores using algorithms a_2 and a_1 , respectively.

In particular, we will compare the *E1C-FastBC* algorithm, labelled with $\mathcal{E}$, with Brandes' algorithm, labelled as $\mathcal{B}$ and with the solution proposed in [13], labelled with $\mathcal{H}$. We chose this algorithm for comparison because it belongs to the same category as ours (cluster-based computation). However, due to the unavailability of source/executable code for $\mathcal{H}$, we only consider the *AS* metric in sequential mode, by relying on the indications provided by the authors in the paper for its computation (see Eq. 7 in [13]).

To further explore the performance of our solution, we also analyze the efficiency of the *E1C-FastBC* algorithm, based on the canonical definition of speedup. Specifically, the speedup obtained with p cores is defined as $S_p = T_s/T_p$, where T_s is the computation time in sequential mode and T_p is the computation time with p cores. The efficiency with p cores, noted as E_p , is then defined as S_p/p .

Finally, we also provide a breakdown analysis of the computation time of our solution, useful to investigate the contribution of each composing sub-algorithms. In all reported tests, we checked the accuracy of our solution by always observing zero error on BC values.

5.1 Datasets

In our tests, we consider both synthetic and real graphs.

For the first category, we focus on scale-free graphs generated using the implementation of the Barabasi-Albert model provided by the Python library NetworkX with a preferential attachment coefficient equal to 1. This way, we have graphs with $m = n - 1$ edges and an average degree approximately equal to 2, *i.e.*, double the preferential attachment coefficient. This choice is motivated by the features of the current implementation of our algorithm that benefits of high modularity. In other words, this class of dataset is considered as best-case scenario. In particular, to analyze the algorithm in terms of performance and scalability, we generate graphs with different sizes (see Table 3).

For the second category, we focus on some real graphs^[6] available in public datasets. Table 3 reports all the graphs we use, together with some relevant prop-

^[6]For each graph, we extract the largest connected component. Then, the latter is converted in an unweighted undirected graph.

erties. In particular, for each graph we consider the average degree (d_{avg}), the max degree (d_{max}) and the average clustering coefficient (cc_{avg}).

	Graph	n	m	d_{avg}	d_{max}	cc_{avg}
Synthetic	barabasi-albert	6,250	6,249	1.999	126	0.000
	barabasi-albert	12,500	12,499	"	225	"
	barabasi-albert	25,000	24,999	"	344	"
	barabasi-albert	50,000	49,999	"	463	"
	barabasi-albert	100,000	99,999	"	1,138	"
	barabasi-albert	200,000	199,999	"	676	"
	barabasi-albert	400,000	399,999	"	1,142	"
	barabasi-albert	800,000	799,999	"	1,587	"
Real	web-webbase-2001[28]	16,062	25,593	3.187	1,679	0.224
	ego-twitter[29]	22,322	31,823	2.851	238	0.072
	internet[28]	124,651	193,620	3.107	151	0.062
	lyon-road-network ^[7]	156,102	178,845	2.291	8	0.017
	email-euAll[30]	224,832	339,925	3.024	7,636	0.079

Table 3: Synthetic & real graphs. The names of the graphs are given in the first column, whereas the number of nodes and edges are given in the second and third columns. d_{avg} and d_{max} are the average and max degree, respectively. cc_{avg} is the average clustering coefficient.

All the datasets, except the one related to the Lyon road network, are scale-free graphs.

5.2 Experimentation testbed

The platform for our experiments is a server machine with 128 GB of RAM and 2 sockets Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, with 14 physical cores and 2 threads per core for a total of 28 logical cores per socket and 56 virtual cores in hyper threading, running Linux Debian as operating system.

