Spanning Edge Centrality: Large-scale Computation and Applications

Charalampos Mavroforakis
Boston University
cmav@cs.bu.edu

Richard Garcia-Lebron
University of Texas
San Antonio
richard.garcialebron@utsa.edu

Ioannis Koutis
University of Puerto Rico
Rio Piedras
ioannis.koutis@upr.edu

Evimaria Terzi
Boston University
evimaria@cs.bu.edu

ABSTRACT

The spanning centrality of an edge e in an undirected graph G is the fraction of the spanning trees of G that contain e. Despite its appealing definition and apparent value in certain applications in computational biology, spanning centrality hasn't so far received a wider attention as a measure of edge centrality. We may partially attribute this to the perceived complexity of computing it, which appears to be prohibitive for very large networks. Contrary to this intuition, spanning centrality can in fact be approximated arbitrary well by very efficient near-linear time algorithms due to Spielman and Srivastava, combined with progress in linear system solvers. In this article we bring theory into practice, with careful and optimized implementations that allow the **fast computation** of spanning centrality in very large graphs with millions of nodes. With this computational tool in our disposition, we demonstrate experimentally that spanning centrality is in fact a **useful tool** for the analysis of large networks. Specifically, we show that, relative to common centrality measures, spanning centrality is more effective in identifying edges whose removal causes a higher disruption in an information propagation procedure, while being very resilient to noise, in terms of both the edges scores and the resulting edge ranking.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data mining*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms, Trees*

Keywords

edge centrality; spanning trees; graph algorithms; social networks

WWW 2015, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3469-3/15/05.
http://dx.doi.org/10.1145/2736277.2741125 .

1. INTRODUCTION

Measures of edge centrality are usually defined on the basis of some assumption about how information propagates or how traffic flows in a network. For example, the betweenness centrality of an edge is defined as the fraction of shortest paths that contain it; the underlying assumption being that information or traffic travels in shortest paths [7]. Although more complicated measures of centrality are conceivable, betweenness centrality is simple *by design*: its goal is to yield a computable measure of importance, which can quickly provide valuable information about the network.

Operating under the requirement for simplicity, all edge-importance measures are subject to weaknesses. Betweenness centrality is no exception, having partially motivated a number of other measures (Section 2). It's clear for example that information doesn't always prefer shortest paths; we have all experienced situations when it makes sense to explore slightly *longer* road paths in the presence of traffic. However, it is not clear how to modify betweenness to accommodate such *randomness*. At the same time, betweenness centrality can be *unstable*; the addition of even one 'shortcut' link can dramatically change the scores of edges in the network [27]. Yet, betweenness centrality is at its core very sensible: information may not always take shortest paths, but it *rarely* takes much longer paths.

These considerations lead us to focus on a *simple* and natural alternative model, where information propagates along paths on *randomly* selected *spanning trees*. The idea can actually be viewed as a relaxation of the shortest-paths propagation model: information is 'allowed' to randomly explore longer paths, which however contribute less in the importance measure, because the associated spanning trees are less frequent, as (in some sense) is reflected by the NP-hardness of finding long paths. A number of findings lend support to this intuition: In social networks, information propagates following tree-shaped cascades [15, 26]. Similarly, in computer networks, packages are distributed in the network through tree-shaped structures [18, 28].

We thus define the spanning centrality of an edge e as the fraction of the spanning trees of the graph that contain e. Spanning centrality as a measure for evaluating the significance of edges was introduced in the network analysis literature by Texeira *et al.* [33] in the context of evaluating phylogenetic trees. Computing spanning centrality, by definition involves counting spanning trees, a task that can

be carried out in polynomial time using Kirchhoff's classical matrix-tree theorem [29]. Using this observation, Texeira *et al.* described an exact algorithm for computing the spanning centrality of all m edges in an n-node graph in $O(mn^{3/2})$ time. Despite its appealing definition, spanning centrality hasn't so far received wider attention as a measure of edge importance. This may be partially because, even with the clever observation in [33], the required computation time appears to be prohibitive for most networks of interest.

In this work we remove the aforementioned computational obstacle. We describe a **fast implementation** of an algorithm for spanning centrality that requires $O(m \log^2 n)$ time, or even less in practice (Section 5). The algorithm is *randomized* and it computes *approximations* to spanning centralities, but with strict theoretical guarantees. In practice, for a network consisting of 1.5 million nodes, we can compute spanning centrality values that are within 5% of the exact ones in 30 minutes. The algorithm is based on the fact that the spanning centrality of an edge is equal to its *effective resistance* when the graph is construed as an electrical resistive network. The core component of our implementation is a fast linear system solver for Laplacian matrices [25]. The computation of spanning centrality is also crucially based on the remarkable work of Spielman and Srivastava [31]. Leveraging these two existing algorithmic tools is however not sufficient: our ability to experiment with large-scale networks also relies on a set of **computational speedups**, which include parallelization, exploitation of the input graph structure, and space-efficient implementations. Incidentally, our implementations allow the faster computation of a larger class of *electrical* centralities (Section 6).

With this computational tool in our disposition we embark in the first experimental analysis of spanning centrality as a measure of edge importance, including comparisons with a number of previously proposed centrality measures (Section 7). Our experimental evaluation demonstrates the **practical utility** of spanning centrality for analyzing very large graphs stemming from different application domains. More specifically, we demonstrate its *resilience* to noise, i.e. additions and deletions of edges. Our experiments illustrate that spanning centrality is significantly more resilient than other edge-importance measures, in terms of both the edges scores and the the resulting edge ranking. Thus, the edges with high spanning centrality scores are robust to noisy graphs or graphs that change over time. Further, we investigate the ability of spanning centrality to capture edge-importance with respect to more realistic *information-propagation processes* that don't readily yield computable measures. Our experiments show that removing edges with high spanning centrality incurs significant disruptions in the underlying information-propagation process, more so than other edge-importance measures. This suggests that an effective and computationally efficient way for disrupting the propagation of an item in a network is cutting the links with high spanning centrality.

2. RELATED WORK

In the graph-mining literature, there exists a plethora of measures for quantifying the importance of network nodes or edges [1, 4, 6, 7, 8, 10, 12, 13, 19, 21, 27, 30, 33]. Here, we limit our review to edge-importance measures.

Betweeness centrality remains very popular, and its simplicity can lead to relatively fast implementations despite its quadratic running time. As a consequence, a lot of work has been devoted in its fast computation. The simplest approach leads to an $O(nm)$ time algorithm for unweighted graphs ($O(nm+n^2 \log n)$ if the graph is weighted), where n (resp. m) is the number of nodes (resp. edges) in the graph [7]. The main bottleneck of that computation lies in finding the all-pairs shortest paths. Existing algorithms for speeding up this computation rely on reducing the number of such shortest-path computations. For example, Brandes and Pich [9] propose sampling pairs of source-destination pairs. Then, they experimentally evaluate the accuracy of different source-destination sampling schemes, including random sampling. Geisberger *et al.* [14] also propose sampling source–destination pairs. The only difference is that, in their case, the contribution of every sampled pair to the centrality of a node depends on the distance of that node from the nodes in the selected pair. Instead of sampling random source-destination pairs, Bader *et al.* [2] sample only sources from which they form a DFS traversal of the input graph. Therefore, the shortest-paths from the selected source to all other nodes are retrieved. The key of their method is that the sampling of such sources is adaptive, based on the exploration (through DFS trees) that has already been made. The trade-off between the speedups and the accuracy in the resulting methods is clear as these methods do not provide any approximation guarantees.

