Daniel *et al. J Big Data* (2021) 8:92
https://doi.org/10.1186/s40537-021-00483-1

Journal of Big Data

RESEARCH Open Access

Fast cluster-based computation of exact betweenness centrality in large graphs

Cecile Daniel[1*], Angelo Furno[1], Lorenzo Goglia[2] and Eugenio Zimeo[2*]

*Correspondence:
cecile.daniel@univ-eiffel.fr;
zimeo@unisannio.it
[1] LICIT UMR_T9401,
University Gustave Eiffel,
University of Lyon, ENTPE,
Lyon, France
[2] Department of Engineering,
University of Sannio,
82100 Benevento, Italy

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 [3–7], 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.

In this paper, we propose an algorithm based on clustering, inspired by previous work on approximated BC computation [6, 12], 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. 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 algorithm.

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 Barabási-Albert graphs [14] represent an important target of our analysis, 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. "Related work" section positions the paper with reference to the main results from the literature. "Background" section 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. "Clustering and BC computation" section presents the main properties we exploit to define the algorithm when clustering is used to identify classes of equivalent nodes. "E1C-FastBC algorithm" section 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. "Experimental evaluation" section reports on the experimental results obtained by running the proposed algorithm on several synthetic and real graphs characterized by different sizes and topological properties. "Mathematical foundations" section is dedicated to the formal proofs of the theorem and claims used in our algorithm for fast BC computation. Finally, "Conclusion" summarizes the results and discusses the limits of the proposed solution by highlighting possible future improvements.

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) reducing complexity by decomposing graphs in clusters. It is worth noting that some proposals may belong to multiple categories since in many cases, techniques falling in a category can be complemented with other ones to further reducing computation time.

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. In [17], the authors propose a MPI-based parallel implementation of the adaptive sampling algorithm KADABRA [18]. Other efforts in this direction try to exploit the large amount of cores available on GPUs to better exploit parallelism [19].

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 [20] 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. *MR-IBC* [21] is a MapReduce-based incremental algorithm for BC computation in large-scale dynamic networks that supports edge addition or removal. The paper exploits distributed computing to achieve scalability and reduce computing time. Even in this case, the focus is on changes related to one single edge. The solution proposed in [22], instead, handles batches of updates in parallel. In particular, it exploits a bi-connected component decomposition technique along with some structural properties to improve performance.

Approximated computation These algorithms aim at achieving low computation time by calculating approximated BC values. Brandes and Pich proposed in [23] 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 [24]. 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.

KADABRA [18] is an adaptive sampling algorithm to approximate betweenness centrality. In particular, it adopts a probabilistic approach: BC of a node v is seen as the probability that, given two randomly selected nodes s and t and a randomly selected shortest path p between them, v belongs to p. The algorithm allows to specify the maximum absolute error and the probability that error is guaranteed.

A similar approach is followed in ABRA [25], 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 [26] 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 [27], the authors use both compression and splitting techniques, including the ones developed in [26], 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 [28] 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.

Reducing complexity by decomposing graphs in clusters 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 [29] 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, typically used 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 hypotheis 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 [30]. 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.

Very recently, a Graph Neural Network (GNN) based model to approximate betweenness and closeness centrality has been proposed [31]. This work, among other similar ones [32], demonstrates that the efficient computation of the BC is a topic of great interest even in the field of deep learning and, particularly, graph neural networks.

In this paper, we propose a technique to reduce the time needed for computing exact values of BC in undirected graphs by: (i) computing BC as the sum of two main contributions, *local* for each cluster and *global* among clusters (category e), (ii) reducing the SSSP explorations for the global phase through the identification of pivot nodes (category c), and (iii) considering HSN-based corrections on local contributions and the properties of undirected graphs to completely remove errors during computation (which affected the first proposal in [12]). This paper extends our previous proposal in [33], by significantly improving the algorithm and its implementations that now have been tested also with real graphs. Moreover, we formally prove the correctness of the proposed technique.

It is worth noting that our algorithm could be complemented by algorithms from different aforementioned categories, such as finer-grained parallelism from the first category or compression-based techniques exploiting graphs topological properties as the ones falling within the second category. Conversely, incremental and approximated computations are approaches for specific classes of applications that regard slowly changing graphs or rank-based exploitation of BC, respectively, which we consider out of the scope of this paper.

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:

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

$BC(v)$ 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.

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:

Table 1 Notation

Notation	Description
$\mathbf{G}$	Undirected unweighted input graph
$\hat{\mathbf{G}}$	A connected sub-graph of $\mathbf{G}$
$\mathbf{V}$	Set of vertices of $\mathbf{G}$ ($\lvert\mathbf{V}\rvert = n$)
$\mathbf{V}_{\hat{\mathbf{G}}}$	Set of vertices of $\mathbf{G}$ inducing $\hat{\mathbf{G}}$ (Set of vertices of $\hat{\mathbf{G}}$)
$\overline{\mathbf{V}_{\hat{\mathbf{G}}}}$	Set of vertices in $\mathbf{V} \setminus \mathbf{V}_{\hat{\mathbf{G}}}$
$\mathbf{V}_{HSN}$	Set of vertices of HSN
$\mathbf{E}$	Set of edges of $\mathbf{G}$ ($\lvert\mathbf{E}\rvert = m$)
$e_{s,t}$	Edge connecting vertices s and t
$d_{\mathbf{G}}(s, t)$	Distance between vertices s and t in $\mathbf{G}$
$\hat{d}_{\mathbf{G}}(s, t)$	Normalized distance between vertices s and t in $\mathbf{G}$
$\sigma_{s,t}$	Number of shortest paths between vertices s and t
$\sigma_{s,t}(v)$	Number of shortest paths between vertices s and t which cross vertex v
$\hat{\sigma}_{s,t}$	Normalized number of shortest paths between vertices s and t
$\mathbf{P}_s(v)$	Set of direct predecessors of vertex v on shortest paths from vertex s
$\mathbf{P}_s(\mathbf{V})$	Set of direct predecessors of vertices in $\mathbf{V}$ on shortest paths from vertex s
$BC(v)$	Betweenness centrality of vertex v
$\delta_{s,t}(v)$	Pair-dependency of pair of vertices (s, t) on the intermediary vertex v
$\delta_{s,\cdot}(v)$	Dependency score of vertex s on vertex v due to all destination vertices
$\delta_{s,\mathbf{V}_{\hat{\mathbf{G}}}}(v)$	dependency score of vertex s on vertex v due to all destination vertices in $\mathbf{V}_{\hat{\mathbf{G}}}$
$\mathbf{C}$	Set of clusters of $\mathbf{G}$
$\mathbf{C}_i$	A generic cluster in $\mathbf{C}$
$\mathbf{C}(v)$	The cluster vertex v belongs to
$\mathbf{C}^*$	Set of extended clusters in $\mathbf{G}$
$\mathbf{C}_i^*$	A generic extended cluster in $\mathbf{C}^*$
$\mathbf{K}$	Set of all the equivalence classes
$\mathbf{K}_i$	An equivalence class
$\mathbf{K}_{\mathbf{C}_i}$	Set of equivalence classes of cluster $\mathbf{C}_i$
$\mathbf{P}$	Set of all the pivots
k_i	Pivot node of the equivalence class $\mathbf{K}_i$
$\mathbf{EN}$	Set of all the external nodes
$\mathbf{EN}_{\mathbf{C}_i}$	Set of external nodes of cluster $\mathbf{C}_i$
$\mathbf{BN}$	Set of all the border nodes
$\mathbf{BN}_{\mathbf{C}_i}$	Set of border nodes of cluster $\mathbf{C}_i$
$\mathbf{BN}_{\mathbf{C}_i}(s, t)$	Set of border nodes of cluster $\mathbf{C}_i$ on shortest paths from $s \in \mathbf{V}_{\mathbf{C}_i}$ to $t \in \overline{\mathbf{V}_{\mathbf{C}_i}}$
b_i	A generic border node in $\mathbf{BN}$
$\delta^{\gamma}_{s,\cdot}(v)$	Global dependency score of s on v due to all $t \in \overline{\mathbf{V}_{\mathbf{C}(s)}}$ (same as $\delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(s)}}}(v)$)
$\delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(v)}}(v)$	Global dependency score of s on v due to all $t \in (\overline{\mathbf{V}_{\mathbf{C}(s)}} \cap \mathbf{V}_{\mathbf{C}(v)})$
$\delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(v)}}(\mathbf{V}_{\mathbf{C}(v)})$	Global dependency score of s on vertices in $\mathbf{V}_{\mathbf{C}(v)}$ due to all $t \in (\overline{\mathbf{V}_{\mathbf{C}(s)}} \cap \mathbf{V}_{\mathbf{C}(v)})$
$\delta^{\gamma}(v)$	Sum of all the global dependency scores (global BC) on v
$\delta^{\gamma}(\mathbf{V})$	Sum of all the global dependency scores (global BC) on vertices in $\mathbf{V}$
$\delta^{\lambda}_{s,\cdot}(v)$	Local dependency score of s on v due to all $t \in \mathbf{V}_{\mathbf{C}(s)} = \mathbf{V}_{\mathbf{C}(v)}$
$\delta^{\lambda}_{s,\cdot}(\mathbf{V})$	Local dependency score of s on vertices in $\mathbf{V}$ due to all $t \in \mathbf{V}_{\mathbf{C}(s)} = \mathbf{V}_{\mathbf{C}(v)}$
$\delta^{\lambda}(v)$	Sum of all the local dependency scores (local BC) on v
$\delta^{\lambda}(\mathbf{V})$	Sum of all the local dependency scores (local BC) on vertices in $\mathbf{V}$
$\delta^{\epsilon}_{s,\cdot}(v)$	Dependency score of s on v, as external node, due to all $t \in \mathbf{V}_{\mathbf{C}(s)}$
$\delta^{\epsilon}_{s,\cdot}(\mathbf{EN})$	Dependency score of s on external nodes $\mathbf{EN}$ due to all $t \in \mathbf{V}_{\mathbf{C}(s)}$
$\delta^{\epsilon}(v)$	Sum of all the dependency scores on v as external node
$\delta^{\epsilon}(\mathbf{EN}_{\mathbf{C}(s)})$	Sum of all the dependency scores on external nodes of cluster $\mathbf{C}(s)$

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

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

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

BC can thus be redefined in terms of dependency score:

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

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:

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

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 backtracking 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 $\mathbf{P}_{\boldsymbol{s}}(v)$ 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 $\delta_{s,\cdot}(v)$ that accumulates the contribution of the source on node v due to all destinations during the back-propagation step.

