2018

Finding important entities in graphs

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

FINDING IMPORTANT ENTITIES IN GRAPHS

by

CHARALAMPOS MAVROFORAKIS

B.S., Athens University of Economics and Business, 2010

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2018

Approved by

First Reader

Evimaria Terzi, PhD
Associate Professor of Computer Science

Second Reader

George Kollios, PhD
Professor of Computer Science

Third Reader

Alina Ene
Assistant Professor of Computer Science

Acknowledgments

This work is dedicated to my parents.

This thesis is the result of a long and beautiful journey. I never felt alone and I have many people to thank for this.

First and foremost, I am extremely grateful to my advisor, Evimaria Terzi. From the first day we met, I was positively surprised by the passion and the curiosity I saw in Evimaria. She quickly turned into a bottomless source of motivation and excitement for me, being a beacon of light whenever I truly needed it. Our collaboration could not have been smoother; she was always clear and straight-forward with everything we discussed and made an effort to make sure that I had access to whatever I needed, whenever I needed it. I will always remember the countless hours we spent in her office, brainstorming and solving problems on the whiteboard. Our conversations with Evimaria soon expanded much further than the academic matters; she was very eager to discuss about life and share her precious advice on difficult topics. I feel lucky and honored to be able to call Evimaria a friend.

Secondly, I want to thank George Kollios for his assistance throughout this journey. George was the first person to welcome me to Boston University and guided me during the busy first years. When he noticed that I found a topic that truly excited me, he was very supportive to let me pursue it. Still, his office door was always open for me. An important yet practical issue that I am grateful to George for is that he always made sure our lab had the best coffee in the department.

Both Evimaria and George were very supportive of the idea of collaborating with other people. As a result, I met and worked together with many brilliant researchers. I would like to thank all of my coauthors in the past years, namely, Ran Canetti, Nathan Chenette, Mark Crovella, Alina Ene, Dora Erdos, Francesco Fusco, Aris Gio-

nis, Manuel Gomez Rodriguez, Yiannis Koutis, Anastasios Kyrillidis, Michael Mathioudakis, Sofia Nikolakaki, Adam O'Neill, Isabel Valera, Vassilios Vassiliadis, and Michalis Vlachos. They all helped me learn new things that significantly expanded my understanding. I would also like to thank Marwan Mattar and Haohan Zhu, who were my mentors during my internships at Electronic Arts and Facebook, respectively. In addition to their technical advice, their precious lessons on soft skills made my life in the companies much easier. A special thank you goes to Alina Ene, with whom I worked very closely during this last year. Alina was very patient with me and taught me a great number of things. I truly enjoyed our collaboration.

I am also grateful to the people at the Boston University Computer Science department, who made every possible effort to keep things running smoothly every day and handled all my strange requests without second thought. Thank you Chris, Jennifer, Jacob, Jess, Theresa, Ellen, Nora, Faith, Paul and Joe. The same feeling also extends to the organizations and foundations that made sure I am getting funded during my PhD; the support and trust of the Gerondelis foundation, the National Science Foundation, and of Nicholas Gage honors me.

I was lucky to always have fun, interesting, and supportive labmates. Our lab was always a warm place and a gathering point. Thank you Behzad, Haohan, Dora, Natali, Xianrui, Sofia, Harshal, Sanaz, Isidora, Hao, and Dmytro for making the place I spent most of my time during the day so pleasant.

A big thank you goes to my friends. I am greatly indebted to my pre-PhD friends, for still being my friends. The new friends I made during my PhD made life more colorful; thank you Antonia, Natali, Jeff, Davide, Sofia, Harshal, Foteini, Sokratis, Stavros, Amina, Behzad, Sanaz, Larissa, Giovanni, Thodoris, Kostas, Anastasia, and Athina for always being there through the best and the worst times.

Returning home to a warm, friendly and relaxing environment is very important

in the day-to-day struggle of the PhD. For this, I have to thank Alaz, George and Konstantinos for being great roommates and even better friends. A special thank you goes to Dimitris Papadopoulos. He was one of the first people I met when I joined Boston University, and ended up being one of the best. While progressing through the challenges of the PhD together, his support was priceless. In him, I found an amazing friend, an awesome roommate, and, in general, the person that everyone would wish to be around.

Last, but not least, nothing would have been possible without the unlimited love and support of my parents and my family. I will always be grateful for their priceless advice and their unconditional sacrifices for me. Thank you for giving me the luxury to pursue my dreams, and for standing by my side when this proved harder than expected. Finally, my deepest gratitude and thanks go to my grandmothers, Christina and Dimitra, who both shared an important part in my upbringing, but did not manage to see this work finished.

FINDING IMPORTANT ENTITIES IN GRAPHS

CHARALAMPOS MAVROFORAKIS

Boston University, Graduate School of Arts and Sciences, 2018

Major Professors: Evimaria Terzi, PhD
Associate Professor of Computer Science

George Kollios, PhD
Professor of Computer Science

ABSTRACT

Graphs are established as one of the most prominent means of data representation. They are composed of simple entities – nodes and edges – and reflect the relationship between them. Their impact extends to a broad variety of domains, e.g., biology, sociology and the Web. In these settings, much of the data value can be captured by a simple question; how can we evaluate the importance of these entities? The aim of this dissertation is to explore novel importance measures that are meaningful and can be computed efficiently on large datasets.

First, we focus on the *spanning edge centrality*, an edge importance measure recently introduced to evaluate phylogenetic trees. We propose very efficient methods that approximate this measure in near-linear time and apply them to large graphs with millions of nodes. We demonstrate that this centrality measure is a useful tool for the analysis of networks outside its original application domain.

Next, we turn to importance measures for nodes and propose the *absorbing random walk centrality*. This measure evaluates a group of nodes in a graph according to how central they are with respect to a set of query nodes. Specifically, given a query set and a candidate group of nodes, we start random walks from the queries and

measure their length until they reach one of the candidates. The most central group of nodes will collectively minimize the expected length of these random walks. We prove several computational properties of this measure and provide an algorithm, whose solutions offer an approximation guarantee. Additionally, we develop efficient heuristics that allow us to use this importance measure in large datasets.

Finally, we consider graphs in which each node is assigned a set of attributes. We define an important connected subgraph to be one for which the total weight of its edges is small, while the number of attributes covered by its nodes is large. To select such an important subgraph, we develop an efficient approximation algorithm based on the primal-dual schema.

Contents

List of Tables

List of Figures

List of Abbreviations

$\mathbb{R}^2$	.....	the Real plane

Chapter 1

Introduction

In the modern world, it has become clear that data is a very significant and precious asset. Not only is it produced at a continuously increasing rate, but at the same time, technology advancements have enabled us to store more data than ever before. This trend, which has been popularized as "big data", is changing the industry. Companies are offering personalized services to their clients, who, in turn, can take more informed decisions and can build better insights. At the same time, certain scientific tools, such as *deep learning*, are becoming increasingly popular, since they seem to be surprisingly successful when they operate on large datasets, and are promising to have a huge impact in our everyday lives.

1.1 Graphs as a means for data representation

As a result of this recent trend, a lot of effort is being focused on data storage techniques, data management solutions, and scalable algorithms. In this context, *graphs* have proven themselves as one of the most prominent methods for representing data.

A *graph* is a mathematical structure, that is composed of two main *entities*; the *nodes* and the *edges*. The former usually represent objects, such as people, products, devices, or even abstract concepts, while the latter capture the relationships between these objects. The applications of graphs are very broad, ranging from computer science and physics to linguistics and art. Specifically in computer science, one may

find all kinds of data stored as a graph;

- traffic patterns on city roads,
- interactions between proteins,
- social interactions and communication,
- terrorist collaboration,
- progression of news stories,
- ontologies

The above is just a brief, non-exhaustive collection of scenarios, where graphs were employed as the chosen method for representation.

1.2 Evaluating the importance of graph entities

A common goal of most data owners is to find a way to turn their data into value. Frequently, this is achieved by asking seemingly simple questions, such as, *"Which entities are more important in my graph?"*. Being a question with multiple answers, this has led to the development of a large and diverse body of works, mostly in the fields of network science, graph mining, and data mining in general. The methods proposed in this context are popularly known as *importance measures*, or *centrality scores*. Depending on the target of the data analysis, these measures are either defined for the nodes or for the edges of the graph.

Although the importance measures are numerous and designed for different purposes, they can be broadly grouped in two families:

- ranking-based measures,
- selection-based measures

Following, we provide a description of these two categories and give representative examples from the literature.

1.2.1 Ranking-based importance measures

The first category of measures consists of methods that assign a score to each entity in the graph, node or edge depending on its definition, separately. Then, an ordering of these entities is reported according to their computed score. The vast majority of the centrality measures belong in this category. Notable examples are degree centrality, Pagerank (Brin and Page, 1998), and betweenness centrality (Freeman, 1977; Anthonisse, 1971) for nodes, and edge betweenness centrality (Anthonisse, 1971), and current-flow betweenness centrality (Brandes and Fleischer, 2005) for edges.

1.2.2 Selection-based importance measures

The ranking-based measures cannot be used directly when the goal is to evaluate a set of entities, since they are defined on individual nodes and edges, and not on groups. The selection-based importance measures are designed to fill this gap. Their goal is, given an integer $k > 0$, to evaluate groups of size k in the graph and, mainly, to find the best scoring such group. Examples in this category are group degree centrality, group betweenness centrality (Everett and Borgatti, 1999), and co-betweenness (Kolaczyk et al., 2009). A sub-category of the selection-based measures are the ones we call *parameter-free*. Instead of requiring k to be part of the input, these measures use a *prize* or *penalty* function to control the size of the group. Representative examples in this sub-category are the prize-collecting Steiner tree problem (Bienstock et al., 1993), and the prize-collecting traveling salesman problem (Balas, 1989).

1.3 Contributions of this thesis

The work in this thesis spans both the ranking-based and the selection-based categories of importance measures.

First, in Chapter 2, we focus on the *spanning edge centrality*, an edge importance measure recently introduced to evaluate phylogenetic trees (Teixeira et al., 2013) that belongs to the ranking-based category. The method for computing this centrality score however is very expensive computationally, which makes it impractical to use in real-world data. Here, we propose very efficient methods that approximate this measure in near-linear time and we apply them to large graphs with millions of nodes (Mavroforakis et al., 2015a). Adding to the measure's already established practical impact, we demonstrate that the spanning edge centrality is a useful tool for the analysis of networks outside its original application domain.

Next, in Chapter 3, we turn to the category of selection-based importance measures and propose the *absorbing random walk centrality* (Mavroforakis et al., 2015b), a centrality score for groups of nodes. This measure evaluates a group of k nodes in a graph according to how central they are with respect to a set of query nodes. Specifically, given a query set and a candidate group of nodes, we start random walks from the queries and measure their length until they reach one of the candidates. The most central group of nodes will collectively minimize the expected length of these random walks. We prove several computational properties of this measure and provide an algorithm that finds a group of k nodes with a constant-factor approximation guarantee. Additionally, we develop efficient heuristics that allow us to use this importance measure in large datasets.

Finally, in Chapter 4, we shift our focus to a *parameter-free* selection-based masure for graphs. Here, we assume that each node is assigned a set of attributes. We define an important connected subgraph (group) to be one for which the total weight of

its edges is small, while the number of attributes covered by its nodes is large. To select such an important subgraph, we develop an efficient approximation algorithm based on the primal-dual schema. Furthermore, we apply this measure on a real co-authorship network, in order to showcase its practical capacities and impact.

Chapter 2

Spanning edge centrality

2.1 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 (Brandes, 2001). 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 2.2). 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 (Newman, 2005). 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 (Gomez-Rodriguez et al., 2012; Lappas et al., 2010). Similarly, in computer networks, packages are distributed in the network through tree-shaped structures (Huitema, 2000; Perlman, 1985).

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.* (Teixeira et al., 2013) 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 (Royle and Godsil, 1997). 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 (Teixeira et al., 2013), the required computation time appears to be prohibitive for most networks of interest.

In this chapter 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 2.5). 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 a an electrical resistive network. The core component of our implementation is a fast linear system solver for Laplacian matrices (Koutis et al., 2011b). The computation of spanning centrality is also crucially based on the remarkable work of Spielman and Srivastava (Spielman and Srivastava, 2011). 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 2.6).

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 2.7). 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.

2.2 Related work

In the graph-mining literature, there exists a plethora of measures for quantifying the importance of network nodes or edges (Anthonisse, 1971; Bavelas, 1948; Borgatti, 2005; Brandes, 2001; Brandes and Fleischer, 2005; De Meo et al., 2012; Freeman, 1977; Freeman et al., 1991; Ishakian et al., 2012; Kang et al., 2011; Newman, 2005; Shimbel, 1953; Teixeira et al., 2013). 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 (Brandes, 2001). 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 (Brandes and Pich, 2007) 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.* (Geisberger et al., 2008) 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.* (Bader et al., 2007) 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 (Brandes and Fleischer, 2005). 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.* (De Meo et al., 2012) 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.

2.3 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 chapter, 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 (Koutis et al., 2011b). 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 (Koutis et al., 2011a; Koutis et al., 2012). 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., $\|x\| = \sqrt{x^T A x}$. 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 $O(m \log(1/\epsilon))$.

2.4 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.* (Teixeira et al., 2013) 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 (Doyle and Snell, 1984), 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 (Bollobas, 1998; Doyle and Snell, 1984) 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 (Doyle and Snell, 1984) 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.

2.5 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.* (Koutis et al., 2012; Koutis et al., 2011a) 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.

2.5.1 Tools

Existing algorithms for computing the SPANNING centrality are based on the celebrated Kirchoff's matrix-tree theorem (Harris et al., 2008; Tutte, 2001). The best known such algorithm has running time $O(mn^{3/2})$ (Teixeira et al., 2013), 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 (Doyle and Snell, 1984):

$$R = B L^{\dagger} B^T, \tag{2.1}$$

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 (2.1) 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 (2.1), 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 (Johnson and Lindenstrauss, 1982). 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 (Spielman and Srivastava, 2011). 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.* (Koutis et al., 2012; Koutis et al., 2011a), 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

$$(1 - \epsilon)^2 R(e) \leq \tilde{R}(e) \leq (1 + \epsilon)^2 R(e). \tag{2.2}$$

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 2.7.

2.5.2 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** $i = 1 \dots k$ **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 2.4, 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 (Tarjan, 1974), 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 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 $O(m)$ (Batagelj and Zaversnik, 2003). Our experiments (Section 2.7) 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.

2.6 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.

2.6.1 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 2·1. 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.

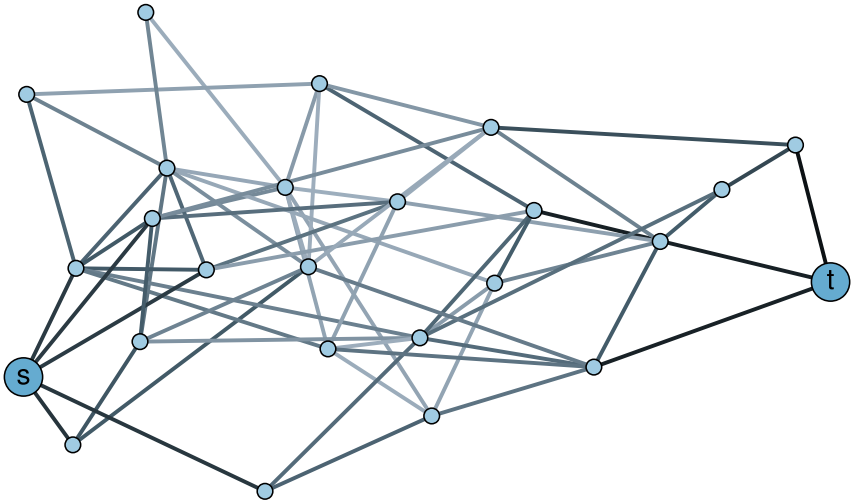


Figure 2·1: A network, viewed as an electrical resistive circuit. The thickness of an edge represents the amount of current it carries, if a battery is attached to nodes s and t.

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 (Brandes and Fleischer, 2005).

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* (Brandes, 2001). 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) (Newman, 2005). 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:

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

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 $\beta = 1$.

2.6.2 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,

$$f_{st}(u, v) = |x(u) - x(v)| \tag{2.3}$$

The algorithm of Brandes and Fleisher (Brandes and Fleischer, 2005): 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 (Brandes and Fleischer, 2005), 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 (Brandes and Fleischer, 2005) we define

$$\overline{\beta\text{-CFC}}(e=\{u,v\}) \triangleq \frac{1}{k}\sum_{(s,t)\in S_k} f_{st}(u,v), \tag{2.4}$$

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 (2.3). 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 2.7.4.