Current-flow centrality is another edge-centrality measure proposed by Brandes and Fleischer [8]. Current-flow assigns high scores to edges that participate in many short paths connecting pairs of nodes. We show that both spanning and current-flow centralities belong in the same class of *electrical* centrality measures and we describe a speedup of the original algorithm (proposed by Brandes and Fleischer). In a more recent work, De Meo *et al.* [10] propose *k-path centrality* as a faster-to-compute alternative to current-flow centrality. This centrality counts the number of times an edge is visited by simple random walks of length at most k starting from every node in the network. We note that on their largest reported dataset consisting of 1.1 million nodes and 4.9 million edges their algorithm requires about 6 hours (for very small values of k). We can deal with very similar datasets in less than 1 hour.

3. PRELIMINARIES

We will assume that the input consists of a connected and undirected graph $G = (V, E)$ with n nodes (i.e., $|V| = n$) and m edges (i.e., $|E| = m$). When we deal with matrices, we will be using MATLAB notation. That is, for matrix X, we will use $X(i,:)$ (resp. $X(:,j)$) to refer to the i-th row (resp. j-th column) of X.

Graphs as electrical networks: Throughout the paper, we will view the input graph as a resistive network, i.e., an electrical circuit where every edge is a resistor with fixed (e.g., unit) resistance. By attaching the poles of a battery to different nodes in the network, we will seek computational methods for evaluating the current that passes through the different edges.

The Graph Laplacian matrix: Given a graph $G = (V, E)$, the Laplacian of G is an $n \times n$ matrix L such that, if $\deg(v)$ is the degree of node v in the graph G, then $L(i,i) = \deg(i)$ for every i and $L(i,j) = -1$ if $(i,j) \in E$; otherwise $L(i,j) = 0$.

The incidence matrix: Given graph $G = (V, E)$, we define the *edge-incidence matrix* B of G to be an $m \times n$ matrix such that each row of B corresponds to an edge in E and each column of B corresponds to a node in V. The entries $B(e, v)$ for $e \in E$ and $v \in V$ take values in $\{-1, 0, 1\}$ as follows: $B(e, v) = 1$ if v is the destination of edge e, $B(e, v) = -1$ if v is the origin of e and $B(e, v) = 0$ otherwise. For undirected graphs, the direction of each edge is specified arbitrarily.

Fast linear solvers: Our methods rely on the Combinatorial Multigrid (CMG) solver [25]. CMG is based on a set of *combinatorial preconditioning* methods that have yielded provably very fast linear system solvers for Laplacian matrices and the more general class of symmetric diagonally dominant (SDD) matrices [23, 24]. SDD systems are of the form $Ax = b$ where A is an $n \times n$ matrix that is symmetric and diagonally dominant: i.e., for every i, $A(i, i) \geq \sum_{j \neq i} |A(i, j)|$. For such systems, the solver finds a solution $\overline{x}$ such that $\|\overline{x} - x\|_A = \epsilon \|x\|_A$, where $\| \cdot \|_A$ is the A-norm of a vector, i.e., $\|x\| = \sqrt{x^T Ax}$. If m is the number of non-zero entries of the system matrix A, the theoretically guaranteed solvers run in $O(m \log n \log(1/\epsilon))$ time, but the CMG solver has an even better empirical running time of $O(m \log(1/\epsilon))$.

4. SPANNING CENTRALITY

The spanning centrality of an edge assigns to the edge an importance score based on the number of spanning trees the edge participates in. That is, important edges participate in many spanning trees. Formally, the measure has been defined recently by Teixeira *et al.* [33] as follows:

DEFINITION 1 (SPANNING). *Given a graph $G = (V, E)$ which is connected, undirected and unweighted, the* SPANNING *centrality of an edge e, denoted by $\mathrm{SC}(e)$, is defined as the fraction of spanning trees of G that contain this edge.*

By definition, $\mathrm{SC}(e) \in (0, 1]$. In cases where we want to specify the graph G used for the computation of the SC of an edge e, we extend the set of arguments of SC with an extra argument: $\mathrm{SC}(e, G)$.

Intuition: In order to develop some intuition, it is interesting to discuss which edges are assigned high SC scores: the only edges that achieve the highest possible SC score of 1 are *bridges*, i.e., edges that, if removed, make G disconnected. This is means that they participate in all possible spanning trees. The extreme case of bridges helps demonstrate the notion of importance captured by the SC scores for the rest of the edges. Assuming that spanning trees encode candidate pathways through which information propagates , then edges with high SC are those that, once removed, would incur a significant burden on the remaining edges.

Spanning centrality as an electrical measure: Our algorithms for computing the SPANNING centrality efficiently rely on the connection between the SC scores and the *effective resistances* of edges. The notion of effective resistance comes from viewing the input graph as an electrical circuit [11], in which each edge is a resistor with unit resistance. The effective resistance $R(u, v)$ between two nodes u, v of the graph — that may or may not be directly connected — is equal to the potential difference between nodes u and v when a unit of current is injected in one vertex (e.g., u) and extracted at the other (e.g., v).

In fact, it can be shown [5, 11] that for any graph $G = (V, E)$ and edge $e \in E$, the effective resistance of e, denoted by $R(e)$, is equal to the probability that edge e appears in a random spanning tree of G. This means that $\mathrm{SC}(e) = R(e)$. This fact makes the theory of resistive electrical networks [11] available to us. The details of these computations are given in the next section.

Spanning centrality for weighted graphs: We note here that all the definitions and the results we explain in the next sections also hold for weighted graphs under the following definition of SPANNING centrality: Given a weighted graph $G = (V, E, w)$, where $w(e)$ is the weight of edge e, the weighted SPANNING centrality of e is again defined as the fraction of all trees of G in which e participates in, but, in this case, the importance of each tree is weighted by its weight. Specifically, the SPANNING centrality in weighted graphs is computed as: $\sum_{T \in \mathcal{T}_e} w(T) / \sum_{T \in \mathcal{T}} w(T)$. Here, $\mathcal{T}$ refers to the set of all spanning trees of G, while $\mathcal{T}_e$ is the set of spanning trees containing edge e. Also, $w(T)$ denotes the weight of a single tree T and is defined as the *product* of the weights of its edges, i.e., $w(T) = \prod_{e \in T} w(e)$. In other words, when the edge weight corresponds to the probability of the existence of that edge, $w(T)$ corresponds to the likelihood of T. The weighted SPANNING centrality maintains the probabilistic interpretation of its unweighted version; it corresponds to the probability that edge e appears in a spanning tree of G, when the spanning trees are sampled with probability proportional to their likelihood $w(T)$.

All the algorithms that we introduce in the next section can be used for weighted graphs with the above definition of SPANNING centrality. The only modification one has to make is to form the $m \times m$ diagonal weight matrix W, such that $W(e, e) = w(e)$, and then define the weighted graph Laplacian as $L = B^T W B$. This matrix can then be used as an input to all of the algorithms that we describe below.

5. COMPUTING SPANNING CENTRALITY

In this section, we present our algorithm for evaluating the spanning centrality of all the edges in a graph. For that, we first discuss existing tools and how they are currently used. Then, we show how the SDD solvers proposed by Koutis *et al.* [24, 23] can speed up existing algorithms. Finally, we present a set of speedups that we can apply to these tools towards an efficient and practical implementation.

5.1 Tools

Existing algorithms for computing the SPANNING centrality are based on the celebrated Kirchoff's matrix-tree theorem [17, 34]. The best known such algorithm has running time $O(mn^{3/2})$ [33], which makes it impossible to use even on networks with a few thousands of nodes and edges.