Equivalence class

To reduce the number of explorations and thus lower the BC 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:

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

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.

Clustering and BC computation

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 $\overline{\mathbf{V}_{\hat{\mathbf{G}}}}$.

In this paper, to easily identify reference nodes, we use clustering, and to increase the chances of identifying a low number of equivalence classes, we consider a clustering technique based on modularity, which 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 proposed approach relies on a set of mathematical properties that, for the sake of readability, are introduced and used in the following subsections, but proved in section "Mathematical foundations", at the end of the paper.

Equivalence class with clustering

Let us assume 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).

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. To this end, we can leverage the following theorem (see "Mathematical foundations" section, Theorem 6.1, for formal proof).

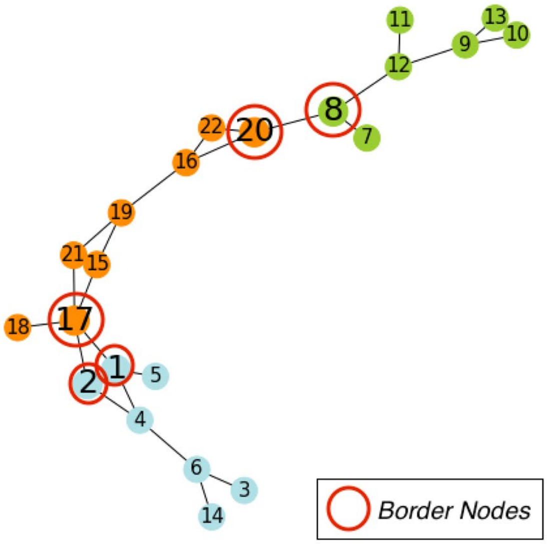
Let $k \in \mathbb{R}^+$ and $l \in \mathbb{R}$, let $\mathbf{C}_i$ be a generic cluster of graph $\mathbf{G}$ with border nodes $\mathbf{BN}_{\mathbf{C}_i}$, and $s, p \in \mathbf{V}_{\mathbf{C}_i}$. If $\forall\, b_j \in \mathbf{BN}_{\mathbf{C}_i}\ \sigma_{s,b_j} = k \cdot \sigma_{p,b_j}$ and $d_{\mathbf{G}}(s,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

In other words, any given pair of nodes $s,\ p$ belonging to the sub-graph induced by nodes in cluster $\mathbf{C}_i$ (i.e., $s, p \in \mathbf{V}_{\mathbf{C}_i}$), produces the same dependency score on all nodes $v \in \overline{\mathbf{V}_{\mathbf{C}_i}}$ for destinations $t \in \overline{\mathbf{V}_{\mathbf{C}_i}}$ if the distances 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.

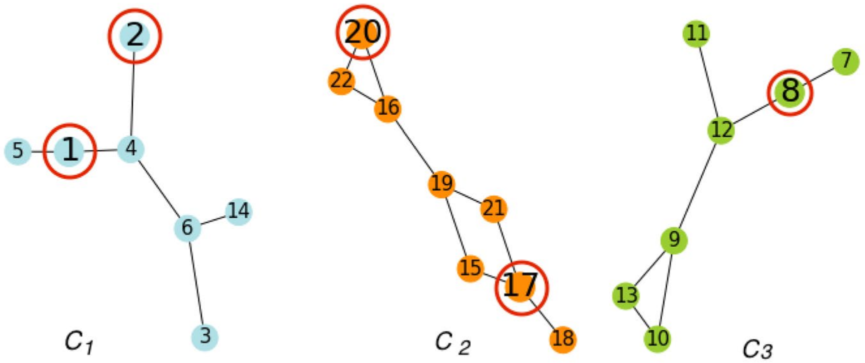
From the previous theorem, we can derive the following corollary (formally proved in section "Mathematical foundations" as Corollary 6.1):

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 $\delta_{s,\overline{\mathbf{V}_{\mathbf{C}_i}}}(v) = \delta_{p,\overline{\mathbf{V}_{\mathbf{C}_i}}}(v),\ \forall v \in \overline{\mathbf{V}_{\mathbf{C}_i}}$, where $\hat{d}_{\mathbf{G}}(s, b_j)$ represents the normalized distance of the generic node s to the generic border node b_j, defined as follows:

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



Example of clustered graph



Three sub-graphs originated by clustering the graph

Fig. 1 Example of clustering

and $\hat{\sigma}_{s,b_j}$ represents the normalized number of shortest paths from the generic node s to the generic border node b_j, and is defined as:

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

Normalized distance and normalized shortest paths simplify the identification of classes since they are characterized by the same vector of normalized distances and shortest paths as explained below with an example and the support of a graphical representation.

Let **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

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

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

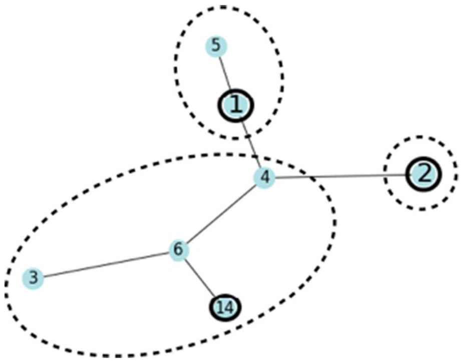


Fig. 2 Classes of equivalent nodes in the blue cluster C_1

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].

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

$$
\begin{aligned}
BC(v) &= \sum_{s \in \mathbf{V}} \delta_{s,\cdot}(v) \\
&= \sum_{s \in \mathbf{V}} \sum_{t \in \mathbf{V_{C(s)}}} \delta_{s,t}(v) + \sum_{s \in \mathbf{V}} \sum_{t \notin \mathbf{V_{C(s)}}} \delta_{s,t}(v) \\
&= \underbrace{\sum_{s \in \mathbf{V_{C(v)}}} \sum_{t \in \mathbf{V_{C(v)}}} \delta_{s,t}(v)}_{\text{sum of local dependency scores} = \delta^{\lambda}(v)} + \underbrace{\sum_{s \in \mathbf{V}} \sum_{t \notin \mathbf{V_{C(s)}}} \delta_{s,t}(v)}_{\text{sum of global dependency scores} = \delta^{\gamma}(v)} \\
&+ \underbrace{\sum_{s \notin \mathbf{V_{C(v)}}} \sum_{t \in \mathbf{V_{C(s)}}} \delta_{s,t}(v)}_{\text{sum of dependency scores on external node} = \delta^{\epsilon}(v)}
\end{aligned}
\tag{7}
$$

---

[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.

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, $\delta^{\lambda}_{s,\cdot}(v)$, 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 section "Algorithm implementation").

The *global* dependency score of a node s on a node v, $\delta^{\gamma}_{s,\cdot}(v)$, 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 $\delta^{\epsilon}_{s,\cdot}(v)$, 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.

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 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.

---

[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.

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 subgraph 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).

Dependency score of pivots

From the equivalence class relationship described in section "Equivalence class", 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:

$$\sum_{s\in\mathbf{K_i}} \delta_{s,\overline{\mathbf{V}_{\mathbf{C}_i}}}(v) = |\mathbf{K}_i| \cdot \delta_{k_i,\overline{\mathbf{V}_{\mathbf{C}_i}}}(v) \ \ \forall v \in \overline{V_{\mathbf{C}_i}}, \mathbf{K}_i \in \mathbf{K}_{\mathbf{C}_i}. \tag{8}$$

This equation can be exploited to speed up computation of BC building on Brandes' algorithm and SSSP explorations, but only holds if $v \in \overline{\mathbf{V}_{C_i}}$. 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 $v \in \mathbf{V}_{C_i}$[3].