2.7 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 (Koutis et al., 2011b) 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

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

Table 2.1: Statistics of the collection of datasets used in our experiments.

the popular *networkx*[1] package (Hagberg et al., 2008). The code is available online[2].

We ran all our experiments on a machine with 4 *Intel Xeon E5-4617 @ 2.9GHz*, with $512GB$ 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 2.1 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 2.1 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.* (Batagelj and Zaversnik, 2003). 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

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

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.

2.7.1 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 (2.2)) 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 2.5.2.

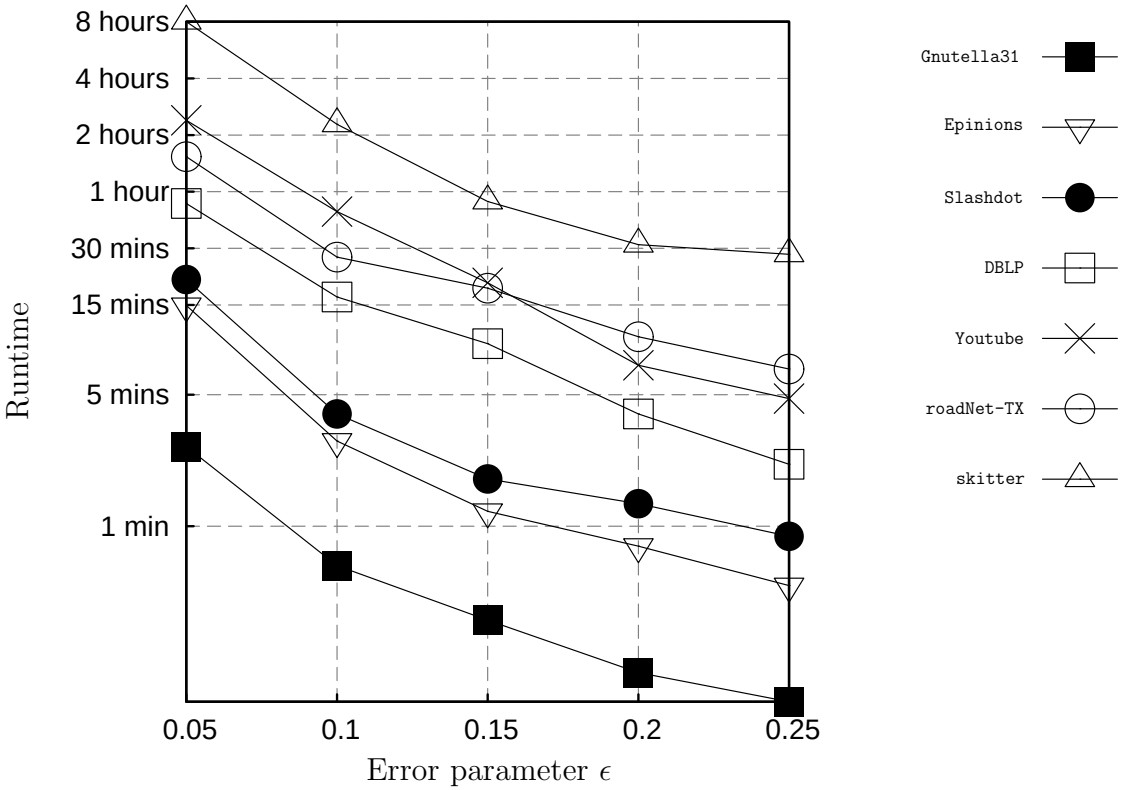


Figure 2·2: Accuracy-efficiency tradeoff; y-axis (logarithmic scale): running time of the `Fast-TreeC` algorithm; x-axis: error parameter ϵ.

The results are shown in Figure 2·2; 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 y-axis in Figure 2·2 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 2.5.2.

Figure 2·3 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 run-

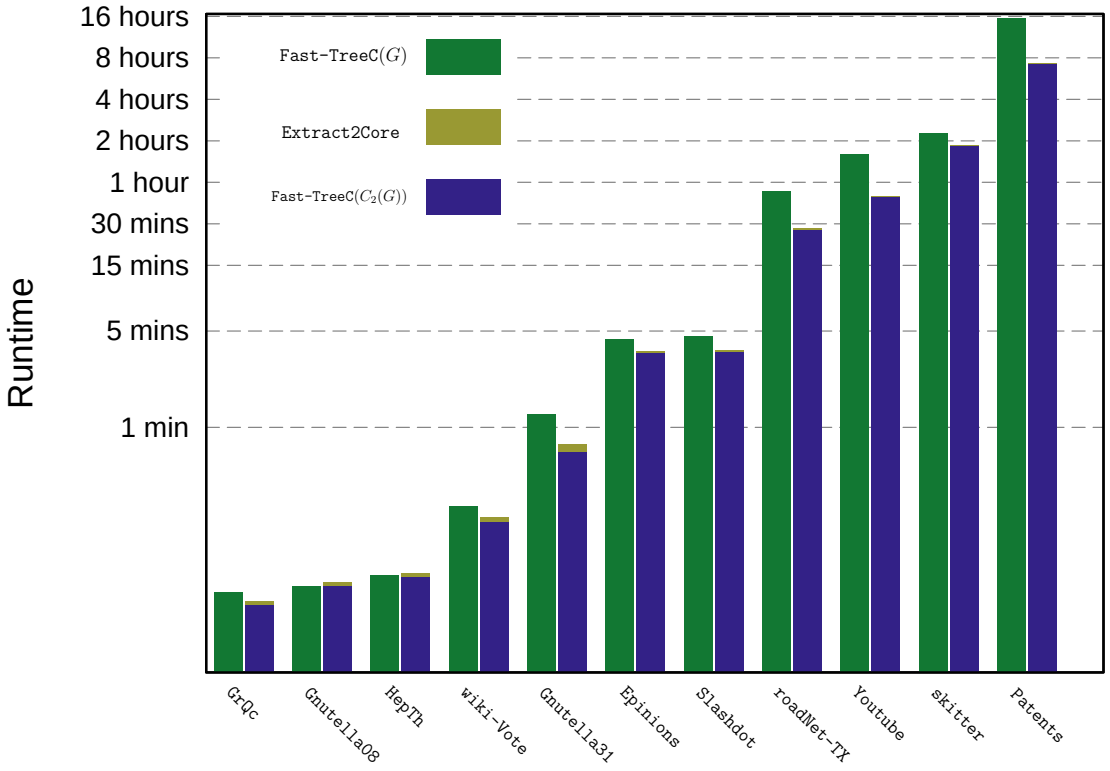


Figure 2·3: Limiting the computation on the 2-core shows a measurable improvement in the running time of `Fast-TreeC`.

ning 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.

2.7.2 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 (Brandes and Fleischer, 2005) and BETWEENNESS centrality (Brandes, 2001; Freeman, 1977), 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:

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

`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 2·4 and 2·5 show the noise resilience of SPANNING, CURRENTFLOW and BETWEENNESS centrality under edge addition. This is measured using both

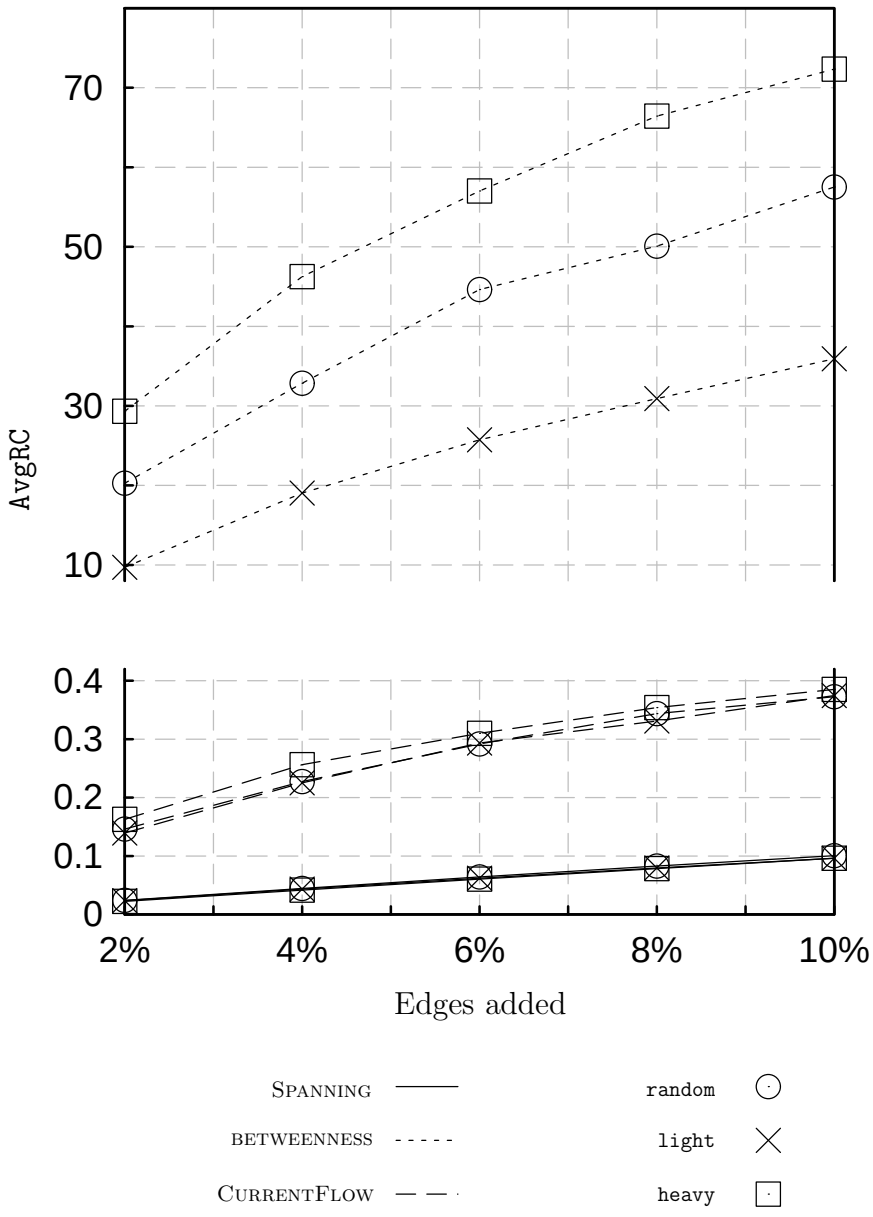


Figure 2·4: Average relative change in the edge importance scores. The x-axis shows the percentage of noisy edges added to the original `HepTh` graph.

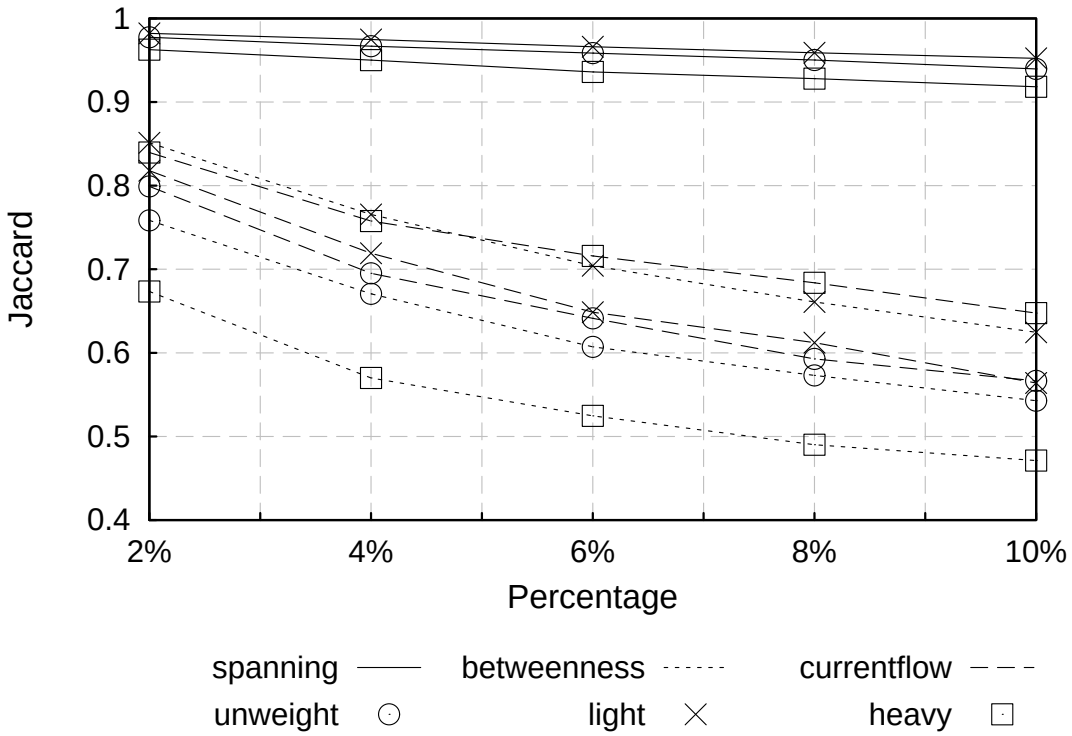


Figure 2·5: Jaccard similarity between the top 10%-scoring edges in the original and the noisy graph. Note how resilient the SPANNING centrality is to noise.

`AvgRC` (Fig. 2·4) as well as `JaccSim` (Fig. 2·5). 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 (Brandes and Fleischer, 2005) and BETWEENNESS (Brandes, 2001) (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 2·5, 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 2·4, SPANNING has, in the worst case, `AvgRC` 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 2·5 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

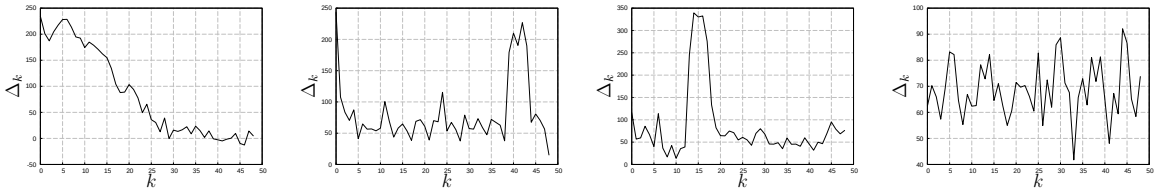
JaccSim has, in the worst case, score equal to 0.91 – which is very close to 1. The corresponding values for both BETWEENNESS and CURRENTFLOW are significantly lower, at 0.47 and 0.57 respectively. In addition to the above, we observed that, even when adding a 10% noise to the graph edges, more than 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.

2.7.3 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,



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

Figure 2·6: The values Δ_k as a function of k for the different importance measures.

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 (Kempe et al., 2003) 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 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 SPREAD(G_k). In order to quantify the influence that the set E_k has on the information-propagation process we compute:

$$\Delta_k = \text{SPREAD}(G) - \text{SPREAD}(G_k).$$

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 (Freeman, 1977). RANDOM simply assigns a random order on the edges in E.

Results: Figure 2·6 shows the values of Δ_k in the case of the `HepTh` network, for $k = 1, \ldots, 49$ 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 2·6a), 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 2·6b and 2·6c) 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 2·6b and 2·6c. 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 2·6a–2·6c is statistically significant.

2.7.4 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 2.6.

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 (2.4). 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 2.2, 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 2.2: Time until termination of `Fast-FlowC` and the exact algorithm. We terminate for $\tau < 0.02$.

Chapter 3

Absorbing random walk centrality

3.1 Introduction

A fundamental problem in graph mining is to identify the most central nodes in a graph. Numerous centrality measures have been proposed, including degree centrality, closeness centrality (Sabidussi, 1966), betweenness centrality (Freeman, 1977), random-walk centrality (Noh and Rieger, 2004), Katz centrality (Katz, 1953), and PageRank (Brin and Page, 1998).

In the interest of robustness many centrality measures use random walks: while the shortest-path distance between two nodes can change dramatically by inserting or deleting a single edge, distances based on random walks account for multiple paths and offer a more global view of the connectivity between two nodes. In this spirit, the random-walk centrality of *one* node with respect to *all nodes* of the graph is defined as the expected time needed to come across this node in a random walk that starts in any other node of the graph (Noh and Rieger, 2004).

In this chapter, we consider a measure that generalizes random-walk centrality for a *set* of nodes C with respect to a set of *query nodes* Q. Our centrality measure is defined as the expected length of a random walk that starts from any node in Q until it reaches any node in C — at which point the random walk is *"absorbed"* by C. Moreover, to allow for adjustable importance of query nodes in the centrality measure, we consider random walks *with restarts*, that occur with a fixed probability α at each step of the random walk. The resulting computational problem is to find

a set of k nodes C that optimizes this measure with respect to nodes Q, which are provided as input. We call this measure k *absorbing random-walk centrality* and the corresponding optimization problem k-ARW-CENTRALITY.