Random projections for spanning centrality: The equivalence between $\mathrm{SC}(e)$ and the effective resistance of edge e, denoted by $R(e)$, allows us to take advantage of existing algorithms for computing the latter. The effective resistances of all edges $\{u, v\}$ are the diagonal elements of the $m \times m$ matrix R computed as [11]:

$$R = BL^{\dagger}B^T, \tag{1}$$

where B is the incidence matrix and $L^{\dagger}$ is the pseudoinverse of the Laplacian matrix L of G. Unfortunately, this computation requires $O(n^3)$ time.

Equation (1) provides us with a useful intuition: the effective resistance of an edge $e = \{u, v\}$ can be re-written as the distance between two vectors that only depend on nodes u and v. To see this consider the following notation: for node $v \in V$ assume an $n \times 1$ unit vector e_v with value one in its v-th position and zeros everywhere else (i.e., $e_v(v) = 1$ and $e_v(v') = 0$ for $v \neq v'$). Using Equation (1), we can write the effective resistance $R(e)$ between nodes $u, v \in V$ as follows:

$$R(e) = (e_u - e_v)^T L^\dagger (e_u - e_v) = \left\| BL^\dagger (e_u - e_v) \right\|_2^2.$$

Thus, the effective resistances of edges $e = \{u, v\}$ can be viewed as pairwise distances between vectors in $\{BL^\dagger e_v\}_{v \in V}$.

This viewpoint of effective resistance as the L_2^2 distance of these vectors, allows us to use the Johnson-Lindenstraus Lemma [20]. The pairwise distances are still preserved if we project the vectors into a lower-dimensional space, spanned by $O(\log n)$ random vectors. This observation led to Algorithm 1, which was first proposed by Spielman and Srivastava [31]. We refer to this algorithm with the name `TreeC`.

Algorithm 1 The `TreeC` algorithm.

Input: $G = (V, E)$.
Output: $R(e)$ for every $e = \{u, v\} \in E$
1: $Z = [\,]$, $L =$ Laplacian of G
2: Construct random projection matrix Q of size $k \times m$
3: Compute $Y = QB$
4: **for** $i = 1 \ldots k$ **do**
5: Approximate z_i by solving: $Lz_i = Y(:, i)$
6: $Z = [Z; z_i^T]$
7: return $R(e) = \|Z(:, u) - Z(:, v)\|_2^2$

In Line 2, a random $\{0, \pm 1/\sqrt{k}\}$ matrix Q of size $k \times m$ is created. This is the projection matrix for $k = O(\log n)$, according to the Johnson-Lindenstrauss Lemma. Using this, we could simply project matrix $BL^\dagger$ on the k random vectors defined by the rows of Q, i.e., computing $QBL^\dagger$. However, this would not help in terms of running time, as it would require computing $L^\dagger$ which takes $O(n^3)$ steps. Lines 3 and 5 approximate $QBL^\dagger$, without computing the pseudoinverse of L: first, $Y = QB$ is computed in time $O(2m \log n)$ — this is because B is sparse and has only $2m$ non-zero entries. Then, Line 5 finds an approximation of the *rows* z_i of matrix $QBL^\dagger$ by (approximately) solving the system $Lz_i = y_i$, where y_i is the i-th row of Y. Therefore, the result of the `TreeC` algorithm is the set of rows of matrix $Z = [z_1^T, \ldots, z_k^T]$, which is an approximation of $QBL^\dagger$. By the Johnson-Lindenstrauss lemma we know that, if $k = O(\log n)$, the `TreeC` algorithm will guarantee that the estimates $\tilde{R}(e)$ of $R(e)$ satisfy

$$(1 - \epsilon) R(e) \leq \tilde{R}(e) \leq (1 + \epsilon) R(e),$$

with probability at least $1 - 1/n$. We call ϵ the *error parameter* of the algorithm. Now, if the running time required to solve the linear system in Line 5 is $I(n, m)$, then the total running time of the `TreeC` algorithm is $O(I(n, m) \log n)$.

Incorporating SDD solvers: Now, if we settle for approximate solutions to the linear systems solved by `TreeC` and we deploy the SDD solver proposed by Koutis *et al.* [24, 23], then we have that $I(n, m) = \tilde{O}(m \log n)$, therefore achieving a running time of $\tilde{O}\left(m \log^2 n \log\left(\frac{1}{\epsilon}\right)\right)$. Additionally, with probability $(1 - 1/n)$, the estimates $\tilde{R}(e)$ of $R(e)$ satisfy

$$(1 - \epsilon)^2 R(e) \leq \tilde{R}(e) \leq (1 + \epsilon)^2 R(e). \quad (2)$$

We refer to the version of the `TreeC` algorithm that uses such solvers as the `Fast-TreeC` algorithm. The running time of `Fast-TreeC` increases linearly with the number of edges and logarithmically with the number of nodes. This dependency manifests itself clearly in our experiments in Section 7.

5.2 Speedups

We now describe three observations that lead to significant improvements in the space and runtime requirements of `Fast-TreeC`.

Space-efficient implementation: First, we observe that intermediate variables Y and Z of Algorithm 1 need not be stored in $k \times n$ matrices but, instead, vectors y and z of size $1 \times n$ are sufficient. The pseudocode that implements this observation is shown in Algorithm 2. In this case, the algorithm proceeds in $k = O(\log n)$ iterations. In each iteration, a single random vector q (i.e., a row of the matrix Q from Algorithm 1) is created and used for projecting the nodes. The effective resistance of edge $e = \{u, v\}$ is computed additively — in each iteration the portion of the effective resistance score that is due to the particular dimension is added to the total effective resistance score (Line 6 of Algorithm 2).

Algorithm 2 The space-efficient version of `Fast-TreeC`.

Input: $G = (V, E)$.
Output: $R(e)$ for every $e = \{u, v\} \in E$
1: $L =$ Laplacian of G
2: **for** $i = 1 \ldots k$ **do**
3: Construct a vector q of size $1 \times m$
4: Compute $y = qB$
5: Approximate z by solving: $Lz = y$
6: return $R(e) = R(e) + \|z(u) - z(v)\|_2^2$

Parallel implementation: Algorithm 2 reveals also that `Fast-TreeC` is amenable to parallelization. The execution of every iteration of the `for` loop (Line 2 of Algorithm 2) can be done independently and in parallel, in different cores, and the results can be combined. This leads to another running-time improvement: in a parallel system with $O(\log n)$ cores, the running time of the parallel version of the `Fast-TreeC` algorithm is $\tilde{O}\left(m \log n \log\left(\frac{1}{\epsilon}\right)\right)$. In all our experiments we make use of this parallelization.

Reducing the size of the input to the 2-core: As it has been observed in Section 4, the *bridges* of a graph participate in all the spanning trees of the graph and thus have SC score equal to 1. Although we know how to extract bridges efficiently [32], assigning to those edges SC score of 1 and applying the `Fast-TreeC` algorithm on each disconnected component would not give us the correct result. It is not clear how to combine the SC scores from the different connected components. However, we observe that this can still be done for a subset of the bridges.

Let us first provide some intuition before making a more general statement. Consider an input graph $G = (V, E)$ and an edge $e = \{u, v\}$ connecting node v of degree one to the rest of the network via node u. Clearly e participates in all spanning trees of G and, therefore, $\mathrm{SC}(e, G) = 1$. Now assume that edge e and node v are removed from G, resulting

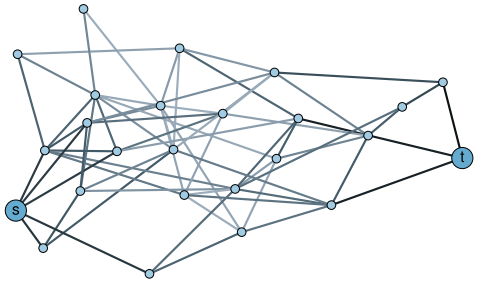


Figure 1: A network, viewed as an electrical resistive circuit. The thickness of an edge represents the amount of current it carries, if a battery is attached to nodes s and t.

into graph $G' = (V \setminus \{v\}, E \setminus \{e\})$. Since e was connecting a node of degree 1 to the rest of G, the *number* of spanning trees in G' is equal to the number of spanning trees in G. Thus, $\mathrm{SC}(e', G') = \mathrm{SC}(e', G)$ for every edge $e' \in E \setminus \{e\}$.