First of all, let us decompose the global dependency scores from Eq. 7 based on the cluster of node v as follows:

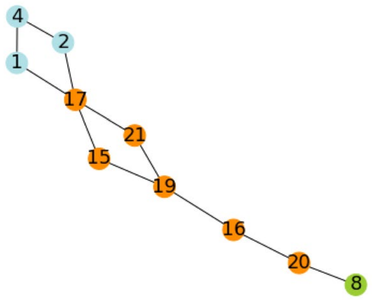
$$\delta^{\gamma}(v) = \sum_{s\notin\mathbf{V}_{\mathbf{C}(v)}} \sum_{t\notin(\mathbf{V}_{\mathbf{C}(v)}\cup\mathbf{V}_{\mathbf{C}(s)})} \delta_{s,t}(v) + \sum_{s\notin\mathbf{V}_{\mathbf{C}(v)}} \sum_{t\in\mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) + \sum_{s\in\mathbf{V}_{\mathbf{C}(v)}} \sum_{t\notin\mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) \tag{9}$$

The previous equation can be further simplified by considering the following claim, which is proved in section "Mathematical foundations" as Claim 6.1. In undirected graphs:

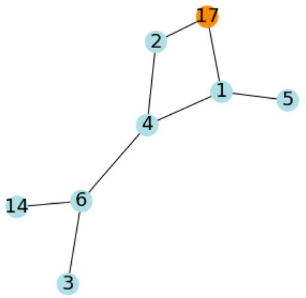
$$\sum_{s\in\mathbf{V}_{\mathbf{C}(v)}} \sum_{t\notin\mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) = \sum_{s\notin\mathbf{V}_{\mathbf{C}(v)}} \sum_{t\in\mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) \tag{10}$$

---

[3] In our previous version of the algorithm, we used Eq. 8 without taking into account the cluster of v and the pivots were chosen to minimize the error. Here, we avoid such an error during the computation of global dependency scores by exploiting the properties of undirected graphs, as explained later.



Hierarchical Sub-Network from clusters in Figure 1b



Cluster $\mathbf{C}_1$, extended with the external node **17**

Fig. 3 Example of external node found through the HSN

By relying on Eq. 10, 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. 9 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. 9 are therefore redefined as follows:

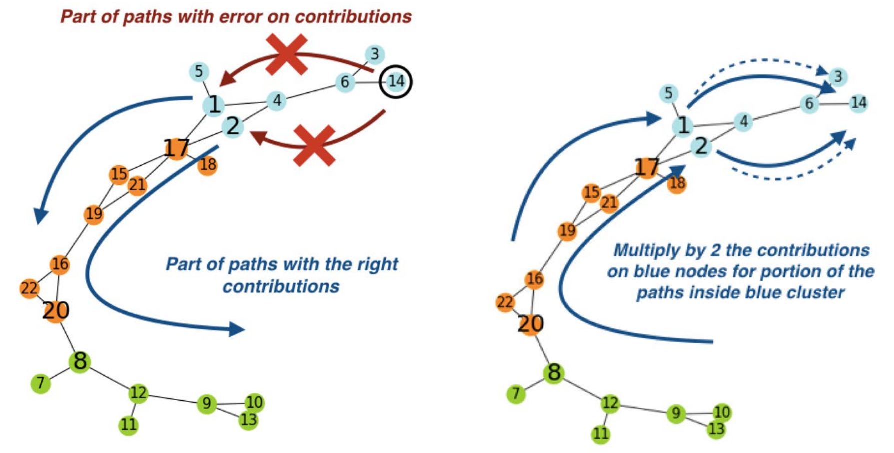
$$\delta^{\gamma}(v) = \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \notin (\mathbf{V}_{\mathbf{C}(v)} \cup \mathbf{V}_{\mathbf{C}(s)})} \delta_{s,t}(v) + 2 \cdot \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \in \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) \tag{11}$$

With this further step, we can now use pivots to efficiently compute the exact global BC. In particular, let $\delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(v)}}(v)$ and $\delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(v)}}}(v)$ 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. 11 can be rewritten as follows:

$$\delta^{\gamma}(v) = \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} [2 \cdot \delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(v)}}(v) + \delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(v)}}}(v))] \tag{12}$$

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

$$\begin{aligned} &\forall v \notin \mathbf{V}_{\mathbf{C}_i}, \mathbf{K}_i \in \mathbf{K}_{\mathbf{C}_i}, \\ &\sum_{s \in \mathbf{K}_i} \left(2 \cdot \delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(v)}}(v) + \delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(v)}}}(v)\right) = |\mathbf{K}_i| \cdot \left(2 \cdot \delta^{\gamma}_{k_i,\mathbf{V}_{\mathbf{C}(v)}}(v) + \delta^{\gamma}_{k_i,\overline{\mathbf{V}_{\mathbf{C}(v)}}}(v)\right) \end{aligned} \tag{13}$$


Part of paths with error on contributions
Part of paths with the right contributions
Multiply by 2 the contributions on blue nodes for portion of the paths inside blue cluster
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22

Problem with nodes belonging to clusters of pivots

Solution for undirected graphs

Fig. 4 Global SSSP explorations from pivots

Equation 13 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.

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[4] of a node v, $\delta^{\gamma}_{s,\cdot}(v)$, in terms of the global dependency scores of w, $\delta^{\gamma}_{s,\cdot}(w)$, 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. 13).

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

---

[4] $\delta^{\gamma}_{s,\cdot}(v)$ is equivalent to $\delta_{s,\overline{\mathbf{V_{C(s)}}}}(v)$

$\delta^{\gamma}_{s,\cdot}(v)$. 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 $\delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(t)}}(v)$. Formally, we have the following recursive formula:

$$\forall \mathbf{C}_i \in \mathbf{C} \setminus \mathbf{C}(s) : \delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}_i}}(v) = \sum_{w : v \in \mathbf{P}_s(w)} \frac{\sigma_{s,v}}{\sigma_{s,w}} * (\nVdash_{w \in \mathbf{C}_i} + \delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}_i}}(w)), \tag{14}$$

where $\nVdash_{w \in \mathbf{C}_i}$ represents a boolean variable equal to 1 if $w \in \mathbf{C}_i$, 0 otherwise[5]. 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:

$$\delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(s)}}}(v) = 2 \cdot \delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(v)}}(v) + \sum_{\mathbf{C}_i \neq \mathbf{C}(v)} \delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}_i}}(v) \tag{15}$$

Finally, according to Eq. 13, $\delta^{\gamma}_{s,\cdot}(v)$ is multiplied by the cardinality of the equivalence class s belongs to.

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 based on MapReduce.

Louvain clustering

To group nodes in clusters and minimize the number of border nodes, $|\mathbf{BN}|$, and consequently $|\mathbf{BN}_{\mathbf{C}_i}|$ for each cluster $\mathbf{C}_i \in \mathbf{C}$, 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 [29] is an example of modularity-based clustering technique. Its time complexity of $O(nlog^2 n)$ 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.

---

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

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.

Algorithm implementation

Algorithm 1 reports the pseudo-code of the *E1C-FastBC* algorithm taking as input an undirected unweighted graph **G** and producing in output the exact values of BC for every node in **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 Algorithm 1.

Algorithm 1 Pseudo-code of the E1C-FastBC algorithm

```
1:  function E1CFastBC($\mathbf{G}$)
2:      $\mathbf{C} \leftarrow modularityBasedClustering(\mathbf{G})$
3:      $\mathbf{BN} \leftarrow findBorderNodes(\mathbf{G}, \mathbf{C})$
4:      $\mathbf{V}_{HSN} \leftarrow buildHSN(\mathbf{BN}, \mathbf{C}, \mathbf{G})$
5:      $\mathbf{EN} \leftarrow findExternalNodes(\mathbf{V}_{HSN}, \mathbf{C}, \mathbf{BN}, \mathbf{G})$
6:      $\mathbf{C}^{*} \leftarrow updateClusters(\mathbf{C}, \mathbf{EN})$
7:      $\delta^{\lambda}, \delta^{\epsilon}, \hat{\sigma}, \hat{d} \leftarrow computeLocal\delta(\mathbf{C}^{*}, \mathbf{C}, \mathbf{EN}, \mathbf{BN}, \mathbf{G})$
8:      $\mathbf{K} \leftarrow findClasses(\hat{\sigma}, \hat{d})$
9:      $\delta^{\gamma} \leftarrow computeGlobal\delta(\mathbf{K}, \mathbf{G}, \mathbf{C})$
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 $\mathbf{BC}$
14: end function
```

At line 2, the Louvain, modularity-based clustering algorithm is exploited for splitting graph **G** into a set of clusters **C** (see section Louvain clustering). 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 **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 Algorithm 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 Algorithm 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

```
function BUILDHSN($\mathbf{BN}, \mathbf{C}, \mathbf{G}$)
    for $s \leftarrow \mathbf{BN}$ do
        $\mathbf{P}_s(\mathbf{V}_{\mathbf{C}(s)}) \leftarrow BFS(s, \mathbf{G}, \mathbf{V}_{\mathbf{C(s)}})$
    end for
    for $s, t \leftarrow \mathbf{BN}$ where $\mathbf{C}(s) == \mathbf{C}(t)$ do
        $\mathbf{V}_{HSN} \leftarrow \mathbf{V}_{HSN} \bigcup crossSPBackwards(t, \mathbf{P}_s(\mathbf{V}_{\mathbf{C}(s)}))$
    end for
    return $\mathbf{V}_{HSN}$
end function
```

At line 5, we identify external nodes **EN** as detailed in Algorithm 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

```
function FINDEXTERNALNODES($\mathbf{V}_{HSN}, \mathbf{C}, \mathbf{BN}, \mathbf{G}$)
    for $s \leftarrow \mathbf{BN}$ do
        $\mathbf{P}_s(\mathbf{V}_{HSN}) \leftarrow BFS(s, \mathbf{G}, \mathbf{V}_{HSN})$
    end for
    for $s, t \leftarrow \mathbf{BN}$ where $\mathbf{C}(s) == \mathbf{C}(t)$ do
        $\mathbf{EN} \leftarrow \mathbf{EN} \bigcup crossSPBackwards(t, \mathbf{P}_s(\mathbf{V}_{HSN}))$
    end for
    return $\mathbf{EN}$
end function
```