Both Brandes' algorithm and *E1C-FastBC* are implemented in Scala and executed using Apache Spark 2.2.0 in standalone mode. In particular, we deploy a spark cluster composed of the master node and one worker node holding all available resources (56 cores and approximately 125GB of memory). Tests are performed employing a driver with one core and 30GB of memory, and a variable number of executors having a variable amount of memory and computing resources. Specifically, except for the case with one core (sequential execution) where there is only

^[7]This dataset was supplied by the French National Institute of Geographic Information (IGN). <https://www.ign.fr>

Figure 6: $AS_p^{E/B}$
 $p=[1,5,10,15,20,25]$

one executor that holds all the resources (*i.e.*, the single core and 90GB of memory), we fix the total number of cores for the experimentation and instantiate a number of executors such that each of them has 5 cores. The amount of memory is divided evenly among executors. For instance, with 5 cores we only deploy one executor with 90GB of memory, while with 10 cores two executors, each with 45GB of memory, are deployed.

The RDDs are decomposed in a number of partitions equal to the total number of cores.

5.3 Synthetic Graphs Analysis

Figure 6 shows the algorithmic speedup of *E1C-FastBC* over Brandes' algorithm, $AS_p^{E/B}$, obtained on the synthetic graphs in both sequential and parallel modes. In particular, we double the number of nodes from 25,000 to 800,000, and we consider a number of cores p equals to 1, 5, 10, 15, 20 and 25. We estimate by log-log regression the computation times with Brandes' algorithm for the graphs with 400,000 and 800,000 nodes, since executions would require weeks to complete, whereas our algorithm ends in maximum 31.5 minutes and 1.64 hours, respectively (sequential mode).

As highlighted by Fig. 6, $AS_p^{\mathcal{E}/\mathcal{B}}$ increases with the size of the graph, meaning that *E1C-FastBC* is not only faster than Brandes' algorithm but its speedup grows with larger graphs. This is due to the fact that the computation of our algorithm is strongly dependent on the number of border nodes ($|\mathbf{BN}|$), pivots ($|\mathbf{K}|$) and external nodes ($|\mathbf{EN}|$), in addition to the number of nodes (n) and edges (m). The first two variables increase slowly compared to the number of nodes and edges, while the third is almost always zero (only in one case it was equal to 2). The drawback is that $AS_p^{\mathcal{E}/\mathcal{B}}$ decreases as the number of cores increases. However, as the following efficiency analysis will further clarify, this does not mean that *E1C-FastBC* is less scalable than Brandes' algorithm, but rather that it needs very large graphs to better exploit the available computing resources. This statement is also confirmed by Fig. 6, which clearly shows that when the graph size is 400,000, a higher number of cores performs even better than a smaller one: in particular, we have that the $AS_p^{\mathcal{E}/\mathcal{B}}$ is better with 5 cores than with 1 core. To have a similar behavior even for a number of cores greater than 5, we should consider larger graphs.

To better understand the performance of *E1C-FastBC*, we investigate its efficiency with respect to that of Brandes' algorithm. Figure 7 reports the results of the efficiency analysis performed for the two algorithms. In both cases, it is possible to

Figure 8: $AS_{p=1}^{E/B}$ and $AS_{p=1}^{H/B}$ with synthetic graphs

observe that: *i*) the efficiency decreases as the number of cores increases and *ii*) for a given number of cores, it increases as the number of nodes increases.

However, it is worth to highlight that in the efficiency analysis, we use different but overlapping ranges of values for the number of nodes. In particular, for our solution we select larger graphs since we aim at showing that our algorithm scale well especially with very large graphs. In fact, the efficiency trend is almost the same in the two cases reported in Fig. 7. Moreover, given the maximum values of the number of nodes for the two algorithms (800,000 for ours, 200,000 for Brandes'), efficiency values are approximately the same with 5 cores (*i.e.*, the first considered parallel configuration) but significantly diverge as the number of cores increases. In particular, efficiency of *E1C-FastBC* decreases with a higher rate.

The reason for this behaviour lies in the reduced amount of computation required by our solution. Indeed, pivots allow to significantly decrease the number of (modified) Brandes' SSSP explorations performed on the whole graph, which represent the heaviest part of the whole computation (see Figs. 11 and 12), thus reducing the workload of each core.

Our solution also introduces another benefit: it allows to mitigate the variability of the computation times due to the different topological characteristics of the graphs and to the partitioning of data performed by Spark during executions. Indeed,

there may exist some partitions of the RDDs characterized by a high concentration of nodes that generate the most complex shortest path trees.