Now the key observation is that the above argument can be applied recursively. Formally, consider the input graph $G = (V, E)$ and let $C_2(G) = (V', E')$ be the 2-*core* of G, i.e., the subgraph of G that has the following property: the degree of every node $v \in V'$ in $C_2(G)$ is *at least* 2. Then, we have the following observation:

LEMMA 1. *If $G = (V, E)$ is a connected graph with 2-core $C_2(G) = (V', E')$, then for every edge $e \in E'$*

$$\mathrm{SC}(e, C_2(G)) = \mathrm{SC}(e, G).$$

The above suggests the following speedup for `Fast-TreeC`: given a graph $G = (V, E)$, first extract the 2-core $C_2(G) = (V', E')$. Then, for every edge $e \in E'$ compute $\mathrm{SC}(e)$ using the `Fast-TreeC` algorithm with input $C_2(G)$. For every $e \in E \setminus E'$, set $\mathrm{SC}(e) = 1$.

The computational savings of such a scheme depend on the time required to extract $C_2(G)$ from G. At a high level, this can be done by recursively removing from G nodes with degree 1 and their incident edges. This algorithm, which we call `Extract2Core`, runs in time $O(m)$ [3]. Our experiments (Section 7) indicate that extracting $C_2(G)$ and applying `Fast-TreeC` on this subgraph is more efficient than running `Fast-TreeC` on the original graph, i.e., the time required for running `Extract2Core` is much less than the speedup achieved by reducing the input size. By default, we use this speedup in all our experiments.

6. A GENERAL FRAMEWORK

In this section, we show that SPANNING centrality is an instance of a general class of edge-centrality measures, which we call *electrical measures* of edge centrality. We introduce a framework that offers a unified view to all the existing measures and leads to novel ones. Finally, we demonstrate how SDD solvers can be utilized within this framework.

6.1 Electrical measures of centrality

The common characteristic of the electrical measures of centrality is that they view the input graph G as a resistive circuit, i.e., an electrical network where every edge is a resistor of constant (e.g., unit) resistance. To get a better understanding of these measures, consider Figure 1. Suppose that we hook the poles of a battery to nodes s and t and apply a voltage difference sufficient to drive one unit of current (1A) from s to t. Doing that, each node in the network will get a voltage value and electrical current will flow essentially *everywhere*. At a high level, the electrical measures quantify the importance of an edge by *aggregating the values of the flows* that pass through it over different choices for pairs of nodes s and t. In fact, specific combinations of aggregation schemes and battery placements lead to different definitions of edge-importance measures.

More formally, consider two fixed nodes s and t on which we apply a voltage difference sufficient to drive one unit of current (1A) from s to t. Let the (s,t)-*flow* of edge $e = \{u, v\}$, denoted by $f_{st}(u, v)$, be the flow that passes through edge e in this configuration. We can now derive the following instances of electrical measures of edge centrality:

SPANNING **centrality:** For spanning centrality, we only consider a single battery placement and get the following alternative definition of the centrality of edge $e = \{u, v\}$:

$$\mathrm{SC}(e = \{u, v\}) = f_{uv}(u, v).$$

CURRENTFLOW **centrality:** If we consider the *average* flow that passes through an edge, where the average is taken over all distinct pairs of nodes (s, t), then we get another centrality measure known as current-flow:

$$\mathrm{CFC}(e = \{u, v\}) \triangleq \frac{1}{\binom{n}{2}} \sum_{(s,t)} f_{st}(u, v).$$

This measure was first proposed by Brandes and Fleisher [8].

From the combinatorial perspective, CURRENTFLOW considers an edge as important if it is used by many paths in the graph, while SPANNING focuses on the participation of edges in trees. The idea of counting paths is also central in the definition of *betweenness centrality* [7]. However, betweenness centrality takes into consideration only the shortest paths between the source-destination pairs. Therefore, if an edge does not participate in many shortest paths, it will have low betweenness score. This is the case even if that edge is still part of many relatively *short* paths. More importantly, the betweenness score of an edge may change by the addition of a small number of edges to the graph (e.g., edges that create triangles) [27]. Clearly, the CURRENTFLOW centrality does not suffer from such unstable behavior since it takes into account the importance of the edge in all the paths that connect all source-destination pairs.

β-CURRENTFLOW **centrality:** Instead of plugging a single battery in two endpoints (s, t), we can consider plugging β batteries into β pairs of distinct endpoints $\langle s_i, t_i \rangle$. For any such placement of β batteries, we can again measure the current that flows through an edge $e = \{u, v\}$ and denote it by $f_{\langle s_i, t_i \rangle}(u, v)$. Then, we define the β-CURRENTFLOW centrality of an edge as:

$$\beta\text{-}\mathrm{CFC}(e = \{u, v\}) = \frac{1}{|C_b|} \sum_{\langle s_i, t_i \rangle \in C_b} f_{\langle s_i, t_i \rangle}(u, v),$$

where C_b denotes the set of all feasible placements of β batteries in the electrical network defined by G. We can view β-CURRENTFLOW as a generalization of CURRENTFLOW; the two measures are identical when $\beta = 1$.

6.2 Computing electrical measures

In order to compute the centralities we described above, we need to be able to compute the flows $f_{st}(u,v)$. Using basic theory of electrical resistive networks, the computation of these flows for a fixed pair (s,t) can be done by solving the Laplacian linear system $Lx = b$. The right hand side of the system is a vector with the total residual flows on the nodes. Specifically, we let $b(s) = 1, b(t) = -1$ and $b(u) = 0$ for all $u \neq s,t$. This is because one unit of current enters s, one unit of current leaves s and a net current of 0 enters and leaves every other node by Kirchoff's law. As we have already discussed, setting a voltage difference between s and t assigns voltages to all the other nodes. The values of these voltages are given in the solution vector x. Then,

$$f_{st}(u,v) = |x(u) - x(v)| \tag{3}$$

The algorithm of Brandes and Fleisher [8]: From the above, the computation of one set of flows for a fixed pair (s,t) requires the solution of one linear system. Of course. we need $\binom{n}{2}$ linear systems in order to account for all pairs s and t. As shown by Brandes and Fleisher [8], it is enough to find the pseudo-inverse $L^\dagger$ of the Laplacian L *once*; then the scores can be computed in $O(mn \log n)$ time. In fact, this computation expresses the solution of each of the $\binom{n}{2}$ linear systems as a simple 'lightweight' linear combination of the solution of n systems that can be found in the columns of $L^\dagger$. Brandes and Fleisher point out that the pseudo-inverse can be computed via solving n linear systems in $O(mn^{3/2} \log n)$.

Proposed speedups: Using our state-of-the-art solver for SDD, the running time of finding $L^\dagger$ drops to $O(mn \log n)$, matching the post-processing part that actually computes the scores. Of course, this running time remains quadratic.