The extended clusters $\mathbf{C}^*$ are generated at line 6 of Algorithm 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 (see section "Equivalence class with clustering"). A modified version of Brandes' algorithm enables the computation of all these metrics, as described in Algorithm 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

```
function computeLocalδ($\mathbf{C}^*, \mathbf{C}, \mathbf{EN}, \mathbf{BN}, \mathbf{G}$)
    for $s \leftarrow \mathbf{V}$ do
        $\delta^{\lambda}_{s\bullet}(\mathbf{V}_{\mathbf{C}(s)}), \delta^{\epsilon}_{s\bullet}(\mathbf{EN}_{\mathbf{C}(s)}), \hat{\sigma}_{s,\mathbf{BN}_{\mathbf{C}(s)}}, \hat{d}_{\mathbf{G}}(s, \mathbf{BN}_{\mathbf{C}(s)})) \leftarrow BrandesModifiedV1(s, \mathbf{C}^*(s), \mathbf{C}(s), \mathbf{BN}, \mathbf{G})$
        $\delta^{\lambda}(\mathbf{V}_{\mathbf{C}(s)}) \leftarrow \delta^{\lambda}(\mathbf{V}_{\mathbf{C}(s)}) + \delta^{\lambda}_{s\bullet}(\mathbf{V}_{\mathbf{C}(s)})$
        $\delta^{\epsilon}(\mathbf{EN}_{\mathbf{C}(s)}) \leftarrow \delta^{\epsilon}(\mathbf{EN}_{\mathbf{C}(s)}) + \delta^{\epsilon}_{s\bullet}(\mathbf{EN}_{\mathbf{C}(s)})$
    end for
    return $\delta^{\lambda}, \delta^{\epsilon}, \hat{\sigma}, \hat{d}$
end function
```

At line 9, we compute the global BC, δ^{γ}. As shown in Algorithm 5, a second modified version of Brandes' algorithm, which exploits Eq. 13, Eq. 14 and Eq. 15, 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, $\delta^{\gamma}_{k_i,\overline{\mathbf{V}_{\mathbf{C}(k_i)}}}(\mathbf{V})$ 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 Algorithm 1, all the previously computed partial terms are aggregated via a sum operation to obtain the exact BC values for each node.

Algorithm 5 Pseudo-code of global BC computation

```
function computeGlobalδ($\mathbf{K}, \mathbf{G}, \mathbf{C}$)
    for $K_i \leftarrow \mathbf{K}$ do                                        ▷ For all classes
        $k_i \leftarrow random(K_i)$
        $\delta^{\gamma}_{k_i,\overline{\mathbf{V}_{\mathbf{C}(k_i)}}}(\mathbf{V}) \leftarrow BrandesModifiedV2(k_i, \mathbf{G}, \mathbf{C})$
        $\delta^{\gamma}(\overline{\mathbf{V}_{\mathbf{C}(k_i)}}) \leftarrow \delta^{\gamma}(\overline{\mathbf{V}_{\mathbf{C}(k_i)}}) + \delta^{\gamma}_{k_i,\overline{\mathbf{V}_{\mathbf{C}(k_i)}}}(\overline{\mathbf{V}_{\mathbf{C}(k_i)}}) \cdot |K_i|$
    end for
    return $\delta^{\gamma}$
end function
```

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

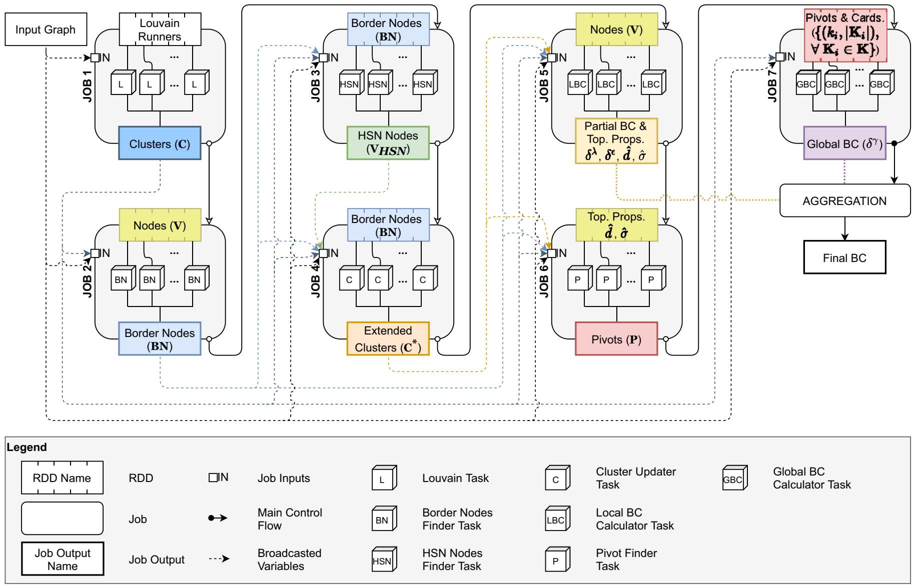


Fig. 5 Map-reduce, Spark-based description of E1C-FastBC. The main execution flow is represented by solid arrows, while dashed ones represent broadcast variables. Jobs and tasks are depicted as rounded rectangles and cubes, respectively. RDDs are shown as non-rounded rectangles

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 of tasks behaviors and needed inputs.

- *Job 1* organizes the graph into clusters (Algorithm 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.

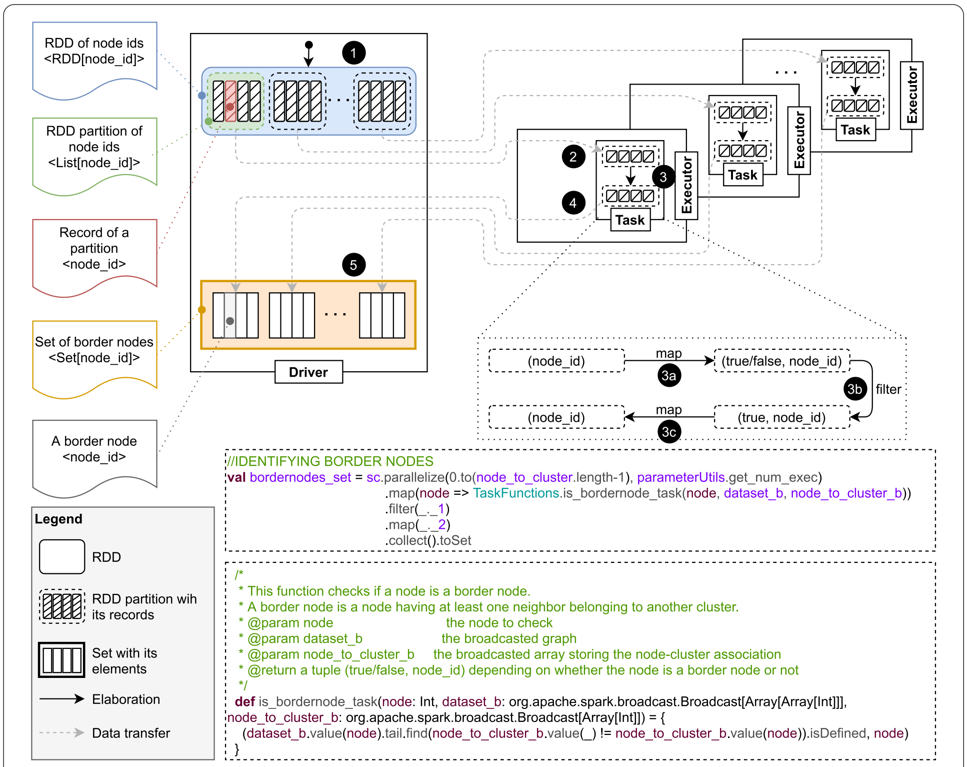
- *Job 2* identifies border nodes (Algorithm 1, line 3), by checking for each node the existence of at least one neighbor belonging to a different cluster. It requires as input the set of clusters and the graph, both passed as broadcast variables. The starting RDD contains all the nodes to analyze, and it is built from the whole set of graph vertices.
- *Job 3* retrieves the HSN nodes (Algorithm 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 (Algorithm 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 (Algorithm 1, line 7). The job receives the graph, the clusters, the extended clusters and the border nodes as inputs, all transferred as broadcast variables[6]. The starting RDD of this job contains all the nodes of the graph.
- *Job 6* identifies the equivalence classes and their pivots (Algorithm 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 (Algorithm 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).