The time required to process these partitions directly impacts the time required to process the whole RDD, since partitions are processed in parallel. However, Spark tasks process each single partition sequentially. This aspect, combined with the fact that the number of partitions of an RDD is always equal to the number of cores and the default partitioning scheme of Spark distributes data evenly across the partitions, explains the punctual efficiency drops that can be observed in the plot related to Brandes' algorithm, when using graphs with 50,000 and 100,000 nodes and a low number of cores (see Fig. 7b).

Figure 8 reports the algorithmic speedup of *E1C-FastBC* over Brandes' algorithm on synthetic graphs in sequential settings, $AS_{p=1}^{\mathcal{E}/\mathcal{B}}$, along with the algorithmic speedup of the approach in [13] over Brandes, $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$, analytically computed based on Eq. 7 provided in [13]. Using such equation, it is possible to observe that: *i*) $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$ depends on the number of clusters ($|\mathbf{C}|$) and the average degree (d_{avg}), and *ii*) when $|\mathbf{C}| + 2 \gg d_{avg}/2$, it can be approximated with $d_{avg}/2$. Therefore, since for synthetic graphs the average degree is constant and the number of clusters increases with the number of nodes, $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$ is always approximately equal to 1 (the average degree is 2). In particular, the higher the number of clusters, the closer to 1 the $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$. We can thus conclude that our solution always outperforms the one in [13] with synthetic graphs.

5.4 Real Graphs Analysis

Figure 9 reports the results of the analysis of $AS_{p=1}^{\mathcal{E}/\mathcal{B}}$ and $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$ carried out on real graphs. $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$ is computed again using Eq. 7 provided in [13]. In all cases, our solution outperforms the one in [13].

To further confirm the considerations on the scalability of our solution, reported in the previous section, we analyze in the following both the algorithmic speedup and efficiency values of *E1C-FastBC* on the *lyon-road-network* graph, for which we observed a very high number of pivots (about 60% of the number of nodes) and the lowest algorithmic speedup factor. As shown in Fig. 10a, the AS is always greater than 1, thus confirming the usefulness of our solution, although the reported values are not comparable to those obtained on synthetic graphs (see Fig. 6) with a similar

number of nodes (100,000 and 200,000). In spite of this, the algorithm becomes more scalable and efficient than in the case of synthetic graphs with 100,000 and 200,000 nodes due to the increased amount of computation resulting from the higher number of border nodes, pivots and external nodes (see Fig. 10b).

5.5 Breakdown of computation time

In this section, we analyze the contributions of the different component sub-algorithms to the overall computation time of E1C-FastBC. The goal of this analysis

is to find bottlenecks that limit scalability and, consequently, room for further improvements.

We split the algorithm in seven parts (see Alg. 1): *Dataset Reading*, *Louvain Clustering* (line 2), *HSN & External Nodes* (lines 3-6), *Local BC & Top. Props.* (line 7), *Classes & Pivots* (line 8), *Global BC* (line 9) and *Final Accumulation* (lines 10-12). Figures 11 and 12 report the time taken by each of the parts above. Results have been obtained by running our algorithm on the synthetic graphs with 100,000 and 200,000 nodes in sequential and parallel modes. The level of parallelism (*i.e.*, total number of exploited cores) is indicated on the x-axis of the figures.

The parts exhibiting the highest computation time are *Global BC* and *Local BC & Top. Props.*. They both show a very good scalability as increasing resources up to the maximum limit (25 cores) translates into a reduction in time.

The third heaviest part is *Louvain Clustering*. Its computation time does not keep decreasing when augmenting parallelism beyond 5 cores. This is due to the fact that we have chosen to launch 10 parallel executions of the Louvain method in different configurations with the aim of selecting the one that produces the clustering with the best modularity score. This aspect highlights a potential bottleneck, since

Figure 12: Breakdown analysis of computation time on synthetic graph with 200,000 nodes

Figure 13: Breakdown analysis of computation time on real graph ego-twitter

the computation time of *Louvain Clustering* at high levels of parallelism becomes comparable with those related to *Global BC* and *Local BC & Top. Props.*.