This worst-case running time can be improved in practice through sampling and parallelism. With sampling one can construct an estimator of a measure, say CFC, denoted by $\overline{\text{CFC}}$ as follows: instead of considering all pairs (s,t) we can only consider a set S_k of k pairs (s,t) that are selected uniformly at random. Similarly to Brandes and Fleisher [8] we define

$$\overline{\beta\text{-CFC}}(e = \{u,v\}) \triangleq \frac{1}{k} \sum_{(s,t) \in S_k} f_{st}(u,v), \tag{4}$$

which is an unbiased estimator of β-CFC(e).

Note that for each (s,t) pair one has to solve a linear system $Lx = b$ in order to obtain the values $f_{st}(u,v)$ of Equation (3). It is for those systems that we use the state-of-the art SDD solver. At the same time we observe that these systems can be solved independently for different vectors b, therefore parallelism can be exploited here too.

We call the algorithm that takes advantage of the state-of-the art SDD solver and the parallelism of `Fast-FlowC`. We evaluate the efficiency of this algorithm in Section 7.4.

7. EXPERIMENTS

In this section, we experimentally evaluate our methods for computing the spanning centrality and we study its properties with respect to edge additions/deletions and information propagation. For the evaluation, we use a large collection of datasets of different sizes, which come from a diverse set of domains.

Experimental setup: We implemented both `Fast-TreeC` and `Fast-FlowC` using a combination of Python, Matlab and C code. The CMG solver [25] is written mostly in C and can be invoked from Matlab. At the same time, in order to make our methods easily accessible, we compiled them as a Python library on top of the popular *networkx*[1] package [16]. The code is available online[2].

We ran all our experiments on a machine with 4 *Intel Xeon E5-4617 @ 2.9GHz*, with 512*GB* of memory. We need to note here that none of our algorithms pushed the memory of the machine near its limit. For `Fast-TreeC` and `Fast-FlowC` we used 12 hardware threads.

Datasets: In order to demonstrate the applicability of our algorithms on different types of data, we used a large collection of real-world datasets of varying sizes, coming from different application domains. Table 1 provides a comprehensive description of these datasets shown in increasing size (in terms of the number of edges). The smallest dataset consists of approximately 4×10^3 nodes, while the largest one has almost 3.8×10^6 nodes.

For each dataset, the first two columns of Table 1 report the number of nodes and the number of edges in the *Largest Connected Component* (LCC) of the graph that correspond to this dataset. The third and fourth columns report the number of nodes and edges in the 2-core of each dataset. The 2-core of a graph is extracted using the algorithm of Batagelj *et al.* [3]. The statistics of these last two columns will be revisited when we explore the significance of applying `Extract2Core` in the running time of `Fast-TreeC`.

In addition to their varying sizes, the datasets also come from a wide set of application domains, including collaboration networks (`HepTh`, `GrQc`, `DBLP` and `Patents`), social networks (`wiki-Vote`, `Slashdot`, `Epinions`, `Orkut` and `Youtube`), communication networks, (`Gnutella08`, `Gnutella31`, `skitter` and `Oregon`) and road networks (`roadNet-TX`). All the above datasets are publicly available through the *Stanford Large Network Dataset Collection* (SNAP). [3] For consistency, we maintain the names of the datasets from the original SNAP website. Since our methods only apply to undirected graphs, if the original graphs were directed or had self-loops, we ignored the directions of the edges as well as the self loops.

7.1 Experiments for SPANNING

Accuracy-efficiency tradeoff: Our first experiment aims to convey the practical semantics of the accuracy-efficiency tradeoff offered by the `Fast-TreeC` algorithm. For this, we recorded the running time of the `Fast-TreeC` algorithm for different values of the error parameter ϵ (see Equation (2)) and for different datasets. Note that the running times reported for this experiment are obtained after applying all the three speedups that we discuss in Section 5.2.

The results are shown in Figure 2; for better readability the figure shows the results we obtained only for a subset of our datasets (from different applications and with different sizes); the trends in other datasets are very similar. As expected, the running time of `Fast-TreeC` decreases as the value of ϵ, the error parameter, increases. Given that the

[1] `http://networkx.github.io/`
[2] `http://cs-people.bu.edu/cmav/centralities`
[3] `http://snap.stanford.edu/data`

Table 1: Statistics of the collection of datasets used in our experiments.

Dataset name	#Nodes in LCC	#Edges in LCC	#Nodes in the 2-Core	#Edges in the 2-Core
GrQc	4 158	13 422	3 413	12 677
Gnutella08	6 299	20 776	4 535	19 012
Oregon	11 174	23 409	7 228	19 463
HepTh	8 638	24 806	7 059	23 227
wiki-Vote	7 066	100 736	4 786	98 456
Gnutella31	62 561	147 878	33 816	119 133
Epinions	75 877	405 739	37 300	367 162
Slashdot	82 168	504 230	52 181	474 243
Amazon	334 863	925 872	305 892	896 901
DBLP	317 080	1 049 866	271 646	1 004 432
roadNet-TX	1 351 137	1 878 201	1 068 728	1 596 792
Youtube	1 134 890	2 987 624	470 164	2 322 898
skitter	1 694 616	11 094 209	1 463 934	10 863 527
Patents	3 764 117	16 511 740	3 093 271	15 840 895

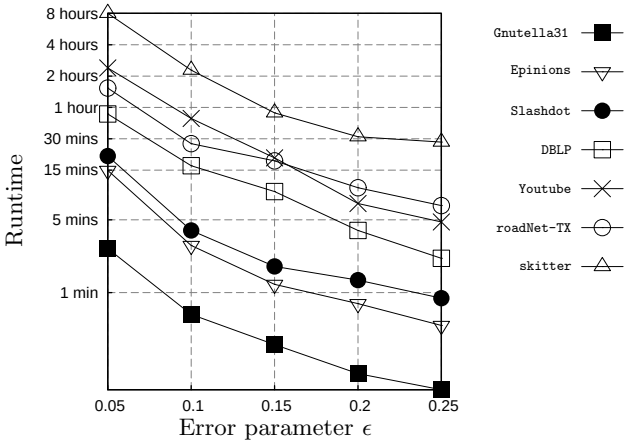


Figure 2: Accuracy-efficiency tradeoff; y-axis (logarithmic scale): running time of the `Fast-TreeC` algorithm; x-axis: error parameter ϵ.

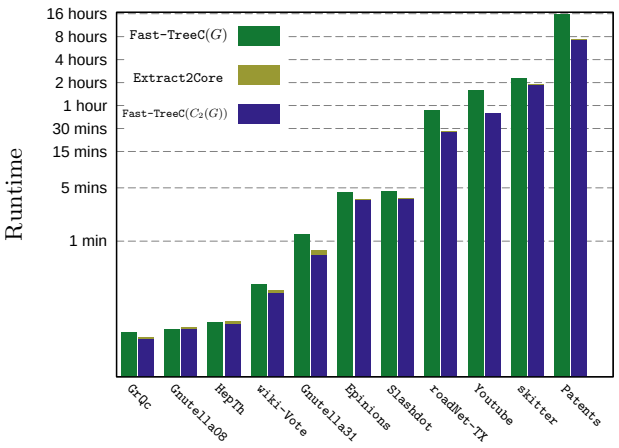


Figure 3: Limiting the computation on the 2-core shows a measurable improvement in the running time of `Fast-TreeC`.

y-axis in Figure 2 is logarithmic, this decrease in the running time is, as expected, exponential. Even for our largest datasets (e.g., `skitter` and `roadNet-TX`), the running time of `Fast-TreeC` even for very small values of ϵ (e.g., $\epsilon = 0.05$) was never more than 8 hours. Also, for $\epsilon = 0.15$, which is a very reasonable accuracy requirement , `Fast-TreeC` calculates the spanning centrality of all the edges in the graphs in less than 1 hour.