Figure 6 reports a detailed description of Job 2 to exemplify how a job is performed. Solid arrows represent elaboration phases, while the dashed ones represent data transfer phases. The dotted box shows the set of transformations applied by the tasks hosted on the executors over the different partitions. The first dashed box reports the source code related to the job, whereas the second reports the source code of the first map task in the pipeline. The job is triggered by the *collect* action. The driver builds the initial RDD by executing the *parallelize* method (1). The number of partitions is equal to the number of

---

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



```
//IDENTIFYING BORDER NODES
val bordernodes_set = sc.parallelize(0.to(node_to_cluster.length-1), parameterUtils.get_num_exec)
                                      .map(node => TaskFunctions.is_bordernode_task(node, dataset_b, node_to_cluster_b))
                                      .filter(_._1)
                                      .map(_._2)
                                      .collect().toSet
```

```
 /*
  * This function checks if a node is a border node.
  * A border node is a node having at least one neighbor belonging to another cluster.
  * @param node                        the node to check
  * @param dataset_b                   the broadcasted graph
  * @param node_to_cluster_b     the broadcasted array storing the node-cluster association
  * @return a tuple (true/false, node_id) depending on whether the node is a border node or not
  */
 def is_bordernode_task(node: Int, dataset_b: org.apache.spark.broadcast.Broadcast[Array[Array[Int]]],
node_to_cluster_b: org.apache.spark.broadcast.Broadcast[Array[Int]]) = {
   (dataset_b.value(node).tail.find(node_to_cluster_b.value(_) != node_to_cluster_b.value(node)).isDefined, node)
 }
```

Fig. 6 Map-reduce, detailed description of *Job2* Includes elaboration and data transfer phases (solid and dashed arrows), the set of transformations applied by the tasks (dotted box) and the related source code (dashed boxes)

executors, i.e., each executor works on a single partition. Partitions are sent to executors along with the operations to be performed on them (2). These operations are encapsulated in a task. In particular, the first *map* operation (3a) generates an intermediate RDD of key-value pairs where the second element is the unique node identifier and the first element is a boolean value (true/false) that depends on whether the node is a border node or not. Then, the *filter* operation creates a new intermediate RDD containing only the pairs with the key equal to true (3b). Finally, the second *map* operation produces an RDD by extracting only the second element of the previous pairs (3c). Hence, the border nodes are collected on the driver (4) and stored in a set (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 $AS_p^{a_1/a_2}$, 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. Hence, the larger the value of AS, the faster a_1 is compared to a_2 with the same computing resources. For example, $AS = 2$ means that the time taken by a_1 is half the time taken by a_2 and therefore that a_1 is two times faster than a_2.

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) and it addresses the problem of exact BC 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.

Efficiency E_p may influence the *AS* metric as demonstrated by the following analysis. From the definition of speedup, $T_p^{a_2} = T_s^{a_2}/S_p^{a_2}$, where $T_s^{a_2}$ and $S_p^{a_2}$ are the execution time in sequential mode and the speedup obtained with p cores of algorithm a_2. Similarly, we can write $T_p^{a_1} = T_s^{a_1}/S_p^{a_1}$. By using these equations in the definition of the *AS* metric, we have that $AS_p^{a_1/a_2} = (T_s^{a_2}/T_s^{a_1}) \cdot (S_p^{a_1}/S_p^{a_2})$. From the definition of efficiency, $S_p^{a_1}/S_p^{a_2} = E_p^{a_1}/E_p^{a_2}$ and consequently $AS_p^{a_1/a_2} = (T_s^{a_2}/T_s^{a_1}) \cdot (E_p^{a_1}/E_p^{a_2})$.

The relationship between *AS* and E_p suggests that if a_1 and a_2 have comparable efficiency with p cores, then the *AS* only depends on the ratio of the execution times of the two algorithms in sequential mode, thus providing interesting insights to comparatively analyze the two different solutions.

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.

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 Barabási-Albert model provided by the Python library NetworkX. According to that model, a graph of n nodes is grown by attaching new nodes, one at a time, each with m' edges that are preferentially attached to existing nodes with high degree. In our case, m', which is called the preferential attachment coefficient, is 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. However, as mentioned in the introduction, this does not limit the applicability of our solution because many real-world systems can be represented with the Barabási-Albert model. In particular, to analyze the algorithm in terms of performance and scalability, we generate graphs with different sizes (see Table 3).

Table 3 Topological information of synthetic & real graphs

	Graph	*n*	*m*	d_{avg}	d_{max}	cc_{avg}
Synthetic	Barabási-albert	6250	6249	1.999	126	0.000
	Barabási-albert	12,500	12,499	1.999	225	0.000
	Barabási-albert	25,000	24,999	1.999	344	0.000
	Barabási-albert	50,000	49,999	1.999	463	0.000
	Barabási-albert	100,000	99,999	1.999	1138	0.000
	Barabási-albert	200,000	199,999	1.999	676	0.000
	Barabási-albert	400,000	399,999	1.999	1142	0.000
	Barabási-albert	800,000	799,999	1.999	1587	0.000
Real	Web-webbase-2001 [34]	16,062	25,593	3.187	1679	0.224
	Ego-twitter [35]	22,322	31,823	2.851	238	0.072
	Internet [34]	124,651	193,620	3.107	151	0.062
	lyon-road-network[1]	156,102	178,845	2.291	8	0.017
	Email-euAll [36]	224,832	339,925	3.024	7636	0.079

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

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

For the second category, we focus on some real graphs[7] available in public datasets. Table 3 reports all the graphs we use, together with some relevant properties. In particular, for each graph we consider the average degree (d_{avg}), the max degree (d_{max}) and the average clustering coefficient (cc_{avg}).

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

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 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 45 GB of memory, are deployed.

---

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

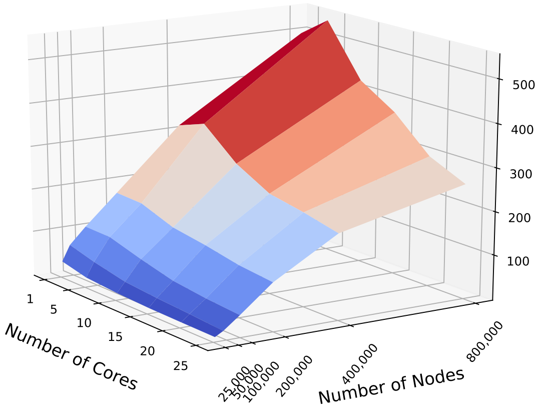


Fig. 7 Comparison with Brandes' algorithm. Algorithmic speedup analysis-$AS_{p=[1,5,10,15,20,25]}^{\mathcal{E}/\mathcal{B}}$

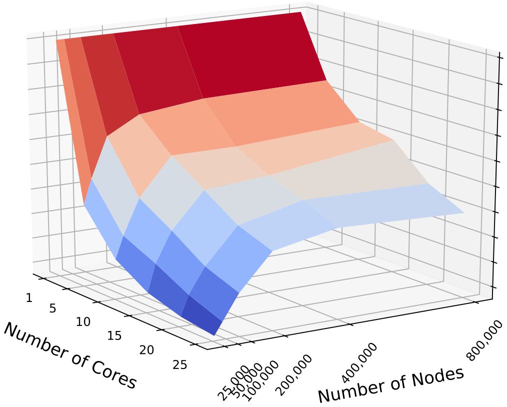
The RDDs are decomposed in a number of partitions equal to the total number of cores.

Synthetic graphs analysis

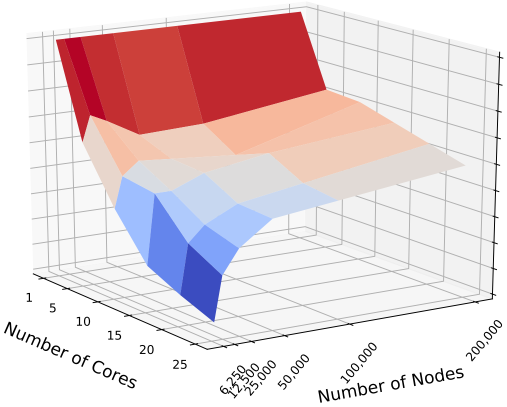
Figure 7 shows the algorithmic speedup of *E1C-FastBC* over Brandes' algorithm, $AS_p^{\mathcal{E}/\mathcal{B}}$, obtained on the synthetic graphs in both sequential and parallel modes. In particular, we double the number of nodes from 25,000 to 800,000, and we consider a number of cores p equals to 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 min and 1.64 h, respectively (sequential mode).

As highlighted by Fig. 7, $AS_p^{\mathcal{E}/\mathcal{B}}$ 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. This behaviour is due to the fact that the Brandes' algorithm is more efficient than *E1C-FastBC* (see Fig. 8). This means that the ratio $E_p^{a_1}/E_p^{a_2}$ in the relationship between the *AS* metric and the efficiency is lower than 1. Consequently, the *AS* value in the sequential case is not preserved 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. 7, 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 8 reports the results of the



E1C-FastBC algorithm



Brandes' algorithm

Fig. 8 Comparison with Brandes' algorithm. Efficiency analysis- $E_{p=[1,5,10,15,20,25]}$

efficiency analysis performed for the two algorithms. In both cases, it is possible to 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. 8. 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)

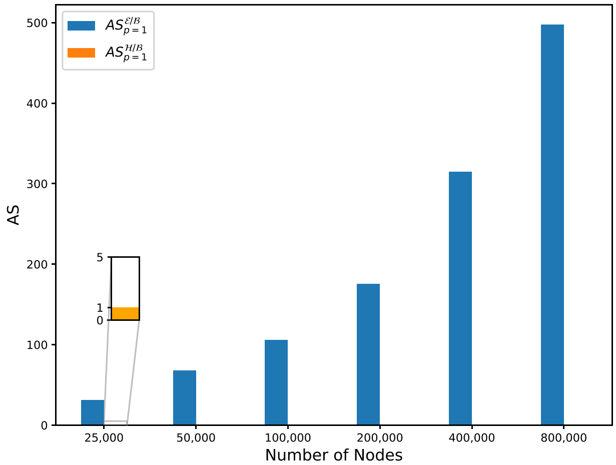


Fig. 9 Comparison with the solution proposed in [13]. Algorithmic speedup analysis with synthetic graphs in sequential mode- $AS_{p=1}^{\mathcal{E}/\mathcal{B}}$ and $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$. $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$ is always equal to 1 for synthetic graphs (see section "Synthetic graphs analysis"). We therefore add only one single zoomed bar in the figure just to make it visible to the reader

Brandes' SSSP explorations performed on the whole graph, which represent the heaviest part of the whole computation (see Figs. 13 and 14), 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. 8b).