To motivate the k-ARW-CENTRALITY problem, let us consider the scenario of searching the Web graph and summarizing the search results. In this scenario, nodes of the graph correspond to webpages, edges between nodes correspond to links between pages, and the set of query nodes Q consists of all nodes that match a user query, i.e., all webpages that satisfy a keyword search. Assuming that the size of Q is large, the goal is to find the k most central nodes with respect to Q, and present those to the user.

It is clear that ordering the nodes of the graph by their individual random-walk centrality scores and taking the top-k set does not solve the k-ARW-CENTRALITY problem, as these nodes may all be located in the same "neighborhood" of the graph, and thus, may not provide a good absorbing set for the query. On the other hand, as the goal is to minimize the expected absorption time for walks starting at Q, the optimal solution to the k-ARW-CENTRALITY problem will be a set of k, both *centrally-placed* and *diverse*, nodes.

This observation has motivated researchers in the information-retrieval field to consider random walks with absorbing states in order to diversify web-search results (Zhu et al., 2007). However, despite the fact that similar problem definitions and algorithms have been considered earlier, the k-ARW-CENTRALITY problem has not been formally studied and there has not been a theoretical analysis of its properties.

Our key results in this chapter are the following: we show that the k-ARW-CENTRALITY problem is **NP**-hard, and we show that the k absorbing random-walk centrality measure is *monotone* and *supermodular*. The latter property allows us

to quantify the approximation guarantee obtained by a natural greedy algorithm, which has also been considered by previous work (Zhu et al., 2007). Furthermore, a naïve implementation of the greedy algorithm requires many expensive matrix inversions, which make the algorithm particularly slow. Part of our contribution is to show how to make use of the Sherman-Morrison inversion formula to implement the greedy algorithm with only one matrix inversion and more efficient matrix $\times$ vector multiplications.

Moreover, we explore the performance of faster, heuristic algorithms, aiming to identify methods that are faster than the greedy approach without significant loss in the quality of results. The heuristic algorithms we consider include the personalized PageRank algorithm (Brin and Page, 1998; Langville and Meyer, 2005) as well as algorithms based on spectral clustering (Von Luxburg, 2007). We find that, in practice, the personalized PageRank algorithm offers a very good trade-off between speed and quality.

The rest of the chapter is organized as follows. In Section 3.2, we overview previous work and discuss how it compares to ours. We define our problem in Section 3.3 and provide basic background results on absorbing random walks in Section 3.4. Our main technical contributions are given in Sections 3.4 and 3.5, where we characterize the complexity of the problem, and provide the details of the greedy algorithm and the heuristics we explore. We evaluate the performance of algorithms in Section 3.7, over a range of real-world graphs.

3.2 Related work

Many works in the literature explore ways to quantify the notion of node centrality on graphs (Boldi and Vigna, 2014). Some of the most commonly-used measures include the following: (*i*) *degree centrality*, where the centrality of a node is simply quantified

by its degree; (*ii*) *closeness centrality* (Leavitt, 1951; Sabidussi, 1966), defined as the average distance of a node from all other nodes on the graph; (*iii*) *betweenness centrality* (Freeman, 1977), defined as the number of shortest paths between pairs of nodes in the graph that pass through a given node; (*iv*) *eigenvector centrality*, defined as the stationary probability that a Markov chain on the graph visits a given node, with Katz centrality (Katz, 1953) and PageRank (Brin and Page, 1998) being two well-studied variants; and (*v*) *random-walk centrality* (Noh and Rieger, 2004), defined as the expected first passage time of a random walk from a given node, when it starts from a random node of the graph. The measure we study in this chapter generalizes the notion of *random-walk centrality* to a set of absorbing nodes.

Absorbing random walks have been used in previous work to select a *diverse* set of nodes from a graph. For example, an algorithm proposed by Zhu et al. (Zhu et al., 2007) selects nodes in the following manner: (*i*) the first node is selected based on its PageRank value and is set as absorbing; (*ii*) the next node to be selected is the node that maximizes the expected first-passage time from the already selected absorbing nodes. Our problem definition differs considerably from the one considered in that work, as in our work the expected first-passage times are always computed from the set of query nodes that are provided in the input, and not from the nodes that participate in the solution so far. In this respect, the greedy method proposed by Zhu et al. is not associated with a crisp problem definition.

Another conceptually related line of work aims to select a diverse subset of query results, mainly within the context of document retrieval (Agrawal et al., 2009; Angel and Koudas, 2011; Vieira et al., 2011). The goal, there, is to select k query results to optimize a function that quantifies the trade-off between relevance and diversity.

Our work is also remotely related to the problem studied by Leskovec et al. on *cost-effective outbreak detection* (Leskovec et al., 2007). One of the problems discussed

there is to select nodes in the network so that the detection time for a set of cascades is minimized. However, their work differs from ours on the fact that they consider as input a set of *cascades*, each one of finite size, while in our case the input consists of a set of query *nodes* and we consider a probabilistic model that generates random walk paths, of possibly infinite size.

3.3 Problem definition

We are given a graph $G = (V, E)$ over a set of nodes V and set of undirected edges E. The number of nodes $|V|$ is denoted by n and the number of edges $|E|$ by m. The input also includes a subset of nodes $Q \subseteq V$, to which we refer as the *query nodes*. As a special case, the set of query nodes Q may be equal to the whole set of nodes, i.e., $Q = V$.

Our goal is to find a set C of k nodes that are *central* with respect to the query nodes Q. For some applications it makes sense to restrict the central nodes to be only among the query nodes, while in other cases, the central nodes may include any node in V. To model those different scenarios, we consider a set of candidate nodes D, and require that the k central nodes should belong in this candidate set, i.e., $C \subseteq D$. Some of the cases include $D = Q$, $D = V$, or $D = V \setminus Q$, but it could also be that D is defined in some other way that does not involve Q. In general, we assume that D is given as input.

The *centrality* of a set of nodes C with respect to query nodes Q is based on the notion of absorbing random-walks and their expected length. More specifically, let us consider a random walk on the nodes V of the graph, that proceeds at discrete steps: the walk starts from a node $q \in Q$ and, at each step moves to a different node, following edges in G, until it arrives at some node in C. The *starting* node q of the walk is chosen according to a probability distribution $\mathbf{s}$. When the walk arrives at

a node $c \in C$ for the first time, it terminates, and we say that the random walk is *absorbed* by that node c. In the interest of generality, and to allow for adjustable importance of query nodes in the centrality measure, we also allow the random walk to restart. Restarts occur with a probability α at each step of the random walk, where α is a parameter that is specified as input to the problem. When restarting, the walk proceeds to a query node selected randomly according to $\mathbf{s}$. Intuitively, larger values of α favor nodes that are closer to nodes Q.

We are interested in the expected length (i.e., number of steps) of the walk that starts from a query node $q \in Q$ until it gets absorbed by some node in C, and we denote this expected length by $\mathrm{ac}^q_Q(C)$. We then define the *absorbing random-walk centrality* of a set of nodes C with respect to query nodes Q, by

$$\mathrm{ac}_Q(C) = \sum_{q \in Q} \mathbf{s}(q)\, \mathrm{ac}^q_Q(C).$$

The problem we consider in this chapter is the following.

Problem 1. (k-ARW-CENTRALITY) *We are given a graph $G = (V, E)$, a set of query nodes $Q \subseteq V$, a set of candidate nodes $D \subseteq V$, a starting probability distribution $\mathbf{s}$ over V such that $\mathbf{s}(v) = 0$ if $v \in V \setminus Q$, a restart probability α, and an integer k. We ask to find a set of k nodes $C \subseteq D$ that* minimizes $\mathrm{ac}_Q(C)$*, i.e., the expected length of a random walk that starts from Q and proceeds until it gets absorbed in some node in C.*

In cases where we have no reason to distinguish among the query nodes, we consider the uniform starting probability distribution $\mathbf{s}(q) = 1/|Q|$. In fact, for simplicity of exposition, hereinafter we focus on the case of uniform distribution. However, we note that all our definitions and techniques generalize naturally, not only to general starting probability distributions $\mathbf{s}(q)$, but also to *directed* and *weighted* graphs.

3.4 Absorbing random walks

In this section we review some relevant background on absorbing random walks. Specifically, we discuss how to calculate the objective function $\mathrm{ac}_Q(C)$ for Problem 1.

Let $\mathbf{P}$ be the transition matrix for a random walk, with $\mathbf{P}(i,j)$ expressing the probability that the random walk will move to node j given that it is currently at node i. Since random walks can only move to absorbing nodes C, but not away from them, we set $\mathbf{P}(c,c) = 1$ and $\mathbf{P}(c,j) = 0$, if $j \neq c$, for all absorbing nodes $c \in C$. The set $T = V \setminus C$ of non-absorbing nodes is called *transient*. If $N(i)$ are the neighbors of a node $i \in T$ and $d_i = |N(i)|$ its degree, the transition probabilities from node i to other nodes are

$$\mathbf{P}(i,j) = \begin{cases} \alpha\, \mathbf{s}(j) & \text{if } j \in Q \setminus N(i), \\ (1-\alpha)/d_i + \alpha\, \mathbf{s}(j) & \text{if } j \in N(i). \end{cases} \tag{3.1}$$

Here, $\mathbf{s}$ represents the starting probability vector. For example, for the uniform distribution over query nodes we have $\mathbf{s}(i) = 1/|Q|$ if $i \in Q$ and 0 otherwise. The transition matrix of the random walk can be written as follows

$$\mathbf{P} = \begin{pmatrix} \mathbf{P}_{TT} & \mathbf{P}_{TC} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}. \tag{3.2}$$

In the equation above, $\mathbf{I}$ is an $(n - |T|) \times (n - |T|)$ identity matrix and $\mathbf{0}$ a matrix with all its entries equal to 0; $\mathbf{P}_{TT}$ is the $|T| \times |T|$ sub-matrix of $\mathbf{P}$ that contains the transition probabilities between transient nodes; and $\mathbf{P}_{TC}$ is the $|T| \times |C|$ sub-matrix of $\mathbf{P}$ that contains the transition probabilities from transient to absorbing nodes.

The probability of the walk being on node j at exactly ℓ steps having started at node i, is given by the (i,j)-entry of the matrix $\mathbf{P}_{TT}^{\ell}$. Therefore, the expected total

number of times that the random walk visits node j having started from node i is given by the (i, j)-entry of the $|T| \times |T|$ matrix

$$\mathbf{F} = \sum_{\ell=0}^{\infty} \mathbf{P}_{TT}^{\ell} = (\mathbf{I} - \mathbf{P}_{TT})^{-1}, \tag{3.3}$$

which is known as the *fundamental matrix* of the absorbing random walk. Allowing the possibility to start the random walk at an absorbing node (and being absorbed immediately), we see that the expected length of a random walk that starts from node i and gets absorbed by the set C is given by the i-th element of the following $n \times 1$ vector

$$\mathbf{L} = \mathbf{L}_C = \begin{pmatrix} \mathbf{F} \\ \\ \mathbf{0} \end{pmatrix} \mathbf{1}, \tag{3.4}$$

where $\mathbf{1}$ is an $T \times 1$ vector of all 1s. We write $\mathbf{L} = \mathbf{L}_C$ to emphasize the dependence on the set of absorbing nodes C.

The expected number of steps when starting from a node in Q and until being absorbed by some node in C is then obtained by summing over all query nodes, i.e.,

$$\mathrm{ac}_Q(C) = \mathbf{s}^T \mathbf{L}_C. \tag{3.5}$$

3.4.1 Efficient computation of absorbing centrality

Equation (3.5) pinpoints the difficulty of the problem we consider: even computing the objective function $\mathrm{ac}_Q(C)$ for a candidate solution C requires an expensive matrix inversion; $\mathbf{F} = (\mathbf{I} - \mathbf{P}_{TT})^{-1}$. Furthermore, searching for the optimal set C involves an exponential number of candidate sets, while evaluating each one of them requires a matrix inversion.

In practice, we find that we can compute $\mathrm{ac}_Q(C)$ much faster approximately, as shown in Algorithm 3. The algorithm follows from the infinite-sum expansion of

Algorithm 3 ApproximateAC

Input: Transition matrix $\mathbf{P}_{TT}$, threshold ϵ, starting probabilities $\mathbf{s}$
Output: Absorbing centrality ac_Q

$\mathbf{x_0} \leftarrow \mathbf{s}^T$
$\delta \leftarrow \mathbf{x_0} \cdot \mathbf{1}$
$\mathrm{ac} \leftarrow \delta$
$\ell \leftarrow 0$
while $\delta < \epsilon$ **do**
    $\mathbf{x}_{\ell+1} \leftarrow \mathbf{x}_\ell \begin{pmatrix} \mathbf{P}_{TT} \\ \mathbf{0} \end{pmatrix}$
    $\delta \leftarrow \mathbf{x}_{\ell+1} \cdot \mathbf{1}$
    $\mathrm{ac} \leftarrow \mathrm{ac} + \delta$
    $\ell \leftarrow \ell + 1$
return ac

Equation (3.5).

$$\begin{aligned}
\mathrm{ac}_Q(C) = \mathbf{s}^T \mathbf{L}_C = \mathbf{s}^T \begin{pmatrix} \mathbf{F} \\ \mathbf{0} \end{pmatrix} \mathbf{1} &= \mathbf{s}^T \begin{pmatrix} \sum_{\ell=0}^{\infty} \mathbf{P}_{TT}^{\ell} \\ \mathbf{0} \end{pmatrix} \mathbf{1} \\
= \mathbf{s}^T \sum_{\ell=0}^{\infty} \begin{pmatrix} \mathbf{P}_{TT}^{\ell} \\ \mathbf{0} \end{pmatrix} \mathbf{1} &= \left(\sum_{\ell=0}^{\infty} \mathbf{s}^T \begin{pmatrix} \mathbf{P}_{TT}^{\ell} \\ \mathbf{0} \end{pmatrix} \right) \mathbf{1} \\
= \left(\sum_{\ell=0}^{\infty} \mathbf{x}_\ell \right) \mathbf{1} &= \sum_{\ell=0}^{\infty} \mathbf{x}_\ell \mathbf{1},
\end{aligned}$$

with

$$\mathbf{x}_0 = \mathbf{s}^T \quad \text{and} \quad \mathbf{x}_{\ell+1} = \mathbf{x}_\ell \begin{pmatrix} \mathbf{P}_{TT} \\ \mathbf{0} \end{pmatrix}. \tag{3.6}$$

Note that computing each vector $\mathbf{x}_\ell$ requires time $\mathcal{O}(n^2)$. Algorithm 3 terminates when the increase of the sum due to the latest term falls below a pre-defined threshold ϵ.

3.5 Problem characterization

We now study the k-ARW-CENTRALITY problem in more detail. In particular, we show that the function ac_Q is monotone and supermodular, a property that is used later to provide an approximation guarantee for the greedy algorithm. We also show that k-ARW-CENTRALITY is **NP**-hard.

Recall that a function $f : 2^V \to \mathbb{R}$ over subsets of a ground set V is *submodular* if it has the *diminishing returns* property

$$f(Y \cup \{u\}) - f(Y) \leq f(X \cup \{u\}) - f(X), \tag{3.7}$$

for all $X \subseteq Y \subseteq V$ and $u \notin Y$. The function f is *supermodular* if $-f$ is submodular. Submodularity (and supermodularity) is a very useful property for designing algorithms. For instance, minimizing a submodular function is a polynomial-time solvable problem, while the maximization problem is typically amenable to approximation algorithms, the exact guarantee of which depends on other properties of the function and requirements of the problem, e.g., monotonicity, matroid constraints, etc.

Even though the objective function $\mathrm{ac}_Q(C)$ is given in closed-form by Equation (3.5), to prove its properties we find it more convenient to work with its descriptive definition, namely, $\mathrm{ac}_Q(C)$ being the expected length for a random walk starting at nodes of Q before being absorbed at nodes of C.

For the rest of this section we consider that the set of query nodes Q is fixed, and for simplicity we write $\mathrm{ac} = \mathrm{ac}_Q$.

Proposition 1 (Monotonicity)**.** *For all $X \subseteq Y \subseteq V$ it is* $\mathrm{ac}(Y) \leq \mathrm{ac}(X)$.