Also, despite the fact that the `roadNet-TX` and `skitter` datasets have almost the same number of nodes, `skitter` runs significantly faster than `roadNet-TX` for the same value of ϵ. This is due to the fact that `skitter` has approximately 5 times more edges than the corresponding graph of `roadNet-TX` and that the running time of `Fast-TreeC` is linear to the number of edges yet logarithmic to the number of nodes of the input graph.

Effect of the 2-core speedup: Here, we explore the impact of reducing the size of the input to the 2-core of the original graph on the running time of `Fast-TreeC`. For this, we fix the value of the error parameter $\epsilon = 0.1$, and run the `Fast-TreeC` algorithm twice; once using as input the original graph G and then using as input the 2-core of the same graph, denoted by $C_2(G)$. Then, we report the running times of both these executions. We separately also compute the time required to extract $C_2(G)$ from G using the `Extract2Core` routine described in Section 5.2.

Figure 3 shows the runtime for all these operations. In the figure, we use `Fast-TreeC`(G) (resp. `Fast-TreeC`$(C_2(G))$) to denote the running time of `Fast-TreeC` on input G (resp. $C_2(G)$). We also use `Extract2Core` to denote the running time of `Extract2Core` for the corresponding input. For each of these datasets, we computed the SC scores of the edges, before (left) and after (right), and we report the running time in the y-axis using logarithmic scale.

Note that on top of the box that represents the runtime of `Fast-TreeC`$(C_2(G))$, we have also stacked a box with size relative to the time it took us to find that 2-core subgraph. It is hard to discern this box, because the time spent for `Extract2Core` is minimal compared to the time it took to compute the SPANNING centralities. Only in the case of smaller graphs is this box visible, but again, in these cases, the total runtime does not exceed a minute. For instance, for `Patents` (our largest graph) spending less than 5 minutes to find the 2-core of the graph lowered the runtime of `Fast-`

TreeC down to less than 8 hours, which is less than half of the original. Moreover, the difference between the height of the left and the right bar for the different datasets behaves similarly. Hence, as the size of the dataset and the runtime of Fast-TreeC grow exponentially, so does the speedup.

7.2 Resilience under noise

In this experiment, we evaluate the resilience of SPANNING centrality to noise that comes in the form of adding and deleting edges in the original graph. We also compare the resilience of SPANNING to the resilience of CURRENTFLOW centrality [8] and BETWEENNESS centrality [7, 12], which are the most commonly used measures of edge centrality.

Noise: Given the original graph $G = (V, E)$ we form its noisy version of $G' = (V, E')$ by either *adding* or *deleting* edges. We consider three methods for edge addition: (a) `random` that picks two disconnected nodes at random and creates an edge, (b) `heavy` that selects two nodes with probability proportional to the sum of their degrees and (c) `light` that selects two nodes with probability inversely proportional to the sum of their degrees. Note that adding edges with `heavy` imitates graph evolution under the scale-free models, while the addition of edges with `light` imitates the evolution of newly-added nodes in an evolving network. The edge deletion is performed similarly; we delete an already existing edge (a) randomly, (b) with probability proportional to the sum of the degrees of its endpoints or (c) with probability inversely proportional to the sum of the degrees of its endpoints.

The number of edges ℓ added to or deleted from G is a parameter to our experiment – expressed as a percentage of the number of original edges in G.

Evaluation: We evaluate the resilience of a centrality measure both in terms of the values it assigns to edges that are both in G and G', as well as the ranking that these values induce.

Formally, given a graph $G = (V, E)$ and its noisy version $G' = (V, E')$, let e be an edge in $E \cap E'$. If c_e is the centrality score of the edge in G and c'_e the value of the same score in G', then we define the relative change in the value of e as:

$$\texttt{RelChange}(e, G, G') = \frac{|c_e - c'_e|}{c_e}.$$

In order to aggregate over all edges in E, we define the average relative change of c with respect to G and G' as

$$\texttt{AvgRC}(G, G') = \frac{1}{|E \cap E'|} \sum_{e \in E \cap E'} \texttt{RelChange}(e).$$

This evaluation metric captures the average relative change in the value of the centrality scores in G and G'. Observe that for edge additions, $E \cap E' = E$, while, for edge deletions $E \cap E' = E'$. In general, $\texttt{AvgRC}(G, G')$ takes values in $[0, \infty)$ and smaller values imply a more resilient centrality measure.

In order to evaluate how the ranking of the edges according to a centrality measure changes in G and G' — ignoring the actual values of the measure — we proceed as follows: first we generate the sets of edges that contain the top-k% edges in G and G', denoted by $I(G, k)$ and $I(G', k)$ resp. Then, we compare these sets using their Jaccard similarity. That is, we define:

$$\texttt{JaccSim}(G, G', k) = \frac{|I(G, k) \cap I(G', k)|}{|I(G, k) \cup I(G', k)|}.$$

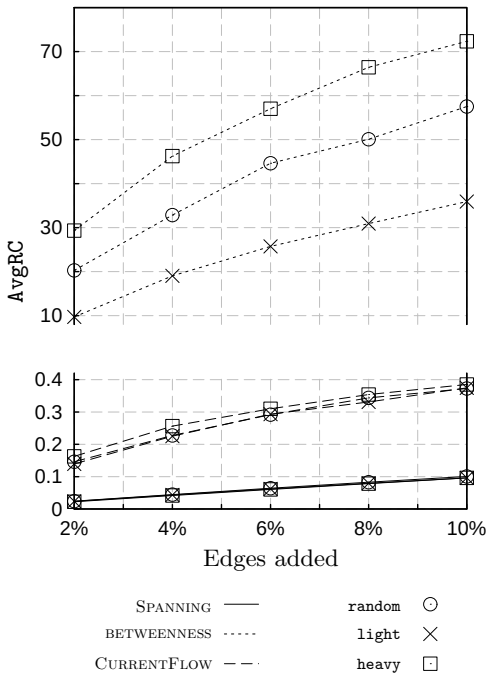


Figure 4: Average relative change in the edge importance scores. The x-axis shows the percentage of noisy edges added to the original HepTh graph.

JaccSim takes values in $[0, 1]$. Large values of Jaccard similarity mean that the sets $I(G, k)$ and $I(G', k)$ are similar. Consequently, larger values of JaccSim indicate higher resilience of the measure under study.

Results: Figures 4 and 5 show the noise resilience of SPANNING, CURRENTFLOW and BETWEENNESS centrality under edge addition. This is measured using both AvgRC (Fig. 4) as well as JaccSim (Fig. 5). The results that are shown are for the HepTh network, but the trend was the same in all the datasets that we tried. Note that while we use the fastest known algorithms for both CURRENTFLOW [8] and BETWEENNESS [7] (implemented in NetworkX), these algorithms remain a bottleneck, so we cannot present comparative experiments for larger networks. The edge additions are performed using all the three sampling methods we mentioned above, i.e. `random`, `heavy` and `light`. The shown results are averages over 10 different independent runs.

In both figures the x-axis corresponds to the number of edges being added to form E' as a fraction of the number of edges in $G = (V, E)$. For Figure 5, we picked $k = 10$ for the percentage of the highest-ranking edges whose behavior we want to explore; results for other values of k have very similar trends.