Figure 9 reports the algorithmic speedup of *E1C-FastBC* over Brandes' algorithm, $AS_{p=1}^{\mathcal{E}/\mathcal{B}}$, alongside with the algorithmic speedup of the approach in [13] over Brandes, $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$, on synthetic graphs and in sequential settings. $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$ is 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 $AS_{p=1}^{\mathcal{H}/\mathcal{B}}$. This means that the algorithm proposed in [13] is not able to improve that of Brandes. Conversely, ours is able to do it by a large

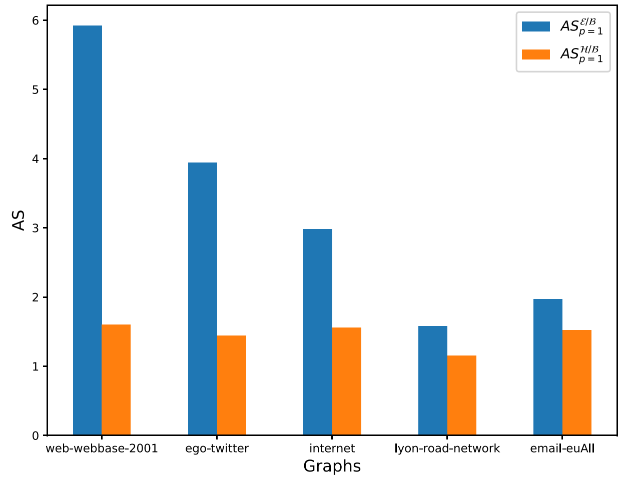


Fig. 10 Comparison with the solution proposed in [13]. Algorithmic speedup analysis with real graphs in sequential mode-$AS^{\mathcal{E}/\mathcal{B}}_{p=1}$ and $AS^{\mathcal{H}/\mathcal{B}}_{p=1}$

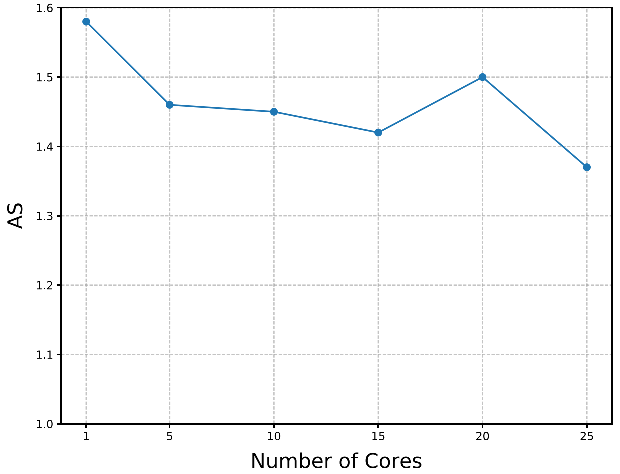


Fig. 11 Algorithmic speedup analysis for lyon-road-network graph in parallel mode-$AS^{\mathcal{E}/\mathcal{B}}_{p=[1,5,10,15,20,25]}$

multiplicative factor. We can thus conclude that our solution always outperforms the one in [13] with synthetic graphs.

Real graphs analysis

Figure 10 reports the results of the analysis of $AS^{\mathcal{E}/\mathcal{B}}_{p=1}$ and $AS^{\mathcal{H}/\mathcal{B}}_{p=1}$ carried out on real graphs. $AS^{\mathcal{H}/\mathcal{B}}_{p=1}$ 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

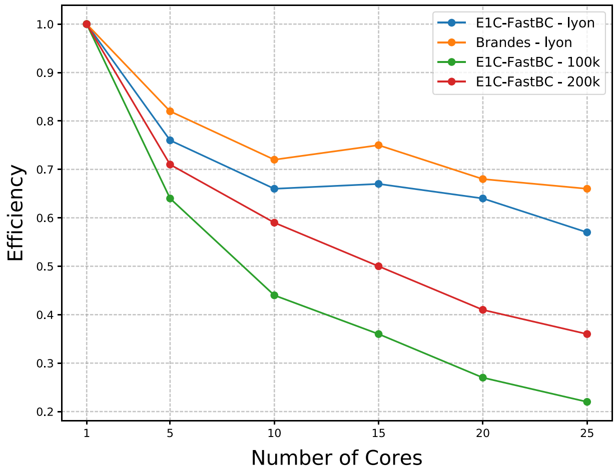


Fig. 12 Efficiency analysis for lyon-road-network graph-$E_{p=[1,5,10,15,20,25]}$. Comparison with synthetic graphs having similar size (100k and 200k nodes) and with Brandes' algorithm on the same graph

algorithmic speedup factor. As shown in Fig. 11, 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. 7) 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. 12). Also in this case, by considering the two figures (Figs. 11 and 12), it is possible to note the dependency relationship between the *AS* metric and the efficiency. In particular, going from 15 to 20 cores the difference between efficiency values for the Brandes' algorithm and of *E1C-FastBC* decreases while *AS* increases.

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 Algorithm 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 13 and 14 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

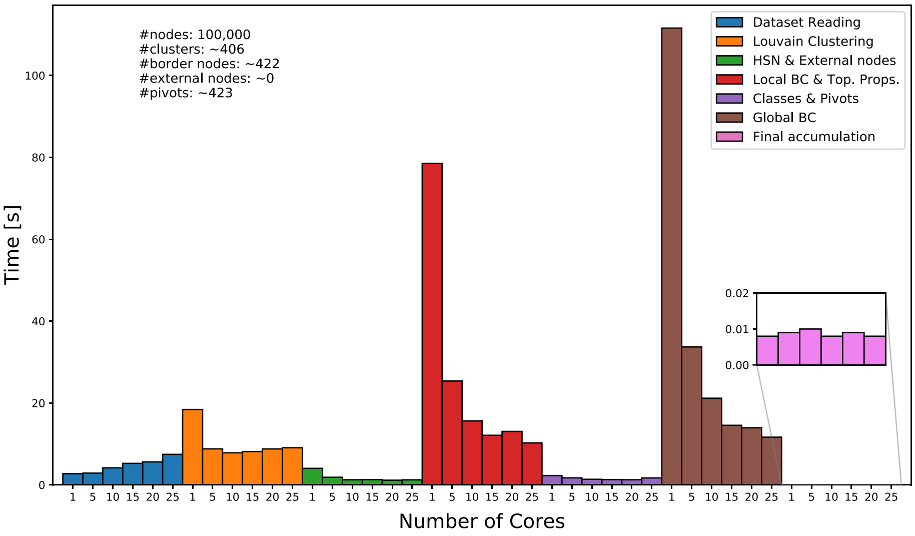


Fig. 13 Breakdown analysis of computation time on synthetic graph with 100,000 nodes

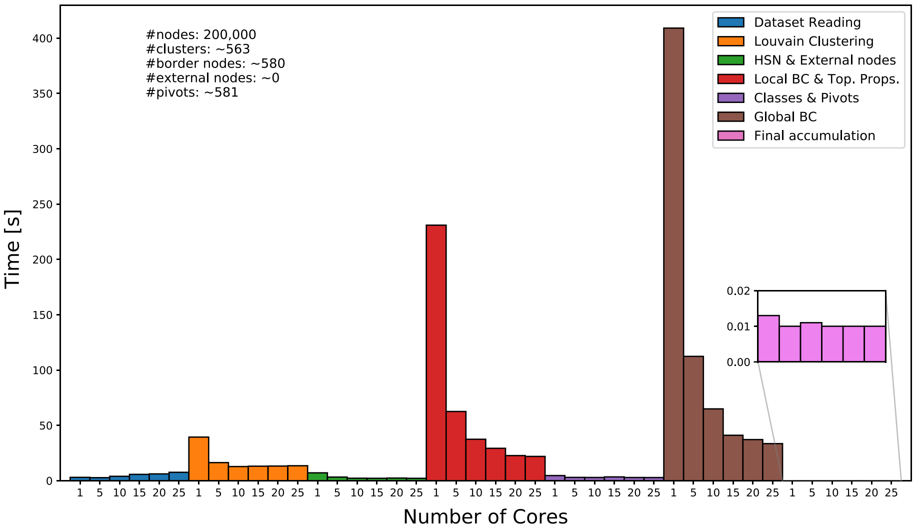


Fig. 14 Breakdown analysis of computation time on synthetic graph with 200,000 nodes

configurations with the aim of selecting the one that produces the clustering with the best modularity score. This aspect highlights a potential bottleneck, since 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 section "Synthetic graphs analysis", 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.

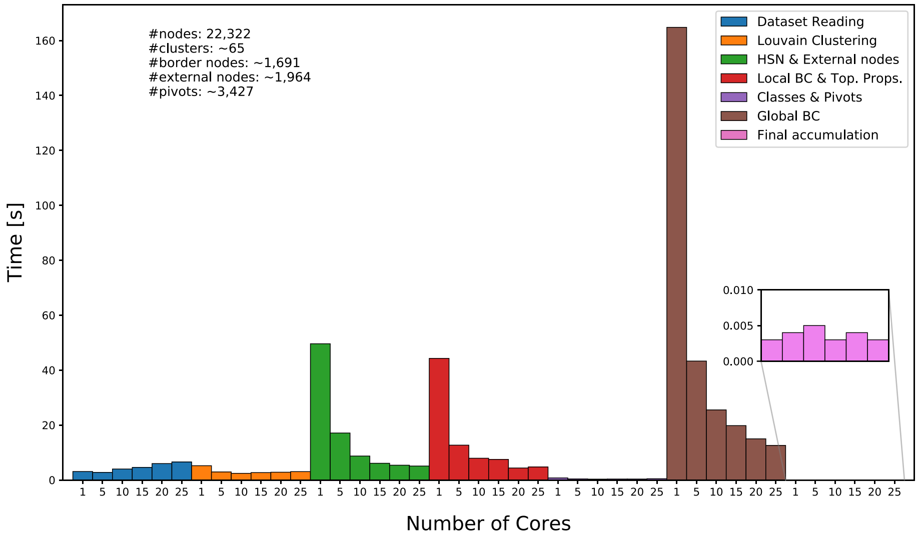


Fig. 15 Breakdown analysis of computation time on real graph ego-twitter

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 15 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.