Proof. Write G_X for the input graph G where the set X are absorbing nodes. Define G_Y similarly. Let $Z = Y \setminus X$. Consider a path p in G_X drawn from the distribution induced by the random walks on G_X. Let $\Pr[p]$ be the probability of the path and

$\ell(p)$ its length. Let $\mathcal{P}(X)$ and $\mathcal{P}(Y)$ be the set of paths on G_X and G_Y. Finally, let $\mathcal{P}(Z, X)$ be the set of paths on G_X that pass from Z, and $\mathcal{P}(\overline{Z}, X)$ the set of paths on G_X that do *not* pass from Z. We have

$$\begin{aligned}
\mathrm{ac}(X) &= \sum_{p \in \mathcal{P}(X)} \Pr[p]\, \ell(p) \\
&= \sum_{p \in \mathcal{P}(\overline{Z}, X)} \Pr[p]\, \ell(p) + \sum_{p \in \mathcal{P}(Z, X)} \Pr[p]\, \ell(p) \\
&\geq \sum_{p \in \mathcal{P}(Y)} \Pr[p]\, \ell(p) \\
&= \mathrm{ac}(Y),
\end{aligned}$$

where the inequality comes from the fact that a path in G_X passing from Z and being absorbed by X corresponds to a shorter path in G_Y being absorbed by Y. □

This proposition states that absorption time decreases with more absorbing nodes. Next we show that the absorbing random-walk centrality measure $\mathrm{ac}(\cdot)$ is supermodular.

Proposition 2 (Supermodularity)**.** *For all sets $X \subseteq Y \subseteq V$ and $u \notin Y$ it is*

$$\mathrm{ac}(X) - \mathrm{ac}(X \cup \{u\}) \geq \mathrm{ac}(Y) - \mathrm{ac}(Y \cup \{u\}). \tag{3.8}$$

Proof. Given an instantiation of a random walk, we define the following propositions for any pair of nodes $i, j \in V$, non-negative integer ℓ, and set of nodes Z:

$A^{\ell}_{i,j}(Z)$: The random walk started at node i and visited node j after exactly ℓ steps, without visiting any node in set Z.

$B^{\ell}_{i,j}(Z, u)$: The random walk started at node i and visited node j after exactly ℓ steps, having previously visited node u but without visiting any node in the set Z.

It is easy to see that the set of random walks for which $A^{\ell}_{i,j}(Z)$ is `true` can be partitioned into those that visited u within the first ℓ steps and those that did not. Therefore, the probability that proposition $A^{\ell}_{i,j}(Z)$ is `true` for any instantiation of a random walk generated by our model is equal to

$$\Pr\left[A^{\ell}_{i,j}(Z)\right] = \Pr\left[A^{\ell}_{i,j}(Z \cup \{u\})\right] + \Pr\left[B^{\ell}_{i,j}(Z, u)\right]. \tag{3.9}$$

Now, let $\mathbf{\Lambda}(Z)$ be the number of steps for a random walk to reach the nodes in Z. $\mathbf{\Lambda}(Z)$ is a random variable and its expected value over all random walks generated by our model is equal to $\mathrm{ac}(Z)$. Note that the proposition $\mathbf{\Lambda}(Z) \geq \ell + 1$ is `true` for a given instantiation of a random walk only if there is a pair of nodes $q \in Q$ and $j \in V \setminus Z$, for which the proposition $A^{\ell}_{q,j}(Z)$ is `true`. Therefore,

$$\Pr\left[\mathbf{\Lambda}(Z) \geq \ell + 1\right] = \sum_{q \in Q} \sum_{j \in V \setminus Z} \Pr\left[A^{\ell}_{q,j}(Z)\right]. \tag{3.10}$$

From the above, it is easy to calculate $\mathrm{ac}(Z)$ as

$$\begin{aligned}
\mathrm{ac}(Z) &= E[\mathbf{\Lambda}(Z)] \\
&= \sum_{\ell=0}^{\infty} \ell \, \Pr\left[\mathbf{\Lambda}(Z) = \ell\right] \\
&= \sum_{\ell=1}^{\infty} \Pr\left[\mathbf{\Lambda}(Z) \geq \ell\right] \\
&= \sum_{\ell=0}^{\infty} \Pr\left[\mathbf{\Lambda}(Z) \geq \ell + 1\right] \\
&= \sum_{\ell=0}^{\infty} \sum_{q \in Q} \sum_{j \in V \setminus Z} \Pr\left[A^{\ell}_{q,j}(Z)\right]. \qquad (3.11)
\end{aligned}$$

The final property we will need is the observation that, for $X \subseteq Y$, $B^{\ell}_{i,j}(Y, u)$ implies $B^{\ell}_{i,j}(X, u)$ and thus

$$\Pr\left[B^{\ell}_{i,j}(X, u)\right] \geq \Pr\left[B^{\ell}_{i,j}(Y, u)\right]. \tag{3.12}$$

By using Equation (3.11), the Inequality (3.8) can be rewritten as

$$\begin{aligned}
&\sum_{\ell=0}^{\infty}\sum_{q\in Q}\sum_{j\in V\setminus X}\Pr\left[A_{q,j}^{\ell}(X)\right] - \\
&\qquad\sum_{\ell=0}^{\infty}\sum_{q\in Q}\sum_{j\in V\setminus\{X\cup\{u\}\}}\Pr\left[A_{q,j}^{\ell}(X\cup\{u\})\right] \\
&\geq \sum_{\ell=0}^{\infty}\sum_{q\in Q}\sum_{j\in V\setminus Y}\Pr\left[A_{q,j}^{\ell}(Y)\right] - \\
&\qquad\sum_{\ell=0}^{\infty}\sum_{q\in Q}\sum_{j\in V\setminus\{Y\cup\{u\}\}}\Pr\left[A_{q,j}^{\ell}(Y\cup\{u\})\right]. \qquad (3.13)
\end{aligned}$$

We only need to show that the inequality holds for an arbitrary value of ℓ and $q \in Q$, that is

$$\begin{aligned}
&\sum_{j\in V\setminus X}\Pr\left[A_{q,j}^{\ell}(X)\right] - \sum_{j\in V\setminus\{X\cup\{u\}\}}\Pr\left[A_{q,j}^{\ell}(X\cup\{u\})\right] \geq \\
&\sum_{j\in V\setminus Y}\Pr\left[A_{q,j}^{\ell}(Y)\right] - \sum_{j\in V\setminus\{Y\cup\{u\}\}}\Pr\left[A_{q,j}^{\ell}(Y\cup\{u\})\right]. \qquad (3.14)
\end{aligned}$$

Notice that $\Pr\left[A_{i,u}^{\ell}(Y\cup\{u\})\right] = 0$, so we can rewrite the above inequality as

$$\begin{aligned}
&\sum_{j\in V\setminus X}\Pr\left[A_{q,j}^{\ell}(X)\right] - \sum_{j\in V\setminus X}\Pr\left[A_{q,j}^{\ell}(X\cup\{u\})\right] \geq \\
&\sum_{j\in V\setminus Y}\Pr\left[A_{q,j}^{\ell}(Y)\right] - \sum_{j\in V\setminus Y}\Pr\left[A_{q,j}^{\ell}(Y\cup\{u\})\right]. \qquad (3.15)
\end{aligned}$$

To show the latter inequality we start from the left hand side and use Inequality

(3.12). We have

$$\begin{aligned}
&\sum_{j\in V\setminus X} \Pr\left[A^{\ell}_{i,j}(X)\right] - \sum_{j\in V\setminus X} \Pr\left[A^{\ell}_{i,j}(X\cup\{u\})\right] \\
&\quad= \sum_{j\in V\setminus X} \Pr\left[B^{\ell}_{i,j}(X,u)\right] \\
&\quad\geq \sum_{j\in V\setminus Y} \Pr\left[B^{\ell}_{i,j}(Y,u)\right] \\
&\quad= \sum_{j\in V\setminus Y} \Pr\left[A^{\ell}_{i,j}(Y)\right] - \sum_{j\in V\setminus Y} \Pr\left[A^{\ell}_{i,j}(Y\cup\{u\})\right],
\end{aligned}$$

which completes the proof. □

Finally, we establish the hardness of k absorbing centrality, defined in Problem 1.

Theorem 1. *The k-ARW-CENTRALITY problem is* **NP***-hard.*

Proof. We obtain a reduction from the VERTEXCOVER problem (Garey and Johnson, 1990). An instance of the VERTEXCOVER problem is specified by a graph $G = (V, E)$ and an integer k, and asks whether there exists a set of nodes $C \subseteq V$ such that $|C| \leq k$ and C is a vertex cover, (i.e., for every $(i, j) \in E$ it is $\{i, j\} \cap C \neq \emptyset$). Let $|V| = n$.

Given an instance of the VERTEXCOVER problem, we construct an instance of the decision version of k-ARW-CENTRALITY by taking the same graph $G = (V, E)$ with query nodes $Q = V$ and asking whether there is a set of absorbing nodes C such that $|C| \leq k$ and $\mathrm{ac}_{Q}(C) \leq 1 - \frac{k}{n}$.

We will show that C is a solution for VERTEXCOVER if and only if $\mathrm{ac}_{Q}(C) \leq 1-\frac{k}{n}$.

Assuming first that C is a vertex cover. Consider a random walk starting uniformly at random from a node $v \in Q = V$. If $v \in C$ then the length of the walk will be 0, as the walk will be absorbed immediately. This happens with probability $|C|/|V| = k/n$. Otherwise, if $v \notin C$ the length of the walk will be 1, as the walk will be absorbed in the next step (since C is a vertex cover all the neighbors of v need to belong in C). This happens with the rest of the probability $1 - k/n$. Thus, the expected length of the random walk is

$$\mathrm{ac}_{Q}(C) = 0 \cdot \frac{k}{n} + 1 \cdot \left(1 - \frac{k}{n}\right) = 1 - \frac{k}{n} \tag{3.16}$$

Conversely, assume that C is not a vertex cover for G. Then, there should be an uncovered edge (u, v). A random walk that starts in u and then goes to v (or starts

in v and then goes to u) will have length at least 2, and this happens with probability at least $\frac{2}{n}\frac{1}{d_{\max}} \geq \frac{2}{n^2}$. Then, following a similar reasoning as in the previous case, we have

$$\begin{aligned} \mathrm{ac}_Q(C) &= \sum_{k=0}^{\infty} k \Pr(\text{absorbed in exactly } k \text{ steps}) \\ &= \sum_{k=1}^{\infty} \Pr(\text{absorbed after at least } k \text{ steps}) \\ &\geq \left(1 - \frac{k}{n}\right) + \frac{2}{n^2} > 1 - \frac{k}{n}. \end{aligned} \tag{3.17}$$

□

3.6 Algorithms

This section presents algorithms to solve the k-ARW-CENTRALITY problem. In all cases, the set of query nodes $Q \subseteq V$ is given as input, along with a set of candidate nodes $D \subseteq V$ and the restart probability α.

3.6.1 Greedy approach

The first algorithm is a standard greedy algorithm, denoted Greedy, which exploits the supermodularity of the absorbing random-walk centrality measure. It starts with the result set C equal to the empty set, and iteratively adds a node from the set of candidate nodes D, until k nodes are added. In each iteration the node added in the set C is the one that brings the largest improvement to ac_Q.

As shown before, the objective function to be minimized, i.e., ac_Q, is supermodular and monotonically decreasing. The Greedy algorithm is not an approximation algorithm for this minimization problem. However, it can be shown to provide an approximation guarantee for maximizing the *absorbing centrality gain* measure, defined below.

Definition 2 (Absorbing centrality gain)**.** *Given a graph G, a set of query nodes Q,*

and a set of candidate nodes D, the absorbing centrality gain of a set of nodes $C \subseteq D$ is defined as

$$\mathrm{acg}_Q(C) = m_Q - \mathrm{ac}_Q(C),$$

where $m_Q = \min_{v \in D}\{\mathrm{ac}_Q(\{v\})\}$.

Justification of the gain function. The reason to define the absorbing centrality gain is to turn our problem into a submodular-maximization problem so that we can apply standard approximation-theory results and show that the greedy algorithm provides a constant-factor approximation guarantee. The *shift* m_Q quantifies the absorbing centrality of the best single node in the candidate set. Thus, the value of $\mathrm{acg}_Q(C)$ expresses how much we gain in expected random-walk length when we use the set C as absorbing nodes compared to when we use the best single node. Our goal is to maximize this gain.

Observe that the gain function acg_Q is not non-negative everywhere. Take for example any node u such that $\mathrm{ac}_Q(\{u\}) > m_Q$. Then, $\mathrm{acg}_Q(\{u\}) < 0$. Note also that we could have obtained a non-negative gain function by defining gain with respect to the *worst* single node, instead of the best. In other words, the gain function $\mathrm{acg}'_Q(C) = M_Q - \mathrm{ac}_Q(C)$, with $M_Q = \max_{v \in D}\{\mathrm{ac}_Q(\{v\})\}$, is non-negative everywhere.

Nevertheless, the reason we use the gain function acg_Q instead of acg'_Q is that acg'_Q takes much larger values than acg_Q, and thus, a multiplicative approximation guarantee on acg'_Q is a weaker result than a multiplicative approximation guarantee on acg_Q. On the other hand, our definition of acg_Q creates a technical difficulty with the approximation guarantee, that is defined for non-negative functions. Luckily, this difficulty can be overcome easily by noting that, due to the monotonicity of acg_Q, for any $k > 1$, the optimal solution of the function acg_Q, as well as the solution returned by Greedy, are both non-negative.

Approximation guarantee. The fact that the Greedy algorithm gives an approximation guarantee to the problem of maximizing absorbing centrality gain is a standard

result from the theory of submodular functions.

Proposition 3. *The function* acg_Q *is monotonically increasing, and submodular.*

Proposition 4. *Let* $k > 1$*. For the problem of finding a set* $C \subseteq D$ *with* $|C| \leq k$*, such that* $\mathrm{acg}_Q(C)$ *is maximized, the* Greedy *algorithm gives a* $\left(1 - \frac{1}{e}\right)$*-approximation guarantee.*

We now discuss the complexity of the Greedy algorithm. A naïve implementation requires computing the absorbing centrality $\mathrm{ac}_Q(C)$ using Equation (3.5) for each set C that needs to be evaluated during the execution of the algorithm. However, applying Equation (3.5) involves a matrix inversion, which is a very expensive operation. Furthermore, the number of times that we need to evaluate $\mathrm{ac}_Q(C)$ is $\mathcal{O}(k|D|)$, as for each iteration of the greedy we need to evaluate the improvement over the current set of each of the $\mathcal{O}(|D|)$ candidates. The number of candidates can be very large, e.g., $|D| = n$, yielding an $\mathcal{O}(kn^4)$ algorithm, which is prohibitively expensive.

We can show, however, that we can execute Greedy significantly more efficiently. Specifically, we can prove the following two propositions.

Proposition 5. *Let* C_{i-1} *be a set of* $i-1$ *absorbing nodes,* $\mathbf{P}_{i-1}$ *the corresponding transition matrix, and let* $\mathbf{F}_{i-1} = (\mathbf{I} - \mathbf{P}_{i-1})^{-1}$*. Let* $C_i = C_{i-1} \cup \{u\}$*. Given* $\mathbf{F}_{i-1}$ *the value* $\mathrm{ac}_Q(C_i)$ *can be computed in* $\mathcal{O}(n^2)$*.*

The proof makes use of the following lemma.