Overall, we observe that as we add more edges, the AvgRC increases for all the centralities, while $\texttt{JaccSim}(G, G', k)$ decreases. This is expected, since we increasingly alter the structure of the graph. What is surprising though is how significantly smaller the values of AvgRC for SPANNING are, especially when compared to the corresponding values for BETWEENNESS, for the same number of edge additions. As shown in Figure 4, SPANNING has, in the worst case, AvgRC

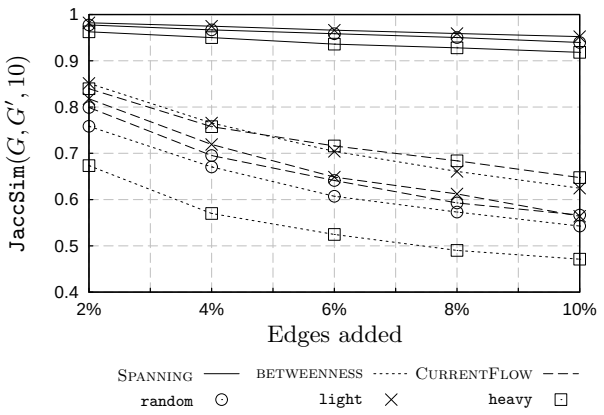


Figure 5: Jaccard similarity between the top 10%-scoring edges in the original and the noisy graph. Note how resilient the SPANNING centrality is to noise.

of value less than 0.1. In contrast, `AvgRC` of BETWEENNESS reaches up to 71.5 for the `light` sampling. This indicates that the values of SPANNING centrality are much more stable under the edge addition schemes we consider than the values of BETWEENNESS centrality. We also observe that CURRENTFLOW exhibits behavior between 2× and 4× worse than SPANNING, which is still much better than BETWEENNESS.

The results shown in Figure 5 show that the ranking of edges implied by the SPANNING measure is also much more stable than the ranking implied by either BETWEENNESS or CURRENTFLOW. More specifically, for $k = 10$, we observe that `JaccSim` has, in the worst case, score equal to 0.91 – which is very close to 1. The corresponding values for both BETWEENNESS and CURRENTFLOW are significantly lower, at 0.47 and 0.57 respectively. In addition to the above, we observed that, even when adding a 10% noise to the graph edges, more than 95% of the edges in $I(G, 10)$ are still deemed important by SPANNING. Note that the trends observed for $\texttt{JaccSim}(G, G', k)$ are similar for values of k that are smaller than 10%. Due to the similarity of these results to the ones we present here, we omitted them.

Overall, the resilience of SPANNING under edge additions demonstrates that even when SPANNING is computed over evolving graphs, it need not be recomputed frequently. In fact, our experiments indicate that even when 10% more edges are added in the original graph, both the centrality values as well the ranking of edges implied by the SPANNING centrality remain almost the same. Note that, although we only show here the results for edge additions, our findings for edge deletions are very similar and thus we omit them.

Resilience of β-CURRENTFLOW: In addition to the experiments we presented below, we also investigated the resilience of β-CURRENTFLOW. Our results are summarized as follows: for larger values of β, the `AvgRC` decreases to values that are smaller than the values we observe for BETWEENNESS. However even the smallest values are in the range $[3, 5]$, thus never as small as the values we observe for SPANNING. The values of `JaccSim` on the other hand are consistently around 0.5. Although these results are not extensive, we conclude that the SPANNING centrality is significantly more resilient than β-CURRENTFLOW under edge additions and deletions.

7.3 Edge-importance measures and information propagation

A natural question to consider is the following: what do all the different edge-importance measures capture and how do they relate to each other? Here, we describe an experiment that allows us to quantitatively answer this question. On a high level, we do so by investigating how edges ranked as important (or less important) affect the result of an information-propagation process in the network.

Methodology: Given a network $G = (V, E)$, we compute the spread of an information propagation process, by picking 5% of the nodes of the graph, running the popular independent cascade model [22] using these nodes as seeds and computing the expected number of infected nodes in the end of this process. By repeating the experiment 20 times and taking the average we compute the $\textsc{Spread}(G)$.

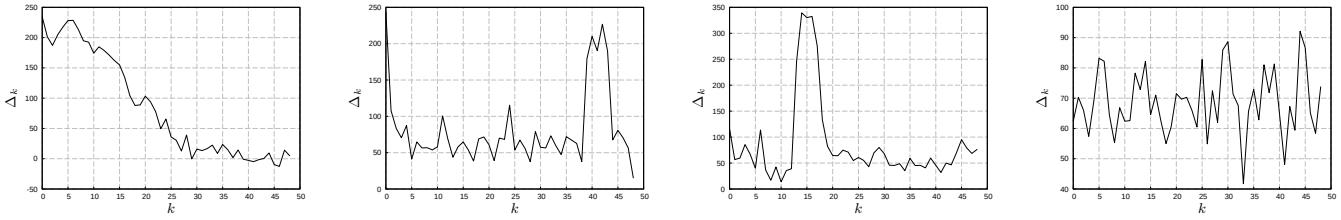
For any edge-importance measure, we compute the scores of all edges according to this measure, rank the edges in decreasing order of this score and then pick a set of ℓ edges, where $\ell = 0.02|E|$, such that they are at positions $((k-1)\ell + 1) \dots k\ell$, for $k = 1, \dots, |E|/\ell$. We refer to the set of edges picked for any k as E_k. For every k, we then remove the edges in E_k, forming graph G_k, and then compute $\textsc{Spread}(G_k)$. In order to quantify the influence that the set E_k has on the information-propagation process we compute:

$$\Delta_k = \textsc{Spread}(G) - \textsc{Spread}(G_k).$$

Clearly, larger the values of Δ_k imply a larger effect of the removed edges E_k on the propagation process.

We experiment with the following four measures of importance: SPANNING, CURRENTFLOW, BETWEENNESS and RANDOM. Recall that the betweenness score of an edge is the fraction of all pairs shortest paths that go through this edge [12]. RANDOM simply assigns a random order on the edges in E.

Results: Figure 6 shows the values of Δ_k in the case of the `HepTh` network, for $k = 1, \dots, 49$ when sets E_k are determined by the different importance measures. Overall we observe that the trend of Δ_k varies across measures. More specifically, in the case of SPANNING centrality (Figure 6a), Δ_k takes larger values for small k and appears to drop consistently until $k = 30$. This behavior can be explained by the definition of the SPANNING centrality – an edge is important if it is part of many spanning trees in the network – and the fact that the propagation of information in a graph can be represented as a spanning tree. CURRENTFLOW and BETWEENNESS (Figures 6b and 6c) behave differently. They appear to give medium importance scores to edges that have high impact on the spread. For CFC, these are the edges in E_k for $k \in [38, 43]$ and, for BETWEENNESS, the edges for $k \in [13, 18]$. These edges correspond to the peaks we see in Figures 6b and 6c. Observe that, for BETWEENNESS, this peak appears for smaller values of k, indicating that in this particular graph, there are edges that participate in relatively many shortest paths and, once removed, they disconnect the network, hindering the propagation process. Finally, the results for RANDOM show no specific pattern, indicating that what we observed in Figures 6a–6c is statistically significant.



(a) SPANNING (b) CURRENTFLOW (c) BETWEENNESS (d) RANDOM

Figure 6: The values Δ_k as a function of k for the different importance measures.

7.4 Experiments with CURRENTFLOW

In this last experiment, we give some indicative examples of the efficiency of `Fast-FlowC`, which we described in the end of Section 6.

One of the major problems that `Fast-FlowC` has to solve is finding the right number of samples k that will be used for the evaluation of Equation (4). In practice we deal with this as follows: we run `Fast-FlowC` in epochs, each epoch consisting of 1000 independent samples of (s,t) pairs. For the rest of the discussion we will use `Fast-FlowC`(i) to refer to `Fast-FlowC` that stops after i epochs. If we use F^* to denote the ground-truth centrality values, computed via an exhaustive algorithm that goes through all (s,t) pairs, and F^i to denote the output of `Fast-FlowC`(i), then we could decide to stop when `CorrD`(F^*, F^i) is reasonably small. Here, `CorrD`(F, F') is the correlation distance between the two vectors F and F', and is computed as 1 minus their correlation coefficient. Thus, `CorrD`$(F, F') \in [0, 2]$.