Mathematical foundations

As discussed in "Clustering and BC computation" section, our algorithm relies on multiple mathematical properties to allow for the exact fast computation of BC. In the following, we provide the mathematical proofs of the mathematical foundation (theorem, corollary and claim) at the basis our algorithm. In theorem 6.1, we prove that two nodes of the same cluster that satisfy some properties produce the same dependency score on nodes outside the cluster for destinations that are outside the cluster.

---

[8] We do not use lyon-road-network graph as in section Real graphs analysis because of the Out of Memory error that arise when running the algorithm in profiling mode.

Theorem 6.1 *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 $\mathbf{BN}_{\mathbf{C}_i}$ and $s, p \in \mathbf{V}_{\mathbf{C}_i}$. 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:

$$\begin{aligned}&\forall v \in \overline{\mathbf{V}_{\mathbf{C}_i}},\\ &\delta_{s,\overline{\mathbf{V}_{\mathbf{C}_i}}}(v) = \delta_{p,\overline{\mathbf{V}_{\mathbf{C}_i}}}(v) \iff \sum_{t \notin \mathbf{V}_{\mathbf{C}_i}} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} = \sum_{t \notin \mathbf{V}_{\mathbf{C}_i}} \frac{\sigma_{p,t}(v)}{\sigma_{p,t}}\end{aligned} \tag{16}$$

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

$$\sigma_{s,t}(v) = k \cdot \sigma_{p,t}(v) \;\; \forall v, t \in \overline{\mathbf{V}_{\mathbf{C}_i}} \tag{17}$$

and

$$\sigma_{s,t} = k \cdot \sigma_{p,t} \;\; \forall t \in \overline{\mathbf{V}_{\mathbf{C}_i}} \tag{18}$$

hold under the hypotheses of the theorem.

Let us first prove the following Lemma 6.1, 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 6.1 *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 $d_{\mathbf{G}}(s, b_j) = d_{\mathbf{G}}(p, b_j) + l \;\; \forall\, b_j \in \mathbf{BN}_{\mathbf{C}_i}$ 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:

$$d_{\mathbf{G}}(s, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(s, t) \tag{19}$$

Given that $b_j \in \mathbf{BN}_{\mathbf{C}_i}(s, t)$ by hypothesis, Eq. 19 can be easily re-written as follows:

$$\begin{aligned}&d_{\mathbf{G}}(s, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(s, t)\\ \iff &d_{\mathbf{G}}(s, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(s, b_j) + d_{\mathbf{G}}(b_j, t)\end{aligned} \tag{20}$$

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:

$$\begin{aligned}&d_{\mathbf{G}}(s, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(s, t)\\ \iff &d_{\mathbf{G}}(p, b_k) + l + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(p, b_j) + l + d_{\mathbf{G}}(b_j, t)\\ \iff &d_{\mathbf{G}}(p, b_k) + d_{\mathbf{G}}(b_k, t) > d_{\mathbf{G}}(p, b_j) + d_{\mathbf{G}}(b_j, t)\end{aligned} \tag{21}$$

From Eq. 21, 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:

$$\mathbf{BN}_{\mathbf{C}_i}(p,t)\subseteq \mathbf{BN}_{\mathbf{C}_i}(s,t) \tag{22}$$

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:

$$d_{\mathbf{G}}(p,b_k)+d_{\mathbf{G}}(b_k,t)>d_{\mathbf{G}}(p,t)\iff \mathbf{BN}_{\mathbf{C}_i}(s,t)\subseteq \mathbf{BN}_{\mathbf{C}_i}(p,t) \tag{23}$$

Therefore, from Eq. 22 and Eq. 23, we can conclude that the following relationship holds:

$$\begin{aligned}&\mathbf{BN}_{\mathbf{C}_i}(s,t)\subseteq \mathbf{BN}_{\mathbf{C}_i}(p,t)\text{ AND }\ \mathbf{BN}_{\mathbf{C}_i}(p,t)\subseteq \mathbf{BN}_{\mathbf{C}_i}(s,t)\\ \iff& \mathbf{BN}_{\mathbf{C}_i}(s,t)=\mathbf{BN}_{\mathbf{C}_i}(p,t)\end{aligned} \tag{24}$$

which proves the lemma. □

To complete the proof of Theorem 6.1, we need now to prove Eq. 17 and Eq. 18. To that purpose, we consider the following lemma.

Lemma 6.2 *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:

$$\sigma_{s,t}=\sum_{b_j\in\mathbf{BN}_{\mathbf{C}_i}(s,t)}\sigma_{s,b_j}\cdot\sigma_{b_j,t}. \tag{25}$$

From the hypothesis of Theorem 6.1, 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. 25 becomes:

$$\sigma_{s,t}=\sum_{b_j\in\mathbf{BN}_{\mathbf{C}_i(s,t)}}k\cdot\sigma_{p,b_j}\cdot\sigma_{b_j,t} \tag{26}$$

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:

$$\begin{aligned}\sum_{b_j\in\mathbf{BN}_{\mathbf{C}_i(s,t)}}k\cdot\sigma_{p,b_j}\cdot\sigma_{b_j,t}&=k\cdot\sum_{b_j\in\mathbf{BN}_{\mathbf{C}_i}(p,t)}\sigma_{p,b_j}\cdot\sigma_{b_j,t}\\&=k\cdot\sigma_{p,t}\end{aligned} \tag{27}$$

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

$$\sigma_{s,t}(v)=k\cdot\sigma_{p,t}(v) \tag{28}$$

□

Eq. 28 and Eq. 27 from Lemma 6.2 prove, via Lemma 6.1, Eq. 17 and Eq. 18 respectively. Therefore Theorem 6.1 is proved. □

We now prove that the normalized distances and number of shortest paths fulfill the conditions of Theorem 6.1 and hence can be used to group nodes into classes of equivalence.

Corollary 6.1 *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* $\delta_{s,\overline{\mathbf{V}_{\mathbf{C}_i}}}(v) = \delta_{p,\overline{\mathbf{V}_{\mathbf{C}_i}}}(v),\ \forall v \in \overline{\mathbf{V}_{\mathbf{C}_i}}$

.

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

$$\hat{d}_{\mathbf{G}}(s,b_j) = \hat{d}_{\mathbf{G}}(p,b_j)\ \ \forall b_j \in \mathbf{BN}_{\mathbf{C}_i} \Longrightarrow d_{\mathbf{G}}(s,b_j) = d_{\mathbf{G}}(p,b_j) + l\ \ \forall b_j \in \mathbf{BN}_{\mathbf{C}_i} \tag{29}$$

and:

$$\hat{\sigma}_{s,b_j} = \hat{\sigma}_{p,b_j}\ \ \forall b_j \in \mathbf{BN}_{\mathbf{C}_i} \Longrightarrow \sigma_{s,b_j} = k \cdot \sigma_{p,b_j}\ \ \forall b_j \in \mathbf{BN}_{\mathbf{C}_i} \tag{30}$$

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

$$\begin{aligned} &\forall b_j \in \mathbf{BN}_{\mathbf{C}_i}, \\ &d_{\mathbf{G}}(s,b_j) - min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} d_{\mathbf{G}}(s,b_k) = d_{\mathbf{G}}(p,b_j) - min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} d_{\mathbf{G}}(p,b_k) \end{aligned} \tag{31}$$

$$\begin{aligned} &\text{AND} \\ &\frac{\sigma_{s,b_j}}{min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} \sigma_{s,b_k}} = \frac{\sigma_{p,b_j}}{min_{b_k \in \mathbf{BN}_{\mathbf{C}_i}} \sigma_{p,b_k}} \end{aligned} \tag{32}$$

By definition, Eq. 31 can be easily re-written as follows:

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

which corresponds to Eq. 29.

Likewise, Eq. 32 can be re-written as:

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

which corresponds to Eq. 30. As the two equations (Eq. 29 and Eq. 30) are jointly satisfied, the corollary is proved from Theorem 6.1. □

Claim 6.1 *In undirected graphs*:

$$
\sum_{s \in \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \notin \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) = \sum_{s \notin \mathbf{V}_{\mathbf{C}(v)}} \sum_{t \in \mathbf{V}_{\mathbf{C}(v)}} \delta_{s,t}(v) \tag{33}
$$

The proof of the claim above, used in section "Dependency score of pivots", is entirely based on the property of undirection. We remove part of the estimation errors we had in the previous implementations ([12]) by changing the computation of global dependency score using this claim.

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:

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

□

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 one labelled with $\mathcal{H}$, 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 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 E1C-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.

Appendix

In our work, we implemented two alternative approaches to back-propagate contributions during the computation of the global BC. The first one, detailed in section "Cluster-based exact BC computation", is based on using one vector of contributions per node, with one entry per cluster of destinations. The main limitation of this approach is related to high memory consumption as it implies working with n vectors ($n =$ number of nodes), each including $|\mathbf{C}|$ contributions. To optimise our solution with respect to this limitation, we developed another approach based on a vector of two elements only per node. This method is described below.