Lemma 2 (Sherman-Morrison Formula (Golub and Van Loan, 2012))**.** *Let* $\mathbf{M}$ *be a square* $n \times n$ *invertible matrix and* $\mathbf{M}^{-1}$ *its inverse. Moreover, let* $\mathbf{a}$ *and* $\mathbf{b}$ *be any two column vectors of size* n*. Then, the following equation holds*

$$(\mathbf{M} + \mathbf{a}\mathbf{b}^T)^{-1} = \mathbf{M}^{-1} - \mathbf{M}^{-1}\mathbf{a}\mathbf{b}^T\mathbf{M}^{-1}/(1 + \mathbf{b}^T\mathbf{M}^{-1}\mathbf{a}).$$

Proof. Without loss of generality, let the set of absorbing nodes be $C_{i-1} = \{1, 2, \ldots, i-1\}$. The expected number of steps before absorption is given by the formulas

$$\mathrm{ac}_Q(C_{i-1}) = \mathbf{s}_Q^T \mathbf{F}_{i-1} \mathbf{1},$$
$$\text{with } \mathbf{F}_{i-1} = \mathbf{A}_{i-1}^{-1} \text{ and } \mathbf{A}_{i-1} = \mathbf{I} - \mathbf{P}_{i-1}.$$

We proceed to show how to increase the set of absorbing nodes by one and calculate the new absorption time by updating $\mathbf{F}_{i-1}$ in $\mathcal{O}(n^2)$. Without loss of generality, suppose we add node i to the absorbing nodes C_{i-1}, so that

$$C_i = C_{i-1} \cup \{i\} = \{1, 2, \ldots, i-1, i\}.$$

Let $\mathbf{P}_i$ be the transition matrix over G with absorbing nodes C_i. Like before, the expected absorption time by nodes C_i is given by the formulas

$$\mathrm{ac}_Q(C_i) = \mathbf{s}_Q^T \mathbf{F}_i \mathbf{1},$$
$$\text{with } \mathbf{F}_i = \mathbf{A}_i^{-1} \text{ and } \mathbf{A}_i = \mathbf{I} - \mathbf{P}_i.$$

Notice that

$$\begin{aligned} \mathbf{A}_i - \mathbf{A}_{i-1} &= (\mathbf{I} - \mathbf{P}_i) - (\mathbf{I} - \mathbf{P}_{i-1}) = \mathbf{P}_{i-1} - \mathbf{P}_i \\ &= \begin{bmatrix} \mathbf{0}_{(i-1)\times n} \\ p_{i,1} \ldots p_{i,n} \\ \mathbf{0}_{(n-i)\times n} \end{bmatrix} = \mathbf{a}\mathbf{b}^T \end{aligned}$$

where $p_{i,j}$ denotes the transition probability from node i to node j in transition matrix $\mathbf{P}_{i-1}$, and the column-vectors $\mathbf{a}$ and $\mathbf{b}$ are defined as

$$\begin{aligned} \mathbf{a} &= [\overbrace{0 \ldots 0}^{i-1}\ 1\ \overbrace{0 \ldots 0}^{n-i}], \text{ and} \\ \mathbf{b} &= [p_{i,1} \ldots p_{i,n}]. \end{aligned}$$

By a direct application of Lemma 2, it is easy to see that we can compute $\mathbf{F}_i$ from $\mathbf{F}_{i-1}$ with the following formula, at a cost of $\mathcal{O}(n^2)$ operations.

$$\mathbf{F}_i = \mathbf{F}_{i-1} - (\mathbf{F}_{i-1}\mathbf{a})(\mathbf{b}^T\mathbf{F}_{i-1})/(1 + \mathbf{b}^T(\mathbf{F}_{i-1}\mathbf{a}))$$

We have thus shown that, given $\mathbf{F}_{i-1}$, we can compute $\mathbf{F}_i$, and therefore $\mathrm{ac}_Q(C_i)$ as

well, in $\mathcal{O}(n^2)$. □

Proposition 6. *Let C be a set of absorbing nodes, $\mathbf{P}$ the corresponding transition matrix, and $\mathbf{F} = (\mathbf{I} - \mathbf{P})^{-1}$. Let $C' = C - \{v\} \cup \{u\}$, $u, v \in C$. Given $\mathbf{F}$ the value $\mathrm{ac}_Q(C')$ can be computed in time $\mathcal{O}(n^2)$.*

Proof. The proof is similar to the proof of Proposition 5. Without loss of generality, let the two sets of absorbing nodes be

$$\begin{aligned} C &= \{1, 2, \ldots, i-1, i\}, \text{ and} \\ C' &= \{1, 2, \ldots, i-1, i+1\}. \end{aligned}$$

Let $\mathbf{P}'$ be the transition matrix with absorbing nodes C'. The absorbing centrality for the two sets of absorbing nodes C and C' is expressed as a function of the following two matrices

$$\mathbf{F} = \mathbf{A}^{-1}, \text{ with } \mathbf{A} = \mathbf{I} - \mathbf{P}, \text{ and}$$

$$\mathbf{F}' = \mathbf{A}'^{-1}, \text{ with } \mathbf{A}' = (\mathbf{I} - \mathbf{P}').$$

Notice that

$$\begin{aligned} \mathbf{A}' - \mathbf{A} &= (\mathbf{I} - \mathbf{P}') - (\mathbf{I} - \mathbf{P}) = \mathbf{P} - \mathbf{P}' \\ &= \begin{bmatrix} \mathbf{0}_{(i-1)\times n} \\ -p_{i,1} \ \ldots \ -p_{i,n} \\ p_{i+1,0} \cdots \ p_{i+1,n} \\ \mathbf{0}_{(n-i-1)\times n} \end{bmatrix} = \mathbf{a}_2\mathbf{b}_2^T - \mathbf{a}_1\mathbf{b}_1^T \end{aligned}$$

where $p_{i,j}$ denotes the transition probability from node i to node j in a transition matrix $\mathbf{P}_0$ where neither node i or $i+1$ is absorbing, and the column-vectors $\mathbf{a}_1$, $\mathbf{b}_1$, $\mathbf{a}_2$, $\mathbf{b}_2$ are defined as

$$\begin{aligned} \mathbf{a}_1 &= [\overbrace{0 \ldots 0}^{i-1} \ 1 \ 0 \ \overbrace{0 \ldots 0}^{n-i-1}] \\ \mathbf{b}_1 &= [p_{i,1} \ \ldots \ p_{i,n}] \\ \mathbf{a}_2 &= [\overbrace{0 \ldots 0}^{i-1} \ 0 \ 1 \ \overbrace{0 \ldots 0}^{n-i-1}] \\ \mathbf{b}_2 &= [p_{i+1,1} \ldots \ p_{i+1,n}]. \end{aligned}$$

Algorithm 4 Greedy

Input: graph G, query nodes Q, candidates D, $k \geq 1$
Output: a set of k nodes C

Compute $\text{ac}_Q(\{v\})$ for arbitrary $v \in D$
For each $u \in (D - \{v\})$, use Prop.6 to compute $\text{ac}_Q(u)$
Select $u_1 \in D$ s.t. $u_1 \leftarrow \arg\max_{u \in D} \text{ac}_Q(u)$
Initialize solution $C \leftarrow \{u_1\}$
for $i = 2..k$ **do**
    For each $u \in D$, use Prop.5 to compute $\text{ac}_Q(C \cup \{u\})$
    Select $u_i \in D$ s.t. $u_i \leftarrow \arg\max_{u_i \in (D-C)} \text{ac}_Q(C \cup \{u\})$
    Update solution $C \leftarrow C \cup \{u_i\}$
return C

By an argument similar with the one we made in the proof of Proposition 5, we can compute $\mathbf{F}'$ in the following two steps from $\mathbf{F}$, each costing $\mathcal{O}(n^2)$ operations for the provided parenthesization

$$\begin{aligned} \mathbf{Z} &= \mathbf{F} - (\mathbf{F}\mathbf{a}_2)(\mathbf{b}_2^T\mathbf{F})/(1 + \mathbf{b}_2^T(\mathbf{F}\mathbf{a}_2)), \\ \mathbf{F}' &= \mathbf{Z} + (\mathbf{Z}\mathbf{a}_1)(\mathbf{b}_1^T\mathbf{Z})/(1 + \mathbf{b}_1^T(\mathbf{Z}\mathbf{a}_1)). \end{aligned}$$

We have thus shown that, given $\mathbf{F}$, we can compute $\mathbf{F}'$, and therefore $\text{ac}_Q(C')$ as well, in time $\mathcal{O}(n^2)$. □

Proposition 5 implies that in order to compute $\text{ac}_Q(C_i)$ for absorbing nodes C_i in $\mathcal{O}(n^2)$, it is enough to maintain the matrix $\mathbf{F}_{i-1}$, computed in the previous step of the greedy algorithm for absorbing nodes C_{i-1}. Proposition 6, on the other hand, implies that we can compute the absorbing centrality of each set of absorbing nodes of a fixed size i in $\mathcal{O}(n^2)$, given the matrix $\mathbf{F}$, which is computed for one arbitrary set of absorbing nodes C of size i. Combined, the two propositions above yield a greedy algorithm that runs in $\mathcal{O}(kn^3)$ and offers the approximation guarantee discussed above. We outline it as Algorithm 4.

Practical speed-up. We found that the following heuristic lets us speed-up Greedy even further, with no significant loss in the quality of results. To select the first node for the solution set C (see Algorithm 4), we calculate the *PageRank* values of all

nodes in D and evaluate ac_Q only for the $t << k$ nodes with highest PageRank score, where t is a fixed parameter. In what follows, we will be using this heuristic version of Greedy, unless explicitly stated otherwise.

3.6.2 Efficient heuristics

Even though Greedy runs in polynomial time, it can be quite inefficient when employed on moderately sized datasets (more than some tens of thousands of nodes). We thus describe algorithms that we study as efficient heuristics for the problem. These algorithms do not offer guarantee for their performance.

Spectral methods have been used extensively for the problem of graph partitioning. Motivated by the wide applicability of this family of algorithms, here we explore three spectral algorithms: SpectralQ, SpectralC, and SpectralD. We start by a brief overview of the spectral method; a comprehensive presentation can be found in the tutorial by von Luxburg (Von Luxburg, 2007).

The main idea of spectral approaches is to project the original graph into a low-dimensional Euclidean space so that distances between nodes in the graph correspond to Euclidean distances between the corresponding projected points. A standard spectral embedding method, proposed by Shi and Malik (Shi and Malik, 2000), uses the "random-walk" *Laplacian* matrix $\mathbf{L}_G = \mathbf{I} - \mathbf{D}^{-1}\mathbf{A}$ of a graph G, where $\mathbf{A}$ is the adjacency matrix of the graph, and forms the matrix $\mathbf{U} = [u_2, \ldots, u_{d+1}]$ whose columns are the eigenvectors of $\mathbf{L}_G$ that correspond to the smallest eigenvalues $\lambda_2 \leq \ldots \leq \lambda_{d+1}$, with d being the target dimension of the projection. The spectral embedding is then defined by mapping the i-th node of the graph to a point in $\mathbb{R}^d$, which is the i-row of the matrix $\mathbf{U}$.

The algorithms we explore are adaptations of the spectral method. They all start by computing the spectral embedding $\phi : V \to \mathbb{R}^d$, as described above, and then, proceed as follows:

SpectralQ performs k-means clustering on the embeddings of the *query nodes*, where k is the desired size of the result set. Subsequently, it selects *candidate nodes* that are close to the computed centroids. Specifically, if s_i is the size of the i-th cluster, then k_i candidate nodes are selected whose embedding is the nearest to the i-th centroid. The number k_i is selected so that $k_i \propto s_i$ and $\sum k_i = k$.

SpectralC is similar to SpectralQ, but it performs the k-means clustering on the embeddings of the *candidate nodes*, instead of the query nodes.

SpectralD performs k-means clustering on the embeddings of the *query nodes*, where k is the desired result-set size. Then, it selects the k candidate nodes whose embeddings minimize the sum of squared ℓ_2-distances from the centroids, with no consideration of the relative sizes of the clusters.

Personalized Pagerank (PPR). This is the standard Pagerank (Brin and Page, 1998) algorithm with a damping factor equal to the restart probability α of the random walk and personalization probabilities equal to the start probabilities $\mathbf{s}(q)$. Algorithm PPR returns the k nodes with highest PageRank values.

Degree and distance centrality. Finally, we consider the standard degree and distance centrality measures.

Degree returns the k highest-degree nodes. Note that this baseline is oblivious to the query nodes.

Distance returns the k nodes with highest distance centrality with respect to Q. The distance centrality of a node u is defined as $\mathrm{dc}(u) = \left(\sum_{v \in Q} d(u, v)\right)^{-1}$.

3.7 Experimental evaluation

3.7.1 Datasets

We evaluate the algorithms described in Section 3.6 on two sets of real graphs: one set of small graphs that allows us to compare the performance of the fast heuristics against

Table 3.1: Dataset statistics

Dataset	$\lvert V\rvert$	$\lvert E\rvert$
`karate`	34	78
`dolphins`	62	159
`lesmis`	77	254
`adjnoun`	112	425
`football`	115	613
`kddCoauthors`	2 891	2 891
`livejournal`	3 645	4 141
`ca-GrQc`	5 242	14 496
`ca-HepTh`	9 877	25 998
`roadnet`	10 199	13 932
`oregon-1`	11 174	23 409

the greedy approach; and one set of larger graphs, to compare the performance of the heuristics against each other on datasets of larger scale. Note that the bottleneck of the computation lies in the evaluation of centrality. Even though the technique we describe in Section 3.4.1 allows it to scale to datasets of tens of thousands of nodes on a single processor, it is still prohibitively expensive for massive graphs. Still, our experimentation allows us to discover the traits of the different algorithms and understand what performance to anticipate when they are employed on graphs of massive size.

The datasets are listed in Table 3.1. Small graphs are obtained from Mark Newman's repository[1], larger graphs from SNAP.[2] For `kddCoauthors`, `livejournal`, and `roadnet` we use samples of the original datasets. In the interest of repeatability, our code and datasets are made publicly available.[3]

[1] http://www-personal.umich.edu/%7Emejn/netdata/
[2] `http://snap.stanford.edu/data/index.html`
[3] https://github.com/harrymvr/absorbing-centrality

3.7.2 Evaluation Methodology

Each experiment in our evaluation framework is defined by a graph G, a set of query nodes Q, a set of candidate nodes D, and an algorithm to solve the problem. We evaluate all algorithms presented in Section 3.6. For the set of candidate nodes D, we consider two cases: it is equal to either the set of query nodes, i.e., $D = Q$, or the set of all nodes, i.e., $D = V$.

Query nodes Q are selected randomly, using the following process: First, we select a set S of s seed nodes, uniformly at random among all nodes. Then, we select a ball $B(v, r)$ of predetermined radius $r = 2$, around each seed $v \in S$.[4] Finally, from all balls, we select a set of query nodes Q of predetermined size q, with $q = 10$ and $q = 20$, respectively, for the small and larger datasets. Selection is done uniformly at random.

Finally, the restart probability α is set to $\alpha = 0.15$ and the starting probabilities $\mathbf{s}$ are uniform over Q.

3.7.3 Implementation

All algorithms are implemented in Python using the NetworkX package (Hagberg et al., 2008), and were run on an Intel Xeon 2.83GHz with 32GB RAM.

3.7.4 Results

Figure 3·1 shows the centrality scores achieved by different algorithms on the small graphs for varying k (note: lower is better). We present two settings: on the left, the candidates are all nodes ($D = V$), and on the right, the candidates are only the query nodes ($D = Q$). We observe that PPR tracks well the quality of solutions returned by Greedy, while Degree and Distance often come close to that. Spectral algorithms do not perform that well.

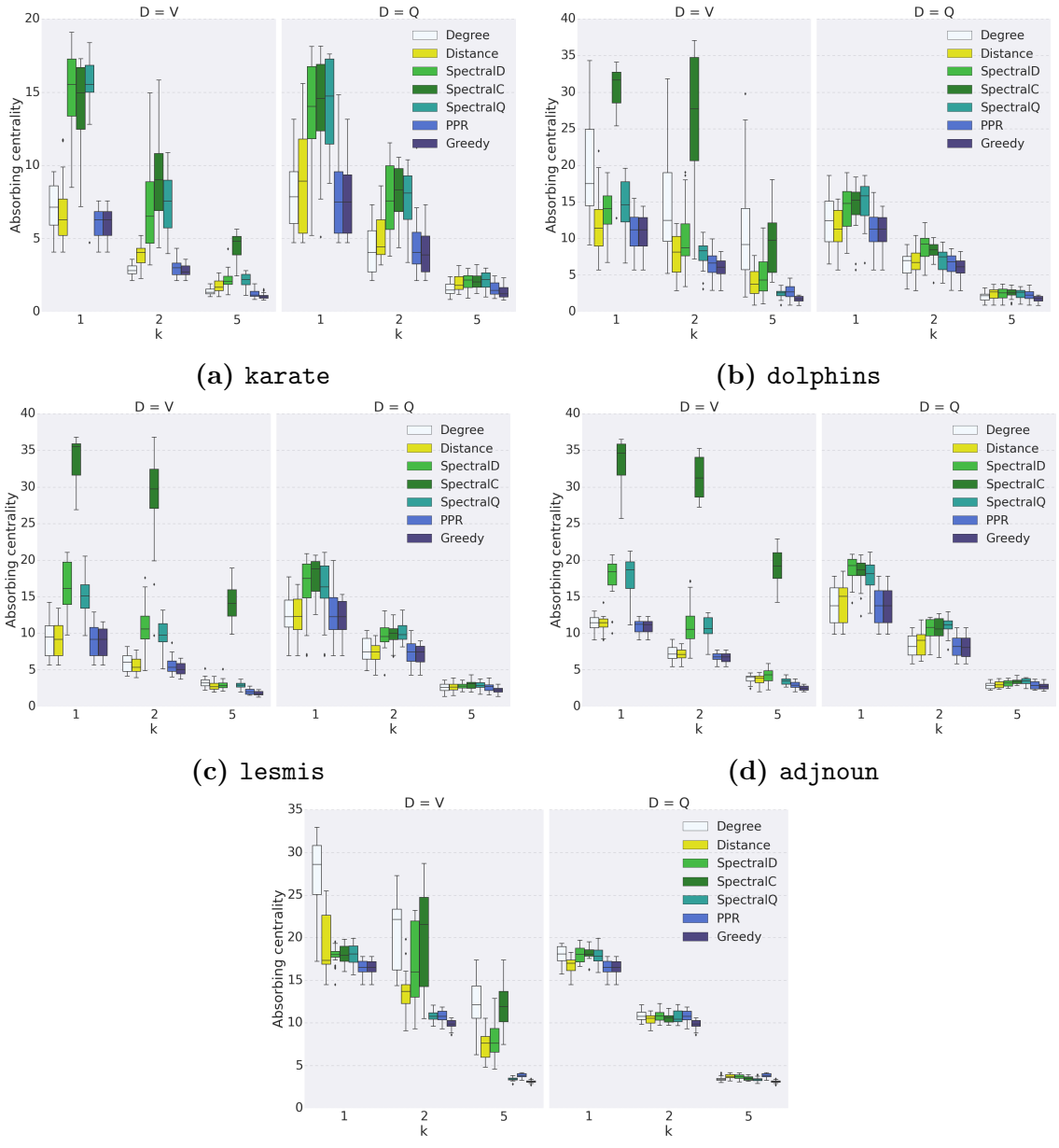
[4] For the planar `roadnet` dataset we use $r = 3$.