In the absence of ground-truth, we use the *self-correlation index* $\tau =$ `CorrD`(F^i, F^{i-1}) to decide whether the number of samples is sufficient. We terminate when τ is close to 0.

In order not to bias our experiments with the large number of edges that have very small centrality scores, we only consider the top-10% scored edges in F^i and F^{i-1} and we compute `CorrD`(F^{i-1}, F^i) projected on the union of these sets of edges. After all, importance measures aim at finding the highly scoring edges.

Also, in our experiments we found that drawing 1000 samples of (s,t) pairs between any two consecutive iterations of `Fast-FlowC`(i) is adequate to guarantee that the correlation between F^i and F^{i-1} is due to the convergence of the sampling procedure and not the closeness of the readings.

Ideally, we would like `CorrD`(F^i, F^*) to be small for values of i for which τ is also small. Our experiments with small datasets indicate that this is the case. As a result, τ can be used as a proxy for convergence of `Fast-FlowC` for larger datasets.

In Table 2, we report the running time of β-`Fast-FlowC` for $\tau < 0.02$ as well as the running time for the exact computation of CURRENTFLOW (from NetworkX) for three different datasets: `GrQc`, `Oregon` and `Epinions`. Note that for `Epinions`, the largest of the three datasets, the exact algorithm does not terminate within a reasonable time. On the other hand for the medium-size dataset, i.e., `Oregon`, `Fast-FlowC` takes only 2-3 minutes (depending on the choice of β), while the exact algorithm requires time close to 5 hours.

Dataset	`Fast-FlowC` algorithm			`Exact`
	($\beta = 1$)	($\beta = 5$)	($\beta = 20$)	algorithm
`GrQc`	2.1 mins	1.3 mins	1.3 mins	20 mins
`Oregon`	3.3 mins	1.9 mins	1.4 mins	4h 40mins
`Epinions`	12h 23mins	5h 26mins	3h	n/a

Table 2: Time until termination of `Fast-FlowC` and the exact algorithm. We terminate for $\tau < 0.02$.

8. CONCLUSIONS

In this paper, we studied the problem of efficiently computing the SPANNING centrality of edges in large graphs. More specifically, we described a randomized approximate algorithm that builds upon seminal work in the theory community on the effective resistances of graphs and on efficiently solving laplacian linear systems. While the algorithm we describe exploits these ideas, we also introduce efficient speedups, that we deploy in order to achieve scalability even for very large graphs. Overall, our experimental evaluation gives ample empirical evidence of the efficiency of the proposed algorithm and its ability to handle large graphs. Indicatively, our method can compute the SPANNING centrality of all the edges of a graph with more than 1.5 million nodes and 11 million edges in less than 30 minutes, while offering low error guarantees. In addition to that, our experiments explored the properties of SPANNING and showed that (i) it is resilient to noise and (ii) ranks the edges according to their participation in an information-propagation process. In an attempt to generalize the ideas behind SPANNING centrality, we showed that it is an instance of a class of edge-importance measures, which we call *electrical measures*. In the paper, we showed multiple instances of such measures and demonstrated how the algorithmic tools we relied upon for the computation of SPANNING can also be used for efficiently computing these measures.

Acknowledgments: The authors would like to thank Petros Drineas, George Kollios and Malik Magdon-Ismail for useful discussions on topics related to this paper. This research was supported by: NSF CAREER #1253393, NSF grants: CNS #1017529, III #1218437, IIS #1320542 and gifts from Microsoft and Hariri Institute of Computing. I. Koutis is supported by NSF CAREER CCF-1149048. Part of this work was undertaken while I. Koutis was visiting ICERM (Institute of Computational and Experimental Research in Mathematics).

9. REFERENCES

[1] J. M. Anthonisse. *The rush in a directed graph*. SMC, 1971.

[2] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, 2007.

[3] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, 2003.

[4] A. Bavelas. A mathematical model for group structure. *Human Organizations*, 7, 1948.

[5] B. Bollobas. *Modern Graph Theory*. Springer, 1998.

[6] S. P. Borgatti. Centrality and network flow. *Social Networks*, 27, Jan. 2005.

[7] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2), 2001.

[8] U. Brandes and D. Fleischer. Centrality measures based on current flow. In *STACS*, 2005.

[9] U. Brandes and C. Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(7), 2007.

[10] P. De Meo, E. Ferrara, G. Fiumara, and A. Ricciardello. A novel measure of edge centrality in social networks. *Knowledge-based Systems*, 30, 2012.

[11] P. Doyle and J. Snell. *Random walks and electric networks*. Math. Assoc. America., Washington, 1984.

[12] L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1), Mar. 1977.

[13] L. C. Freeman, S. P. Borgatti, and D. R. White. Centrality in valued graphs: A measure of betweenness based on network flow. *Social networks*, 13(2), 1991.

[14] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, 2008.

[15] M. Gomez-Rodriguez, J. Leskovec, and A. Krause. Inferring networks of diffusion and influence. *TKDD*, 5(4), 2012.

[16] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Aug. 2008.

[17] J. M. Harris, J. L. Hirst, and M. J. Mossinghoff. *Combinatorics and Graph Theory*. Undergraduate Texts in Mathematics, Springer, 2008.

[18] C. Huitema. *Routing in the Internet*. Prentice Hall, 2000.

[19] V. Ishakian, D. Erdös, E. Terzi, and A. Bestavros. Framework for the evaluation and management of network centrality. In *SDM*, 2012.

[20] W. Johnson and J. Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. In *Conference in modern analysis and probability*, 1982.

[21] U. Kang, S. Papadimitriou, J. Sun, and H. Tong. Centralities in large networks: Algorithms and observations. In *SDM*, 2011.

[22] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003.

[23] I. Koutis, G. L. Miller, and R. Peng. A nearly-m log n time solver for sdd linear systems. In *FOCS*, 2011.

[24] I. Koutis, G. L. Miller, and R. Peng. A fast solver for a class of linear systems. *Commun. ACM*, 55(10), 2012.

[25] I. Koutis, G. L. Miller, and D. Tolliver. Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing. *Computer Vision and Image Understanding*, 115(12), 2011.

[26] T. Lappas, E. Terzi, D. Gunopulos, and H. Mannila. Finding effectors in social networks. In *KDD*, 2010.

[27] M. Newman. A measure of betweenness centrality based on random walks. *Social networks*, 27(1), 2005.

[28] R. J. Perlman. An algorithm for distributed computation of a spanningtree in an extended lan. In *SIGCOMM*, 1985.

[29] G. Royle and C. Godsil. *Algebraic Graph Theory*. Graduate Texts in Mathematics. Springer Verlag, 1997.

[30] A. Shimbel. Structural parameters of communication networks. *Bulletin of Mathematical Biology*, 15, 1953.

[31] D. A. Spielman and N. Srivastava. Graph sparsification by effective resistances. *SIAM J. Comput.*, 40(6), 2011.

[32] R. E. Tarjan. A note on finding the bridges of a graph. *Inf. Process. Lett.*, 2(6), 1974.

[33] A. S. Teixeira, P. T. Monteiro, J. A. Carrico, M. Ramirez, and A. P. Francisco. Spanning edge betweenness. In *Workshop on Mining and Learning with Graphs*, 2013.

[34] W. Tutte. *Graph Theory*. Cambridge University Press, 2001.