As already discussed in Subsec. 5.3, the number of external nodes for our synthetic graphs is almost always equal to zero. Therefore, the contribution of *HSN & External*

Nodes is not relevant as well as those of *Classes & Pivots* and *Final Accumulation*. In particular, the latter is a sequential step entirely performed on the driver. Therefore it does not vary with the number of cores.

For *Dataset Reading*, computation time slowly increases with the number of cores due to the overhead introduced for creating the initial RDD, by reading data from the file system, with a number of partitions equal to the number of cores. Even for this step, the computation time becomes comparable with those obtained for *Local BC & Top. Props.* and *Global BC* when the parallelism increases.

It is worth to note that synthetic graphs represent a sort of best-case scenario: the very low number of border nodes and pivots, together with the almost complete absence of external nodes, allows for excellent performance. A more realistic scenario is analyzed in the following, by focusing on a real graph.

Figure 13 reports the time taken by each part of the algorithm on the ego-twitter graph^[8], in sequential and parallel modes. In this case, *HSN & External Nodes* becomes the second heaviest contribution with values comparable to those of *Local BC & Top. Props.*. This is mainly due to the increased number of external nodes and confirms the importance of achieving ideal clustering. Similar considerations as those made for synthetic graphs apply as well to the remaining contributions from the breakdown analysis related to the ego-twitter graph.

6 Conclusion

In this paper, we presented a very fast algorithm for performing the exact computation of BC in undirected graphs. The algorithm exploits clustering and structural properties of graphs to reduce the computing time. In particular, the algorithm exhibits an impressive speedup (compared with Brandes' algorithm and the ones proposed in [13]), especially for very large scale-free graphs with an attachment coefficient $m = 1$. A significant speedup is achieved also with other kinds of graphs, as demonstrated by the results obtained with real graphs. The reduction of the computation time is mainly due to the adoption of pivots, *i.e.*, nodes that contribute equally to the dependency score of other graph nodes.

The paper described both a sequential and a Map-Reduce parallel version of the algorithm implemented in Scala over Spark. The experimental analysis, performed

^[8]We don't use lyon-road-network graph as in Subsection 5.4 because of the Out of Memory error that arise when running the algorithm in profiling mode.

with reference to the number of cores exploited for computation, revealed that the efficiency is slightly lower than Brandes' algorithm but it increases with graph size. In fact the granularity per Spark-task of the SSSP computations is small when graphs are not very large due to the relative low number of pivots.

The speedup of EIC-FastBC strongly depends on the number of pivots; thus clustering and modularity play a key role for the computation time of the algorithm. As future work, we aim to study other clustering methods for more effectively identifying border nodes in (synthetic and real) graphs with different topologies. Finally, we will investigate a better mapping of the algorithm on distributed resources, when data-parallelism is exploited, by improving locality especially when different Spark executors are used.

Abbreviations

BC: Betweenness Centrality; BFS: Breadth-First Search; SSSP: Single-Source Shortest Path; HSN: Hierarchical Sub-Network; RDD: Resilient Distributed Dataset; AS: Algorithmic Speedup; S: Speedup; T: Computation Time.

Declarations

Acknowledgements

Not applicable.

Funding

This work has been supported by the French ANR research project PROMENADE (grant number ANR-18-CE22-0008) and by the Italian MIUR PRIN research project GAUSS (grant number 2015KWREMX).

Availability of data and materials

The datasets related to real graphs analysed during the current study are available in the *Stanford Large Network Dataset Collection* repository, <https://snap.stanford.edu/data/>, and *The Network Data Repository with Interactive Graph Analytics and Visualization* repository, <http://networkrepository.com/>. The datasets related to *lyon-road-network* graph and synthetic graphs are available from the corresponding author.

Ethics approval and consent to participate

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Consent for publication

Not applicable.