Figure 3·2 is similar to Figure 3·1, but results on the larger datasets are shown, not including Greedy. When all nodes are candidates, PPR typically has the best performance, followed by Distance, while Degree is unreliable. The spectral algorithms typically perform worse than PPR.

When only query nodes are candidates, all algorithms demonstrate similar performance, which is most typically worse than the performance of PPR (the best performing algorithm) in the previous setting. Both observations can be explained by the fact that the selection is very restricted by the requirement $D = Q$, and there is not much flexibility for the best performing algorithms to produce a better solution.

In terms of running time on the larger graphs, Distance returns within a few minutes (with observed times between 15 seconds to 5 minutes) while Degree returns within seconds (all observed times were less than 1 minute). Finally, even though Greedy returns within 1-2 seconds for the small datasets, it does not scale well for the larger datasets (running time is orders of magnitude worse than the heuristics and not included in the experiments).

Based on the above, we conclude that PPR offers the best trade-off of quality versus running time for datasets of at least moderate size (more than 10 k nodes).

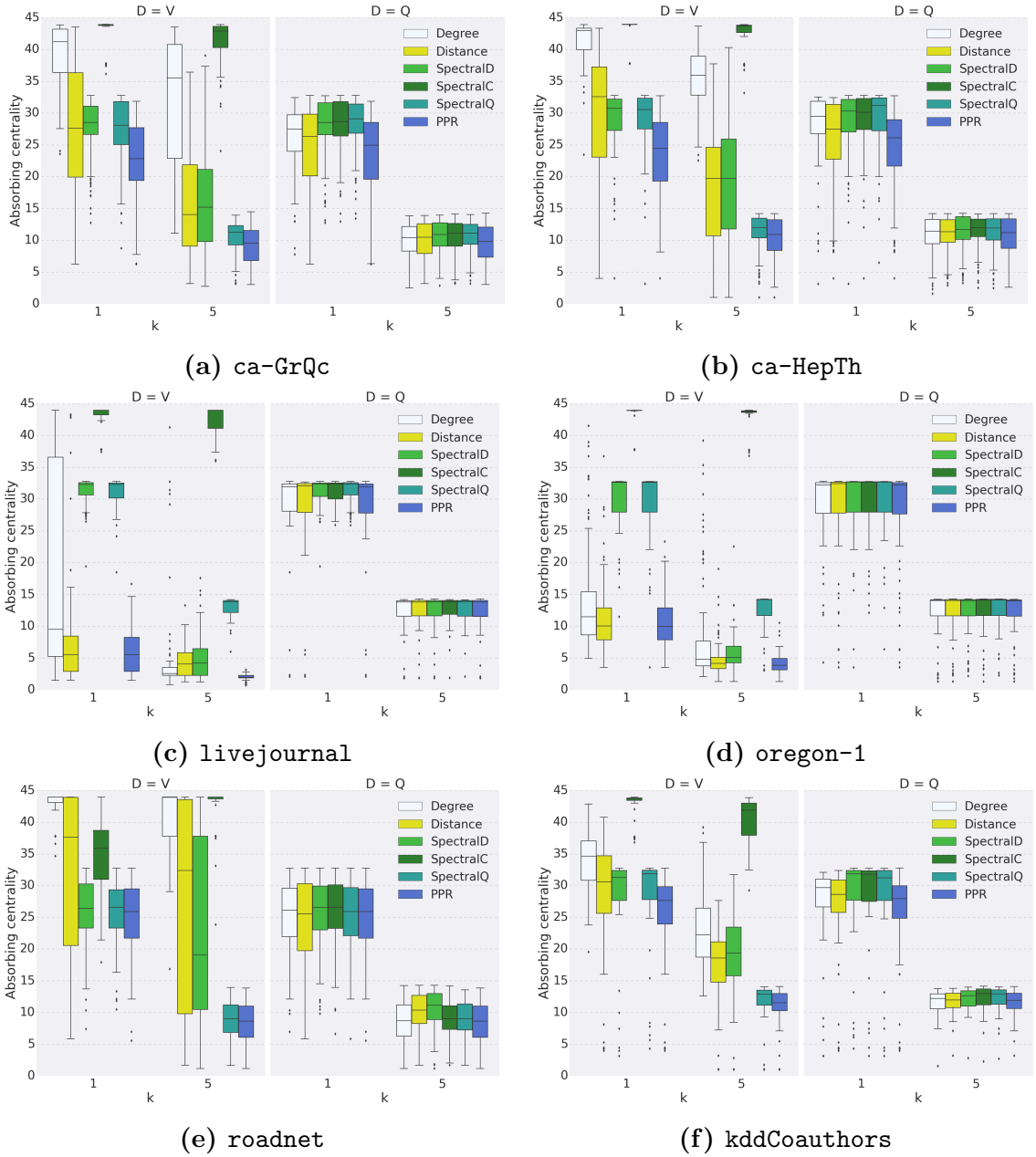


(a) `karate` **(b)** `dolphins`

(c) `lesmis` **(d)** `adjnoun`

(e) `football`

Figure 3·1: Results on small datasets for varying k and $s = 2$.



(a) `ca-GrQc` **(b)** `ca-HepTh`

(c) `livejournal` **(d)** `oregon-1`

(e) `roadnet` **(f)** `kddCoauthors`

Figure 3·2: Results on large datasets for varying k and $s = 5$.

Chapter 4

Coverage-based prize-collecting Steiner tree

4.1 Introduction

In this chapter, we focus on the family of *selection-based* importance measures, and, more specifically, the ones that are *parameter-free*. As a reminder, the selection-based measures aim at finding a group of k entities, with k being part of the input, that collectively are important in the graph. The measures under the *parameter-free* subcategory additionally define a *prize function* on the nodes of the graph. As a result, instead of requiring the size of the group to be fixed, they let the *prize* function guide this decision.

One of the most popular examples of a *parameter-free selection-based* measure is the *prize-collecting Steiner tree (PCST)* (Bienstock et al., 1993). In this setting, each node in the graph is associated with a *prize*. Given a group (subgraph), this importance measure focuses on two factors: (i) the cost for keeping the group connected, i.e., the weight of the edges in the subgraph, and (ii) the foregone profit, i.e., the prize of the nodes that are not part of the selected group. An "important" group is one that *minimizes* both of these values simultaneously[1].

The original application of PCST was in cable networks; a phone company for

[1]In the original definition of PCST, there is a special set of nodes, called *terminal*, which have to be part of the solution, i.e., the solution is required to connect the terminal nodes with each other. However, in this chapter, in order for PCST to fit our use case, we drop this requirement.

example wants to dig the roads and install their own cable network, in order to connect their customers. However, the further they need to dig, the more expensive the project becomes. At the same time, the company wants to minimize the foregone profits, i.e., avoid building a very small (and cheap) network, which however cannot reach potentially well-paying customers. This is exactly the motivation that describes the impact of PCST in the real world. More recently, PCST has been applied to various other domains as well. The work by Rozenshtein *et al.* (Rozenshtein et al., 2014) focuses on activity networks — graphs, in which nodes represent an event in a city (festival, accident, etc.), together with its coordinates, while edge weights reflect the distance between a pair of such event coordinates. Their target is to report compact regions in cities that minimize the ignored activity. Nikolakaki *et al.* (Nikolakaki et al., 2018) used PCST in the setting of road networks. There, the prizes on the nodes correspond to that place's popularity, e.g., the number of photos on Flickr taken in that particular location. The aim of the work is to report tours and paths in the city that are short and cover most of the popular locations.

In this chapter, we consider a different notion for *prize*. We assume that each node $v \in V$ is assigned a set of discrete attributes $A_v = \{a_1, \ldots, a_\ell\}$. The prize of a node is now defined as the size of its attribute set. Equivalently, the prize of a set of nodes $S \subseteq V$ is equal to the number of attributes covered by all the nodes in S, i.e., $\left|\bigcup_{v \in S} A_v\right|$. The intuition behind this choice of prize function is that an "important" group is one, which is cheap to connect and which minimizes the coverage of the nodes not included in it. This change of the prize function initially seems to make the problem of finding the best group completely impractical, requiring the use of techniques for submodular function minimization. We provide algorithms that avoid these computationally expensive tools, while guaranteeing solutions that are at most a factor of 2 worse than the optimal.

4.2 Preliminaries

In this chapter, unless otherwise noted, we will use the following notation. A graph $G = (V, E)$ is a connected, weighted and undirected graph with $|V| = n$ nodes and $|E| = m$ edges. The weight function, also called *edge cost*, is described by a function $c : E \to \mathbb{R}_0^+$ that returns non-negative values. For a subset of edges $M \subseteq E$, we write $c(M) = \sum_{e \in M} c(e)$. Let us also introduce a *prize function* defined on subsets of nodes of G. We will use the notation $\pi(S)$ to denote the prize of the set $S \subseteq V$. More details on how this function is defined will be provided in the following sections. The notation $\overline{V}$ will be used to denote the complement of a set V. Finally, whenever the meaning is clear from the context, we use the subgraph $H = (V_H, E_H)$ of G in place of the corresponding node or edge sets.

4.3 Prize-collecting Steiner Tree

We first start with an overview of the classic prize-collecting Steiner tree (PCST) problem. We formally define the problem and describe a practical algorithm from Hegde *et al.* (Hegde et al., 2015; Hegde et al., 2014) that comes with strong theoretical guarantees.

4.3.1 Problem definition

In this setting, we assume that each node $v \in V$ is assigned a non-negative prize $\pi(v) \geq 0$, and that the prize function of a set $S \subseteq V$ is defined as $\pi(S) = \sum_{v \in V} \pi(v)$. In other words, the prize of a set is equal to the sum of the prizes of the nodes in the set. The goal of the Prize-collecting Steiner tree problem is to find a connected subgraph T that minimizes $c(T) + \pi(\overline{T})$.

Problem 2. *Prize-collecting Steiner tree (PCST): Given a graph $G = (V, E)$, a cost function c, a prize function π and a root node r, identify a connected subgraph*

$T = (V_T, E_T)$, *such that* $r \in V_T$ *and*

$$c(E_T) + \pi(V \setminus V_T) \tag{4.1}$$

is minimized.

Here, we defined the rooted version of the PCST problem. Equivalently, one can define the *unrooted* version, where the solution is not forced to include any particular node. In the rest of the chapter, unless explicitly stated, we will refer to the rooted variant of the PCST problem.

4.3.2 Algorithms for the prize-collecting Steiner tree problem

Being a generalization of the Steiner tree problem, the PCST problem is NP-hard (Karp, 1972). Hence, research on the problem has been focused in finding polynomial-time algorithms that provide good *approximation guarantees*. The work by Goemans and Williamson (Goemans and Williamson, 1995) describes a $2-$approximation algorithm that runs in $O(n^2 \log n)$ time. More recently, Hegde *et al.* (Hegde et al., 2014) proposed a more efficient algorithm based on the one by Goemans and Williamson. This new work delivers a solution with the same approximation guarantees but runs in $O(dm \log n)$, where d denotes the number of bits needed to represent the cost function. The rest of this section provides an overview of this faster algorithm.

Formulating PCST as a primal-dual problem

Both the aforementioned algorithms are based on the *primal-dual* scheme. According to this, the analyst formulates the problem (primal), transforms it into its dual and uses this second formulation to guide the algorithm design. Let us formally define the linear relaxation of the PCST problem.

(Primal)

$$\min \quad \sum_{e \in E} c_e x_e + \sum_{S \subseteq V} \pi(S) z_S \tag{4.2}$$

$$\sum_{e \in \delta(L)} x_e + \sum_{S \supseteq L} z_S \geq 1 \qquad \forall L \subseteq V \setminus \{r\}$$

$$x_e \geq 0 \qquad \forall e \in E$$

$$z_S \geq 0 \qquad \forall S \subseteq V \setminus \{r\}$$

This problem formulation is also called *primal*. Here, $\delta(S)$ is the set of edges with exactly one endpoint in S, also known as the cut of S.

For each edge $e \in E$, we introduce a variable x_e that indicates if e belongs to the solution or not. Similarly, for each set of nodes $S \subset V$, let us use z_S to indicate whether S the a set of nodes for which we will pay the prize $\pi(S)$. The main constraint of the linear program reflects that, for each set of nodes L that does not contain the root, either we pay the cost of some edge e in the set $\delta(L)$, or the set L is contained in the set S that is not in the solution. The above explanation is more straightforward to understand in the case of the integer program version of 4.2. In that case, the non-negativity constraints are transformed to integrality, i.e., variables x_e and z_S are either 0 or 1. However, we choose to work with the linear relaxation of this problem, because it is necessary for the process of designing the primal-dual algorithm.

Let us now introduce the dual formulation of 4.2. Instead of x_e and z_S, we will use the dual variables y_L, for each set of nodes $L \subseteq V \setminus \{r\}$.

(Dual)

$$\max \quad \sum_{L \subseteq V \setminus \{r\}} y_L \tag{4.3}$$

$$\sum_{L:\, e \in \delta(L)} y_L \leq c_e \qquad \forall e \in E$$

$$\sum_{L \subseteq S} y_L \leq \pi(S) \qquad \forall S \subseteq V \setminus \{r\}$$

$$y_L \geq 0 \qquad \forall L \subseteq V \setminus \{r\}$$

The first constraint now can be interpreted as follows: for each edge e, the total dual value (sum of dual variables) of all the sets that contain one of the endpoints of e cannot be larger than the cost of that edge. Similarly, the second constraint establishes that for a given set S, the total dual value of all subsets of S is at most equal to the prize of S.

The primal-dual algorithm

The above formulation leads to the 2−approximation algorithm that was introduced by Goemans and Williamson (Goemans and Williamson, 1995); we will call this algorithm GWAlgorithm. Following, we give a high-level overview of GWAlgorithm and discuss about the improvements introduced by Hegde *et al.*. We will refer to this faster variant of the GWAlgorithm as PCSTFast.

Let us first introduce the notion of the *laminar family*, which is going to be crucial for the rest of the chapter. A collection of *sets* is called *laminar* if, for any two intersecting sets, one is contained in the other.

Definition 3. *Laminar family: A collection of sets $\mathcal{L} = \{L_1, L_2, \ldots, L_k\}$ is called laminar if, for any two sets $L_i, L_j \in \mathcal{L}$ either (i) $L_i \cap L_j = \emptyset$, or (ii) $L_i \subseteq L_j$, or (iii) $L_j \subseteq L_i$.*

We will be using the laminar family to capture a partition of the nodes of G into nested sets, i.e., $L_i \subseteq V$ for all $L_i \in \mathcal{L}$. In other words, the laminar family will be used to track a hierarchical clustering of the nodes, where a cluster is produced only by merging two existing clusters together. We will use the terms *cluster* and *component* to refer to members of the laminar family.

GWAlgorithm works in two stages. First, it operates on a laminar family of clusters by merging and deactivating them, until there is only a single active cluster left. Then, it executes a pruning process to remove unnecessary nodes from the final cluster and reports a solution. We will focus solely on the first part, as the second part is a heuristic that is used specifically to improve the cost of the solution. For further details on pruning strategies, we refer the reader to (Goemans and Williamson, 1995) and (Johnson et al., 2000).