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 $\delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(v)}}(v)$ or $\delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(v)}}}(v)$ 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:

$$\begin{aligned}
\delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(v)}}(v) &= \begin{cases} \frac{\sigma_{s,v}}{\sigma_{s,w}} \cdot (1 + \delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(w)}}(w)) & \text{if } \mathbf{C}(v) = \mathbf{C}(w), \\ 0 & \text{otherwise} \end{cases} \\
\delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(v)}}}(v) &= \begin{cases} \frac{\sigma_{s,v}}{\sigma_{s,w}} \cdot \delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(w)}}}(w) & \text{if } \mathbf{C}(v) = \mathbf{C}(w), \\ \frac{\sigma_{s,v}}{\sigma_{s,w}} \cdot (1 + \delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(w)}}(w) + \delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(w)}}}(w)) & \text{otherwise} \end{cases}
\end{aligned} \tag{34}$$

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:

$$\delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(s)}}}(v) = 2 \cdot \delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(v)}}(v) + \delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(v)}}}(v) \tag{35}$$

Finally, according to Eq. 13, $\delta^{\gamma}_{s,\cdot}(v)$ is multiplied by the cardinality of the equivalence class s belongs to. However, this solution introduces an error. In fact, in Eq. 34, $\delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(w)}}}(w)$ may contain contributions of destination nodes belonging to $\mathbf{V}_{\mathbf{C}(v)}$, which should be moved from $\delta^{\gamma}_{s,\overline{\mathbf{V}_{\mathbf{C}(v)}}}(v)$ to $\delta^{\gamma}_{s,\mathbf{V}_{\mathbf{C}(v)}}(v)$, so that they can be correctly multiplied by 2 as required by Eq. 35. 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 $\sigma^{ext}_{b_1,b_2}$ between them. We perform this operation when searching for external nodes.

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.

Acknowledgements

This work has been carried out in the framework of the French ANR PROMENADE research project and of the CINI (Consorzio Interuniversitario Nazionale per l'Informatica) Smart Cities and Communities National Lab.

Authors' contributions

CD contributed to the definition and implementation of the algorithms, data processing, experiments, results analysis, validation, and writing of the draft of the paper. LG contributed to the refinement of the algorithms, data processing, experiments, results analysis, validation, and writing of the draft of the paper. AF 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. EZ supervised the design and implementation of the work, proposed the algorithms and the techniques presented in the paper and elaborated them in collaboration with AF. AF and EZ contributed to results analysis, validation, and review & editing of the paper. AF and EZ contributed to the funding acquisition. All the authors have approved the final version of this paper submitted to publication. All authors read and approved the final manuscript.

Funding

This work has been funded 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 source code is available at https://github.com/licit-lab/e1c_fast_bc and https://github.com/UniSannio-DiStLab/E1C-FastBC. The repositories also contain the datasets related to synthetic and real graphs used in the current study. In particular, the real graphs, except lyon-road-network, have been retrieved from 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/.

Declarations

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Received: 11 March 2021 Accepted: 8 June 2021

References

1. Freeman LC. A set of measures of centrality based on betweenness. Sociometry. 1997;25:35–41.
2. Borgatti SP, Mehra A, Brass DJ, Labianca G. Network analysis in the social sciences. Science. 2009;323(5916):892–5.
3. King D, Shalaby A. Performance metrics and analysis of transit network resilience in Toronto. Transp Res Rec. 2016.
4. Zhang Y, Wang X, Zeng P, Chen X. Centrality characteristics of road network patterns of traffic analysis zones. Transp Res Rec. 2011;2256:16–24.
5. Berche B, von Ferber C, Holovatch T, Holovatch Y. Resilience of public transport networks against attacks. Eur Phys J B. 2009;71(1):125–37.
6. 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) 2017. p. 1289–94. https://doi.org/10.1109/BigData.2017.8258057.
7. Furno A, Faouzi NE, 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. 2018. p. 234–41. https://doi.org/10.1109/SMARTCOMP.2018.00096.
8. Holme P, Kim BJ, Yoon CN, Han SK. Attack vulnerability of complex networks. Phys Rev E. 2002;65(5):056109.
9. Carpenter T, Karakostas G, Shallcross D. Practical issues and algorithms for analyzing terrorist networks. Proceedings of the Western Simulation Multi Conference. 2002.
10. Floyd RW. Algorithm 97: Shortest path. Commun ACM. 1962;5(6):345. https://doi.org/10.1145/367766.368168.
11. Brandes U. A faster algorithm for betweenness centrality. J Math Soc. 2001;2:163.
12. Suppa P, Zimeo E. A clustered approach for fast computation of betweenness centrality in social networks. In: 2015 IEEE International Congress on Big Data. 2015. p. 47–54. https://doi.org/10.1109/BigDataCongress.2015.17.
13. Li Y, Li W, Tan Y, Liu F, Cao Y. Lee KY Hierarchical decomposition for betweenness centrality measure of complex networks. Sci Rep. 2017;7:1–2.
14. Barabási A-L, Albert R. Emergence of scaling in random networks. Science. 1999;286(5439):509–12. https://doi.org/10.1126/science.286.5439.509.
15. Bader DA, Madduri K. Parallel algorithms for evaluating centrality indices in real-world networks. In: International Conference On Parallel Processing, 2006. ICPP 2006. 2006. p. 539–50. https://doi.org/10.1109/ICPP.2006.57.
16. Madduri K, Ediger D, Jiang K, Bader DA, Chavarria-Miranda D. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In: IEEE International Symposium On Parallel & Distributed Processing, 2009. IPDPS 2009. 2009. p. 1–8. https://doi.org/10.1109/IPDPS.2009.5161100.
17. van der Grinten A, Meyerhenke H. Scaling betweenness approximation to billions of edges by mpi-based adaptive sampling. 2020. p. 527–35. https://doi.org/10.1109/IPDPS47924.2020.00061.
18. Borassi M, Natale E. Kadabra is an adaptive algorithm for betweenness via random approximation. J Exp Algorithm. 2016. https://doi.org/10.1145/3284359.
19. Shi Z, Zhang B. Fast network centrality analysis using GPUs. BMC Bioinform. 2011;12(1):149. https://doi.org/10.1186/1471-2105-12-149.
20. Kourtellis N, Morales GDF, Bonchi F. Scalable online betweenness centrality in evolving graphs. 2016 IEEE 32nd International Conference on Data Engineering (ICDE) 2016. p. 1580–1. https://doi.org/10.1109/ICDE.2016.7498421.
21. Behera R, Naik D, Ramesh D, Rath S. Mr-ibc: Mapreduce-based incremental betweenness centrality in large-scale complex networks. Soc Netw Anal Mining. 2020. https://doi.org/10.1007/s13278-020-00636-9.
22. Shukla K, Regunta S, Harsh S, Kothapalli K. Efficient parallel algorithms for betweenness- and closeness-centrality in dynamic graphs. 2020. p. 1–12. https://doi.org/10.1145/3392717.3392743.
23. Brandes U, Pich C. Centrality estimation in large networks. Int J Bifurcat Chaos. 2007;17(07):2303–18.
24. Geisberger R, Sanders P, Schultes D. Better approximation of betweenness centrality. In: ALENEX. 2008. p. 90–100. SIAM. https://doi.org/10.1137/1.9781611972887.9.
25. Riondato M, Upfal E. Abra: Approximating betweenness centrality in static and dynamic graphs with rademacher averages 2018; https://doi.org/10.1145/3208351
26. Puzis R, Elovici Y, Zilberman P, Dolev S, Brandes U. Topology manipulations for speeding betweenness centrality computation. J Complex Netw. 2014;3(1):84–112. https://doi.org/10.1093/comnet/cnu015.
27. Sariyüce AE, Kaya K, Saule E, Çatalyürek UV. Graph manipulations for fast centrality computation. ACM Trans Knowl Discov Data (TKDD). 2017;11(3):26.
28. 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. 2012. p. 450–6. https://doi.org/10.1109/ASONAM.2012.79.
29. Blondel VD, Guillaume J-L, Lambiotte R, Lefebvre E. Fast unfolding of communities in large networks. J Stat Mech. 2008;2008(10):10008. https://doi.org/10.1088/1742-5468/2008/10/p10008.
30. Erdős D, Ishakian V, Bestavros A, Terzi E. A divide-and-conquer algorithm for betweenness centrality. 2014. https://doi.org/10.1137/1.9781611974010.49.
31. Maurya SK, Liu X. Murata T Graph neural networks for fast node ranking approximation. ACM Trans Knowl Discov Data. 2021. https://doi.org/10.1145/3446217.
32. Fan C, Zeng L, Ding Y, Chen M, Sun Y, Liu Z. Learning to identify high betweenness centrality nodes from scratch: a novel graph neural network approach. CIKM '19. Association for Computing Machinery, New York, NY, USA. 2019. p. 559–68. https://doi.org/10.1145/3357384.3357979.

33. 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). 2019. p. 603–8. https://doi.org/10.1109/BigData47090.2019.9006576.
34. Rossi RA, Ahmed NK. The network data repository with interactive graph analytics and visualization. In: AAAI (2015). http://networkrepository.com.
35. Mcauley J, Leskovec J. Learning to discover social circles in ego networks. NIPS. 2012;1:539–47.
36. Leskovec J, Kleinberg J, Faloutsos C. Graph evolution: densification and shrinking diameters. ACM Trans Knowledge Discov Data. 2006;1:2

Publisher's Note