Authors' contributions

C.D. contributed to the definition and implementation of the algorithms, data processing, experiments, results analysis, validation, and writing of the draft of the paper. L.G. contributed to the refinement of the algorithms, data processing, experiments, results analysis, validation, and writing of the draft of the paper. A.F. co-supervised the design and implementation of the work and contributed in the inception phase to the definition of the algorithms and the techniques presented in the paper. E.Z. supervised the design and implementation of the work, proposed the algorithms and the techniques presented in the paper and elaborated them in collaboration with A.F. A.F. and E.Z. contributed to results analysis, validation, and review & editing of the paper. A.F. and E.Z. contributed to the funding acquisition. All the authors have approved the final version of this paper submitted to publication.

Authors' information

Cecile Daniel (cecile.daniel@univ-eiffel.fr) is a Ph.D Student at University Gustave Eiffel and University of Lyon (ENTPE), France.

Angelo Furno (angelo.furno@univ-eiffel.fr) is a Researcher at University Gustave Eiffel and University of Lyon (ENTPE), France.

Lorenzo Goglia (lorenzo.goglia@unisannio.it) is a Ph.D Student at the University of Sannio, Benevento, Italy.

Eugenio Zimeo (zimeo@unisannio.it) is an Associate Professor at the University of Sannio, Benevento, Italy, and deputy director of the CINI national Lab on Smart Cities and Communities.

Author details

¹LICIT UMR T9401, University Gustave Eiffel, University of Lyon, ENTPE, Lyon, France. ²Dept. of Engineering, University of Sannio, Benevento 82100, Italy.

References

- Freeman, L.C.: A set of measures of centrality based on betweenness. *Sociometry* **25**, 35–41 (1997)
- Borgatti, S.P., Mehra, A., Brass, D.J., Labianca, G.: Network analysis in the social sciences. *Science* **323**(5916), 892–895 (2009)
- King, D., Shalaby, A.: Performance metrics and analysis of transit network resilience in Toronto. *Transportation Research Record* (2016)
- Zhang, Y., Wang, X., Zeng, P., Chen, X.: Centrality characteristics of road network patterns of traffic analysis zones. *Transportation Research Record: Journal of the Transportation Research Board* **2256**, 16–24 (2011)
- Berche, B., von Ferber, C., Holovatch, T., Holovatch, Y.: Resilience of public transport networks against attacks. *The European Physical Journal B* **71**(1), 125–137 (2009)
- Furno, A., El Faouzi, N.-E., Sharma, R., Zimeo, E.: Two-level clustering fast betweenness centrality computation for requirement-driven approximation. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 1289–1294 (2017). IEEE
- Furno, A., Faouzi, N.E., Sharma, R., Cammarota, V., Zimeo, E.: A graph-based framework for real-time vulnerability assessment of road networks. In: 2018 IEEE International Conference on Smart Computing, SMARTCOMP 2018, Taormina, Sicily, Italy, June 18–20, 2018, pp. 234–241 (2018). doi:10.1109/SMARTCOMP.2018.00096
- Holme, P., Kim, B.J., Yoon, C.N., Han, S.K.: Attack vulnerability of complex networks. *Physical Review E* **65**(5), 056109 (2002)
- Carpenter, T., Karakostas, G., Shallcross, D.: Practical issues and algorithms for analyzing terrorist networks. *Proceedings of the Western Simulation MultiConference* (2002)
- Floyd, R.W.: Algorithm 97: Shortest path. *Commun. ACM* **5**(6), 345 (1962). doi:10.1145/367766.368168
- Brandes, U.: A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology* **25**(163) (2001)
- Suppa, P., Zimeo, E.: A clustered approach for fast computation of betweenness centrality in social networks. In: 2015 IEEE International Congress on Big Data, pp. 47–54 (2015). doi:10.1109/BigDataCongress.2015.17
- Li, Y., Li, W., Tan, Y., Liu, F., Cao, Y., Lee, K.Y.: Hierarchical decomposition for betweenness centrality measure of complex networks. In: *Sci. Rep.* **7** (2017)
- Barabási, A.-L., Albert, R.: Emergence of Scaling in Random Networks. *Science* **286**(5439), 509–512 (1999). doi:10.1126/science.286.5439.509
- Bader, D.A., Madduri, K.: Parallel algorithms for evaluating centrality indices in real-world networks. In: *Parallel Processing, 2006. ICPP 2006. International Conference On*, pp. 539–550 (2006). IEEE
- Madduri, K., Ediger, D., Jiang, K., Bader, D.A., Chavarria-Miranda, D.: A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium On*, pp. 1–8 (2009). IEEE
- Shi, Z., Zhang, B.: Fast network centrality analysis using GPUs. *BMC Bioinformatics* **12**(1), 149 (2011). doi:10.1186/1471-2105-12-149
- Kourtellis, N., Morales, G.D.F., Bonchi, F.: Scalable online betweenness centrality in evolving graphs. 2016 IEEE 32nd International Conference on Data Engineering (ICDE) **00**, 1580–1581 (2016). doi:10.1109/ICDE.2016.7498421