Let us now describe the first stage of GWAlgorithm, also called the *growth phase*. We start by initializing $\mathcal{L}$ to be a collection of singleton sets, one for each node $v \in V$. We label all these sets as *active*, except for the one corresponding to the root node r, which is marked as deactivated. Also, for each set $L \in \mathcal{L}$, we assign to it a value $y_L = 0$, which we call *dual value*. During the execution of the growth phase, the dual values of the active clusters will be increase. However, two properties will always hold, which correspond to the main constraints of Equation 4.3:

- For each edge $e \in E$, let $\mathcal{L}_e \subseteq \mathcal{L}$ be the clusters that contain exactly one endpoint of e. More specifically, if $e = (u, v)$, then for each $C \in \mathcal{L}_e$, either $u \in C$ or $v \in C$, but not both. Then the total dual value of $\mathcal{L}_e$ cannot grow beyond $c(e)$, i.e., $\sum_{L \in \mathcal{L}_e} y_L \leq c(e)$. This is exactly the first constraint of the dual problem. When this holds as an equality, we will say that the *edge constraint* for edge e became *tight*.

- For all clusters $L \in \mathcal{L}$, let $\mathcal{L}_L \subseteq \mathcal{L}$ be the collection of clusters $C \in \mathcal{L}$ for

which $C \subseteq L$. Intuitively, if the laminar family can be viewed as a hierarchy of clusters, $\mathcal{L}_L$ corresponds to the set of the nodes in the tree rooted at L. This contains the nodes that were merged to create L, together with their respective predecessors. Then, the total dual value of the clusters in $\mathcal{L}_L$ cannot grow beyond the prize of L, i.e., $\sum_{C \in \mathcal{L}_L} y_C \leq \pi(L)$. Again, this corresponds to the second constraint of the dual, and, whenever this holds as an equality, we will say that the *cluster constraint* for cluster L became *tight*.

Overall, GWAlgorithm starts by initializing the laminar family as we described above, and begins increasing the dual values of the active clusters until either an edge constraint or a cluster constraint becomes tight. In the first case, let us assume that the constraint of edge $e = (u, v) \in E$ became tight. We denote as $C_u, C_v \in \mathcal{L}$ the maximal clusters in the laminar family that contain nodes u and v respectively. The algorithm labels both C_u and C_v as *deactivated*, creates a new active[2] cluster $C' = C_u \cup C_v$, updates $\mathcal{L}$ to be $\mathcal{L} \cup C'$ and sets the dual value of this new cluster $y_{C'}$ equal to 0. Otherwise, if the constraint for cluster C became tight, the algorithm labels C as *deactivated*. After both of these cases, GWAlgorithm continues its execution by increasing the dual values of the active clusters further. We summarize GWAlgorithm in Algorithm 5.

The bottleneck of this algorithm is at Lines 6–7, where it finds when the next event (cluster constraint or edge constraint becoming tight) will happen. Specifically, one can find which is the next cluster to become tight quite efficiently, by maintaining a priority queue with the cluster prizes. However, finding which edge becomes tight (Line 7) is much more challenging. The straightforward approach is to check all edges at each iteration of GWAlgorithm, which would incur an $O(mn)$ increase in the computational complexity of the algorithm. Hegde *et al.* proposed an alternative

[2]If either C_u or C_v contains the root node r, C' will be marked as *inactive*.

Algorithm 5 High-level description of the primal-dual algorithm GWAlgorithm

$\mathcal{L} \leftarrow \{\{v\} : v \in V\}$ ▷ Laminar family of clusters
$y_L \leftarrow 0$ for all $L \in \mathcal{L}$ ▷ Initial dual solution
$E_F = \emptyset$ ▷ Initial forest
$\mathcal{A} \leftarrow \mathcal{L}$ ▷ Family of active components
while $|\mathcal{A}| > 1$ **do** ▷ Growth stage
    $\epsilon_d \leftarrow$ next cluster deactivation ▷ Cluster constraint becomes tight
    $\epsilon_m \leftarrow$ next edge event time
    $\epsilon = \min\{\epsilon_d, \epsilon_m\}$
    for $L \in \mathcal{A}$ **do**
        $y_L \leftarrow y_L + \epsilon$ ▷ Fast-forward time
    if $\epsilon_d \leq \epsilon_m$ **then** ▷ Cluster event
        for $L \in \mathcal{A}$, such that $\sum_{C \in \mathcal{L} :\, C \subseteq L} y_C = \pi(L)$ **do**
            $\mathcal{A} \leftarrow \mathcal{A} \setminus \{L\}$ ▷ Mark L as deactivated
    else ▷ Edge event
        Let e be such that $\sum_{L \in \mathcal{L} :\, e \in \delta(L)} y_L = c(e)$ and $e \in \delta(L)$ for some $L \in \mathcal{A}$
        Let C_u and C_v be the clusters of $\mathcal{L}$ containing the endpoints of e
        $\mathcal{L} \leftarrow \mathcal{L} \cup \{C_u \cup C_v\}$ ▷ Merge the two components
        $E_F \leftarrow E_F \cup \{e\}$ ▷ Add e to the solution
        $y_{C_u \cup C_v} \leftarrow 0$
        $\mathcal{A} \leftarrow \mathcal{A} \cup \{C_u \cup C_v\}$
        $\mathcal{A} \leftarrow \mathcal{A} \setminus \{C_u, C_v\}$ ▷ Deactivate the merged components
Run the pruning function on E_F with respect to the last active component

method for keeping track of these edge events, which is based on *dynamic edge splitting* and runs in nearly-linear time (Hegde et al., 2014). We give a brief overview of this approach in the next paragraph.

Instead of considering the edges as simple entities, PCSTFast splits each edge $e = (u, v)$ into two parts, e_u and e_v. For each edge part, we maintain an *event value* μ, which tracks how much the dual value of its active endpoint can grow before it triggers an *edge event*. A property that holds throughout the algorithm's execution is that $\mu(e_u) + \mu(e_v) = c(e)$. Initially, we set $\mu(e_u) = \mu(e_v) = \frac{c(e)}{2}$. An edge event will occur, when the total dual value of the clusters that contain the endpoint node is equal to the corresponding μ value, i.e.,

$$\mu(e_u) - \sum_{L \in \mathcal{L}:\, u \in L} y_L = 0 \tag{4.4}$$

This does not however mean that the corresponding edge constraint has become tight. It might instead mean that only one of the edge's endpoints has been actively growing and it has grown as much as it could. In that case, we compute the remaining slack of the edge, i.e., $c(e) - \sum_{L \in \mathcal{L}_e} y_L$, and update the values $\mu(e_u)$ and $\mu(e_v)$ respectively. The authors show that, together with a smart choice of data structures, this method guarantees an $O(dm \log n)$ runtime. Here, d represents the number of bits that are required to represent the values (costs and prizes) in the input, which also reflects the maximum number of times an edge event can be triggered for a particular edge.

4.4 Coverage-based PCST

After revisiting the definition of the PCST problem and presenting two seminal algorithms for finding approximate solutions, we proceed by defining our variation of the PCST, when the prize is the coverage function. As mentioned previously, we assume that there exists a universe of attributes A_G, and that each node $v \in V$ is described

by a subset of these attributes $A_u \subseteq A_G$. We will denote the set of all A_u's as A, i.e., $A = \{A_u : \forall u \in V\}$.

Problem 3. *Coverage-based prize-collecting Steiner tree (PCST-Cover): Given a graph $G = (V, E, A)$, a cost function c, and a root node r, identify a connected subgraph $T = (V_T, E_T)$, such that $r \in V_T$ and*

$$c(E_T) + \pi(V \setminus V_T) \tag{4.5}$$

is minimized. Here, the prize function of a set $S \subseteq V$ is defined as the attribute coverage of S, i.e., $\pi(S) = |\cup_{v \in S} A_v|$

4.4.1 Analysis of the primal-dual approach

The algorithms we will present for PCST-Cover will follow the same primal-dual scheme as the GWAlgorithm. In the later sections, we will elaborate more about the computational challenges that the coverage function introduces and how we deal with them. However, from a broad perspective, all algorithms will perform the same steps as GWAlgorithm. Here, we will show that the 2-approximation guarantee of these algorithms is still valid.

Let us first analyze the prize function, since the analysis for the cost is the same as in the original PCST setting. We start with some useful observations.

Lemma 3. *Let $\overline{y}(S) = \sum_{L \subseteq S} y_L$ denote the total dual value of a set S. The function $\overline{y}$ is* supermodular*, i.e., $\overline{y}(S) + \overline{y}(T) \leq \overline{y}(S \cap T) + \overline{y}(S \cup T)$.*

Proof. For every set L not containing r, we have the following disjointed cases: (i) $L \not\subseteq S \cup T$, (ii) $L \subseteq S \cap T$, (iii) $L \subseteq S$ and $L \not\subseteq T$, (iv) $L \subseteq T$ and $L \not\subseteq S$, and (v) $L \subseteq S \cup T$ and $L \not\subseteq S$ and $L \not\subseteq T$. In each of these cases, we can see that the contribution of L to the right-hand side of the claimed inequality is at least as much as its contribution to the left-hand side. Actually, the contribution is the same in all cases except for the last. In that case, L only contributes to the union on the right-hand side. □

We will say that a set S is y-tight, if $\overline{y}(S) = \pi(S)$, i.e., when the dual value of the

set is equal to the number of attributes covered by S. Since $\overline{y}$ is supermodular and π is submodular (the coverage function is submodular), we can show that if two sets S and T are y-tight, then $S \cap T$ and $S \cup T$ are also y-tight.

Lemma 4. *For any feasible dual solution y, if S and T are $y-$tight, then $S \cap T$ and $S \cup T$ are also $y-$tight.*

Proof. We have

$$\begin{aligned} \pi(S) + \pi(T) &= \overline{y}(S) + \overline{y}(T) && (S \text{ and } T \text{ are } y\text{-tight}) && (4.6)\\ &\leq \overline{y}(S \cap T) + \overline{y}(S \cup T) && (\overline{y} \text{ is supermodular}) && (4.7)\\ &\leq \pi(S \cap T) + \pi(S \cup T) && (y \text{ is feasible}) && (4.8) \end{aligned}$$

Due to the submodularity of π, we also have that $\pi(S) + \pi(T) \geq \pi(S \cap T) + \pi(S \cup T)$. As a result, the inequalities above hold with equality. □

The above lemma means that at any point of the execution of the algorithm, there exists a unique inclusion-maximal cluster that becomes tight and which contains all the clusters that became tight. Moreover, each cluster that becomes tight will remain tight throughout the execution, and, in particular, it will be tight with respect to the final dual solution.

Now, let us consider the vertices that are not in the final solution T, which are the ones whose prize we pay for. All of these vertices are in the union of the maximal clusters whose constraints became tight. This holds because every inactive cluster is either containing the root, in which case it will be part of the solution, or otherwise it was marked as deactivated because it was contained in a cluster whose constraint became tight. Let X be the union of the clusters whose constraint became tight. Each of these clusters is tight with respect to the final dual solution y, which means that X is y-tight as well. Thus, the prize of X is at most equal to the total dual value

of X.

$$\pi(V \setminus V_T) \leq \pi(X) \qquad (V \setminus v_T \subseteq X \text{ and } \pi \text{ is monotone}) \tag{4.9}$$

$$= \sum_{L \subseteq X} y_L \qquad (X \text{ is } y\text{-tight}) \tag{4.10}$$

At the same time, the edge cost of the final solution is the same as in the original PCST formulation:

$$\sum_{e \in E_T} c(e) \leq \sum_{L \subsetneq X} y_L \tag{4.11}$$

Theorem 2. *The primal-dual algorithm for the PCST-Cover problem retrieves solutions that are a 2-approximation to the optimal.*

Proof. Combining Equations 4.10 and 4.11, we have that

$$\sum_{e \in E_T} c(e) + 2\pi(V \setminus V_T) \leq 2 \sum_{L \subseteq V} y_L \tag{4.12}$$

where T is a solution returned by the primal-dual algorithm. Following the same steps as in Feofiloff *et al.* (Feofiloff et al., 2010), we can show that

$$\sum_{e \in E_T} c(e) + 2\pi(V \setminus V_T) \leq 2 \left(\sum_{e \in E_{\text{OPT}}} c(e) + \pi(V \setminus V_{\text{OPT}}) \right) \tag{4.13}$$

where OPT corresponds to the optimal solution of PCST-Cover on the input. □

4.5 Approximation algorithms for the coverage-based PCST

Although we mentioned that there are strong similarities between the algorithms that we will introduce below and the GWALGORITHM, there are still some important differences. These are caused by the change in the prize function, which switches from a linear function (sum) to a submodular (coverage). These differences are concentrated on how the algorithm interacts with the components in the laminar family; (i) how it picks the *next cluster to deactivate*, and (ii) how it executes the *deactivation*

process. Following, we will look into these differences, we will go over the new computational challenges that are introduced and, finally, we will propose elegant and efficient algorithmic solutions.

4.5.1 Finding the next component to deactivate

As we previously established, a component L deactivates when its corresponding constraint becomes *tight*. This happens, when its total dual value $\overline{y}(L)$ becomes equal to its prize $\pi(L)$. We also mentioned that as "time" passes, the algorithm slowly increases the dual values of all the active components. Consequently, an increase of the dual values by ϵ will cause $\overline{y}(L)$ to increase by $\epsilon \cdot \text{active}(L)$, where $\text{active}(S) = |\{C \in \mathcal{A} \colon C \subseteq S\}|$. This means that, for each set $S \subseteq V$, we can increase its dual values by *at most* $\frac{\pi(S)-\overline{y}(S)}{\text{active}(S)}$.

The aim of Line 6 in GWAlgorithm is to find the maximum value ϵ that we can increase the duals before a component becomes deactivated, and also identify this maximal component. In the case of the original PCST this is straight-forward, as we only need to iterate over the clusters $C \in \mathcal{A}$, and return the one with the minimum slack $\pi(C) - \overline{y}(C)$. A submodular prize function however complicates the situation. Consider for example two nodes, u and v, with attributes $A_u = \{a, b, c, d\}$ and $A_v = \{b, c, d, e\}$ respectively. Let us assume that the corresponding singleton components in the laminar family, i.e., $\{u\}$ and $\{v\}$, are active, and that all the dual values are equal to zero. The slack of these two components *individually* is equal to $\frac{\pi(\{u\})}{\text{active}(\{u\})} = \frac{\pi(\{v\})}{\text{active}(\{v\})} = 4$. However, looking at the set $\{u, v\}$, we see that its slack is much smaller, i.e., $\frac{\pi(\{u,v\})}{\text{active}(\{u,v\})} = 2.5$. This example aims to show that the change of prize function introduces significant computational challenges to the problem.

Thus, finding the next component that deactivates turns into a submodular function minimization problem. Specifically, the method for identifying which cluster constraint becomes tight first can be summarized as:

1. Make a guess $\lambda \in \mathbb{R}^+$

2. Find the minimizer $S^\star$ of the function

$$f(S) = \pi(S) - \overline{y}(S) - \lambda \cdot \text{active}(S) \tag{4.14}$$

   and check that $f(S^\star) \geq 0$.

3. Repeat steps 1 and 2 to find the maximum feasible λ and the corresponding $S^\star$.

The function f is a submodular function, since π is submodular, $\overline{y}$ is supermodular, and active is modular. As a result, given a value λ, minimizing f becomes an instance of a *submodular function minimization problem*. Using such a generic process to find the largest feasible λ would be computationally very expensive. Instead, we will show how we can reduce this optimization problem to the minimum cut problem on a specially constructed graph.

4.5.2 Reduction to minimum-cut

Here, we will introduce the *coverage graph*, an object specifically constructed to quickly find the minimizer of the function introduced in Equation 4.14. Then, we will prove that minimizing f is equivalent to finding the minimum cut on the coverage graph.

For each set L in the laminar family $\mathcal{L}$, let $w(L) = y_L + \lambda$ if L is active, and $w(L) = y_L$ if it is inactive. Notice that the way we defined w, it is a non-negative function. Our goal from Equation 4.14 is to minimize $\pi(S) - \sum_{L \subseteq S} w(L)$. This is equivalent to minimizing $\pi(S) + \sum_{L \not\subseteq S} w(L) - \sum_{L \subseteq V} w(L)$. However, the last component of this formula is a constant, so instead, one may choose to minimize

$$\pi(S) + \sum_{L \not\subseteq S} w(L) \tag{4.15}$$

Now, let us construct the coverage graph. This will be a tri-partite network, consisting of the following layers:

- The first layer contains a vertex a_i, for each element i in the ground set of attributes, A_G.