19. Brandes, U., Pich, C.: Centrality estimation in large networks. *International Journal of Bifurcation and Chaos* **17**(07), 2303–2318 (2007)
20. Geisberger, R., Sanders, P., Schultes, D.: Better approximation of betweenness centrality. In: *ALLENEX*, pp. 90–100 (2008). SIAM
21. Riondato, M., Upfal, E.: Abra: Approximating betweenness centrality in static and dynamic graphs with rademacher averages **12**(5) (2018). doi:10.1145/3208351
22. Puzis, R., Elovici, Y., Zilberman, P., Dolev, S., Brandes, U.: Topology manipulations for speeding betweenness centrality computation. *Journal of Complex Networks* **3**(1), 84–112 (2014). doi:10.1093/comnet/cnu015
23. Sariyüce, A.E., Kaya, K., Saule, E., Çatalyürek, Ü.V.: Graph manipulations for fast centrality computation. *ACM Transactions on Knowledge Discovery from Data (TKDD)* **11**(3), 26 (2017)
24. Baglioni, M., Geraci, F., Pellegrini, M., Lastres, E.: Fast exact computation of betweenness centrality in social networks. In: *International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2012, Istanbul, Turkey, 26–29 August 2012*, pp. 450–456 (2012). doi:10.1109/ASONAM.2012.79
25. Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* **2008**(10), 10008 (2008). doi:10.1088/1742-5468/2008/10/p10008
26. Erdős, D., Ishakian, V., Bestavros, A., Terzi, E.: A divide-and-conquer algorithm for betweenness centrality (2014). doi:10.1137/1.9781611974010.49
27. Daniel, C., Furno, A., Zimeo, E.: Cluster-based computation of exact betweenness centrality in large undirected graphs. In: *2019 IEEE International Conference on Big Data (Big Data)*, pp. 603–608 (2019). IEEE
28. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: *AAAI* (2015). <http://networkrepository.com>
29. McAuley, J., Leskovec, J.: Learning to discover social circles in ego networks. *NIPS* **1**, 539–547 (2012)
30. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graph evolution: Densification and shrinking diameters. *ACM Trans Knowledge Discov Data* **1** (2006)

Figures

Figure 1

Example of clustering.

Figure 2

Classes of equivalent nodes in the blue cluster C1

Figure 3

Example of external node found through the HSN.

Figure 4

Global SSSP explorations from pivots

Figure 5

Map-reduce, Spark-based description of E1C-FastBC.

Figure 6

Please see the Manuscript PDF file for the complete figure caption.

(a) E1C-FastBC algorithm

(b) Brandes' algorithm

Figure 7

Ep analysis

Figure 8

Please see the Manuscript PDF file for the complete figure caption.

Figure 9

Please see the Manuscript PDF file for the complete figure caption.

(a) $AS_{p=[1,5,10,15,20,25]}^{\mathcal{E}/\mathcal{B}}$

(b) E_p

Figure 10

Lyon-road-network analysis

Figure 11

Breakdown analysis of computation time on synthetic graph with 100,000 nodes

Figure 12

Breakdown analysis of computation time on synthetic graph with 200,000 nodes

Figure 13

Breakdown analysis of computation time on real graph ego-twitter