- The second layer contains a vertex b_j, for each set $A_j \in A$. Initially, there are $|V|$ such vertices, each corresponding to the attribute set of each original node $v \in V$.

- The third layer has a vertex c_L, for each set L in the laminar family.

We finally add a *source node* s and a *sink node* t. In this coverage graph, we add the following edges:

- Between the first and the second layer, we add an *infinite capacity* edge from a_i to any node b_j, such that $i \in A_j$.

- Between the second and the third layer, we add an *infinite capacity* edge from b_j to c_L, if $j \in L$.

- We connect the source node with every node in the first layer, using an edge of capacity 1.

- We connect each node c_L in the third layer with the sink node. That edge will have capacity $w(L)$, i.e., $y_L + \lambda$ if L is active, or y_L otherwise.

We will denote this coverage graph as G^{Cov}.

Now, we will show that instead of minimizing the quantity in Equation 4.15, which is equivalent to minimizing our original function f that we defined in Equation 4.14, we can compute a minimum cut on G^{Cov}. The nodes of the third layer that belong in the sink-side cut will correspond to the minimizer $S^\star$, while the value of the cut

will be equal to $f(S^\star)$. The proof includes two steps, that are summarized in the following Lemmas.

Lemma 5. *Any solution S of Equation 4.15 can be mapped to a cut on G^{Cov} with cost $\pi(S) + \sum_{L \not\subseteq S} w(L)$.*

Proof. Consider a set $S \subseteq \{1, 2, \ldots, n\}$. We obtain a valid cut in the graph G^{Cov} as follows:

1. We cut all the edges (s, a_j), with $j \in A_i$ for all $i \in S$. The total capacity of these edges is $\pi(S)$.
2. We cut all the edges (c_L, t), with $L \not\subseteq S$. The total capacity of these edges is $\sum_{L \not\subseteq S} w(L)$.

We need to verify that, after removing these edges, s is disconnected from t. Any path from s to t is of the form $s \to a_i \to b_j \to c_L \to t$, where $j \in L$ and $i \in A_j$. If $j \in S$, then we removed the edge (s, a_i), and the path did not survive. Otherwise, we removed the edge (c_L, t), and again the path was invalidated. Therefore, the removed edges form a cut, with total capacity equal to $\pi(S) + \sum_{L \not\subseteq S} w(L)$. □

Lemma 6. *The minimum cut in G^{Cov} can be mapped to a solution $S \subseteq \{1, 2, \ldots, n\}$, whose value $\pi(S) + \sum_{L \not\subseteq S} w(L)$ is equal to the minimum cut capacity.*

Proof. Let Z be the set of edges in a minimum cut of G^{Cov}. Let S be the union of the components that are *not* reachable from the source, i.e.

$$S = \bigcup_{\substack{L \in \mathcal{L}: \\ c_L \text{ is not reachable from } s \text{ in } G^{\text{Cov}} \setminus Z}} L \tag{4.16}$$

Now, consider the nodes c_L that correspond to components not in S, i.e., $L \not\subseteq S$. These are by design reachable from s and, consequently, the edge (c_L, t) has to be part of the cut. The total capacity of these edges is $\sum_{L \not\subseteq S} w(L)$. Furthermore, since we are not allowed to remove edges between the second and the third layer, the components that are in S have been disconnected from the source, because edges (s, a_i), such that $i \in A_j$ for all $j \in S$, are in the cut. The total capacity of these edges is $\pi(S)$. As a result, the overall capacity of the minimum cut Z is $\pi(S) + \sum_{L \not\subseteq S} w(L)$. □

Together, Lemmas 5 and 6 provide a proof that the solution to the minimization of Equation 4.14 can be given by a minimum cut computation on the coverage graph.

4.5.3 A 2-approximation algorithm based on the minimum-cut reduction

We can now describe in detail the changes that need to be done in GWAlgorithm (Algorithm 5) in order to design an algorithm for PCST-Cover. Actually, the algorithm we will build upon in practice is PCSTFast and not GWAlgorithm, however, the changes do not affect the novelties that PCSTFast introduces to achieve the improved running time.

Initialization

During the initialization phase of the algorithm, we need to build the coverage graph, as we described in Section 4.5.2. Following, we will show how we can maintain it up-to-date throughout the algorithm's execution.

Finding the next cluster deactivation (Line 6)

Here, we will combine the reduction we showed previously, together with a search on the values of λ. Specifically, we will follow this process:

1. Initialize λ with a small value.
2. Solve the minimum cut on the coverage graph, for the particular value of λ. Find minimizer $S^\star$ and cut value $f(S^\star)$.
3. While $f(S^\star) \geq 0$, run *galloping search* on the values of λ; set $\lambda = 2 \cdot \lambda$ and go to Step 2.
4. After finding an upper bound for λ, run *binary search* on the values of λ in the range $\left[\frac{\lambda}{2}, \lambda\right]$, each time following the same steps as in Step 2. If the resulting $f(S^\star) < 0$, update the upper bound of the range, otherwise, update the lower bound.

It suffices to repeat the binary search until we narrow the search interval down to $\Theta(\frac{1}{n})$ around 0. The total number of binary calls is $O\big(\log(n \max_{S \subseteq V} \pi(S))\big)$.

Computing the minimum cut

This is going to be a central part of the algorithm. The minimum cut computation will need to be repeated multiple times and, as result, we need to speed it up as much as possible. We will use the very practical *preflow-push algorithm* (Goldberg and Tarjan, 1988), together with the global relabeling and gab relabeling heuristics (Cherkassky and Goldberg, 1997). Its complexity is $O(n^2 m)$, however it has proven to be very fast in practice. Additionally, it is worth noting that we do not need the complete flow assignment on the edges of the coverage graph. We simply need the cut and its value, which we can retrieve early in the algorithm's execution. Finally, oftentimes, we will be able to use a solution of the minimum cut for a specific value of λ as a preflow in the next iteration of the search.

Cluster deactivation (Lines 12 – 13)

After the binary search above indicates that the constraint for a set S will become tight next, we need to deactivate the corresponding clusters, when that event is triggered. We mark as inactive all clusters $L \in \mathcal{A}$, such that $L \subseteq S$.

Update the coverage graph

In order to maintain a valid coverage graph, we will need to update it throughout the execution of the algorithm. The only time that we need to change the graph is whenever two clusters need to be merged into a new one. Let us assume that c_L and c_M are the two nodes in the third layer of the coverage graph that correspond to the two cluster being merged. We will create a new node, $c_{L \cup M}$, which will correspond to the new cluster, and add it to the third layer of G^{Cov}. Furthermore, we will iterate

over all the *in*-neighbors of c_L and c_M, and connect them to this new node with edges of infinite capacity. Finally, we will connect $c_{L\cup M}$ with t using an edge of capacity $y_{L\cup M} = 0$.

We refer to the algorithm that builds on top of PCSTFast and implements the changes we just mentioned as PCSTCover.

4.5.4 A parametric-search approach for finding λ

In the previous section, we described how we can find by how much we can grow the dual values before a cluster deactivation happens. For this, we had to solve multiple minimum-cut problems, each time guessing a different value for λ. Now, we will show how we can find the best such λ doing a single minimum-cut computation.

In order to understand how we can transform our problem into a parametric-search, we need to look back at how we constructed the coverage graph. Across the different minimum-cut calls of our search for λ, the only quantity that changes in the graph is the capacity of the edges from nodes in the third layer that correspond to active clusters to the sink node t. This situation sounds very similar to the parametric max-flow problem definition, as discussed by Gallo *et al.* (Gallo et al., 1989). In their problem definition, the capacities of the edges that are outgoing from the source node and the edges incoming to the target node are considered to be functions of a single parameter, which they conveniently call λ. More specifically, Gallo *et al.* require that (i) the outgoing edges from the source have capacity, which is described as a *non-decreasing* function of λ, and (ii) the incoming edges to the sink have capacity, which is defined as a *non-increasing* function of λ. In their work, they show that in this setting, the preflow-push algorithm can be modified to compute all possible cuts, for all the values of λ, with a single call. In fact, because of their capacity requirements, they can show that these cuts are nested and are at most n. Equivalently, there are at most n different values of λ that produce these different cuts. Moreover, they show

that these values of λ, together with their corresponding cuts, can be computed in increasing order, i.e., the algorithm finds $\lambda_1 < \lambda_2 < \ldots < \lambda_n$.

In our case, our problem is to find the next cluster deactivation time, i.e., find the value $\lambda^\star = \min_{S \subseteq V} \frac{\pi(S) - \overline{y}(S)}{\text{active}(S)}$. This can equivalently be written as a parametric-search problem:

$$\lambda^\star = \max\{\lambda \colon \pi(S) - \overline{y}(S) - \lambda \cdot \text{active}(S) \geq 0,\ \forall S \subseteq V\} \tag{4.17}$$

The above holds because, for $\lambda^\star$, there exists a set $S^\star$, which minimizes the function f. In Equation 4.14, we are requiring $f(S^\star) \geq 0$. Consequently, $f(S) \geq f(S^\star) \geq 0$, for any $S \subseteq V$ when using $\lambda^\star$. Following, we will describe an algorithm that solves parametric-search problems like this, and we will show how we can use it.

A discrete Newton's method for finding the next cluster deactivation time

Algorithm 6 describes a method for finding the next cluster deactivation time, denoted by $\lambda^\star$. Technically, $\lambda^\star$ represents how much we can grow the dual values of the active clusters in the laminar family, before one of them becomes tight.

Algorithm 6 Finding the next cluster deactivation time as a parameter-search

```
1: Initialize a value λ_0 > λ*
2: i ← 0
3: while true do
4:     S_i ← the minimizer of π(S) − ȳ(S) − λ_i · active(S)
5:     if π(S_i) − ȳ(S_i) − λ_i · active(S_i) = 0 then
6:         λ* ← λ_i
7:         break
8:     else
9:         λ_{i+1} ← (π(S_i) − ȳ(S_i)) / active(S_i)
10:        i ← i + 1
11: return λ*
```

Following, we will prove that indeed the algorithm computes what it is expected to.

Lemma 7. *Algorithm 6 retrieves a decreasing sequence of values* $\lambda_0 > \lambda_1 > \ldots \geq \lambda^\star$.

Proof. Let $S^\star$ be the minimizing set that corresponds to $\lambda^\star$, i.e., $\lambda^\star = \frac{\pi(S^\star)-\overline{y}(S^\star)}{\text{active}(S^\star)}$. We have

$$\begin{aligned}
&\pi(S_i)-\overline{y}(S_i) - \lambda_i \cdot \text{active}(S_i) \\
&\qquad \leq \pi(S^\star) - \overline{y}(S^\star) - \lambda_i \cdot \text{active}(S^\star) && (S_i \text{ is the minimizer}) && (4.18)\\
&\qquad \leq \pi(S^\star) - \overline{y}(S^\star) - \lambda^\star \cdot \text{active}(S^\star) && (\lambda_i \geq \lambda^\star \text{ and } \text{active}(S^\star) \geq 0) && (4.19)\\
&\qquad = 0 && && (4.20)
\end{aligned}$$

Therefore, $\pi(S_i) - \overline{y}(S_i) - \lambda_i \cdot \text{active}(S_i) \leq 0$. If this value is equal to 0, then $\lambda^\star = \lambda_i$, and we are done. Otherwise, we let $\lambda_{i+1} = \frac{\pi(S_i)-\overline{y}(S_i)}{\text{active}(S_i)}$. Note that $\lambda_{i+1} \geq \lambda^\star$, since

$$\lambda_{i+1} = \frac{\pi(S_i) - \overline{y}(S_i)}{\text{active}(S_i)} \geq \min_{S \subseteq V} \frac{\pi(S) - \overline{y}(S)}{\text{active}(S)} = \lambda^\star \tag{4.21}$$

Additionally, $\lambda_{i+1} < \lambda_i$, since $\pi(S_i) - \overline{y}(S_i) - \lambda_{i+1} \cdot \text{active}(S_i) = 0$ and $\pi(S_i) - \overline{y}(S_i) - \lambda_i \cdot \text{active}(S_i) < 0$. □

Let us now return to the parametric maximum flow algorithm by Gallo *et al.*, and describe how we can use it within Algorithm 6, and more specifically at Line 4. There are two main problems with using the former algorithm as is:

1. The algorithm requires that the capacity of the edges incoming to the sink node is a non-increasing function of the parameter λ. In our case, this function is either non-decreasing ($y_L + \lambda$), or constant.

2. The algorithm produces a sequence of solutions that correspond to increasing values of λ. However, Algorithm 6 retrieves a decreasing sequence of values $\lambda_0 > \lambda_1 > \ldots \geq \lambda^\star$.

In order to deal with these two issues, we switch the capacity of the edges from $y_L + \lambda$ to $y_L - \lambda$ and we consider the sequence $-\lambda_0 < -\lambda_1 < \ldots \leq -\lambda^\star$. This little trick allows to combine the discrete Newton's method of Algorithm 6 with the parametric max-flow algorithm of Gallo *et al.*. The running time complexity of this

combined algorithm for the PCST-Cover problem is $O(dm \log n + dm^2 n^2)$. As before, the solutions retrieved by the algorithm are still a 2-approximation to the optimal.

Choosing a value for λ_0

Algorithm 6 requires an initial guess for λ_0. The closer this guess is to $\lambda^\star$, the fewer number of iterations are needed before the algorithm terminates. A pessimistic guess would be the number of attributes in the ground set, i.e. $\lambda_0 = |A_G|$. We will call this algorithm PCSTCover-parametric*.

Another, more aggressive strategy, is to set λ_0 equal to the smallest component slack, i.e.,

$$\lambda_0 = \min_{L \in \mathcal{A}} \frac{\pi(L) - \overline{y}(L)}{\text{active}(L)} \tag{4.22}$$

The intuition behind this choice is the following: looking at the active clusters, there is one, which we will denote as $L^\star$, whose slack is the smallest. The longest $L^\star$ can remain active is equal to its remaining slack value. There is no way that the λ returned by Algorithm 6 will point to a later deactivation time than this. As a result, we can use this value as a good first guess. This is the most efficient algorithm among the ones we provide, as we will see in the next section. We refer to this algorithm as PCSTCover-parametric.

4.6 Experiments

In this section, we will explore the behavior of our algorithms both in a controlled environment and on a real-world collaboration dataset. First, using synthetic data, we will investigate how changes in the input affect different aspects of the algorithms' output. Later, using the real data, we will provide an example of the type of practical information that the proposed importance measure offers. Furthermore, we will see how the analyst can preprocess the data depending on the specific aim of the analysis,

and how this can affect the end result.

Experimental setup

All of our proposed algorithms were implemented in C++. For PCSTFast, we used the implementation that is available on GitHub[3]. The experiments were ran on a MacBook Pro (early 2015 version), with a 2.7 GHz Intel Core i5 CPU and 8GB of memory.

4.6.1 Experiments on synthetic data

First, we will synthetically generate the graph G, the attribute sets A and the cost function c, in order to explore how changes in the input can affect the performance of our algorithms, as well as the properties of the solution.

More specifically, we will create our input in the following way: the graph G will be an instance of an Erdős-Rényi random graph with $n = 1000$ nodes and $\rho = 0.05$ connection probability. Each node v in the graph will be fixed to contain 12 attributes, i.e., $|A_v| = 12$. In order to control how common these attributes are, we will split the attribute set of each node in two parts: (i) the *common* attributes, that exist in all the nodes of the graph, and (ii) the *unique* attributes, that are only present in the particular node. Putting together the common and the unique attributes for each node results in the attribute set A_v of that particular node. The default sizes for these two parts is 6 common attributes and 6 unique attributes per node. Finally, each edge is assigned a weight uniformly at random $c(e) \in (0, 800)$.

Experiment parameters

For the purpose of exploring the algorhtms' behavior, we will control and change the following variables when generating the input data:

[3] `https://github.com/fraenkel-lab/pcst_fast`

- *number of nodes*: One important property of the data is the size of the graph. As indicated by the complexity of our algorithms, we are expecting that larger graphs will be more computationally demanding.

- *number of edges*: Another important property of the graph is its density. This is expected to have a significant impact on the running time. Additionally, adding more edges in a graph gives the algorithm more flexibility as to how to build the solution. Therefore, the size of the returned subgraph is also expected to grow.

- *number of attributes*: Another parameter that can significantly affect the behavior of our algorithms is the size of A_v, i.e., the number of attributes that each node has. Whenever we change this value, we still assume that the ratio of the common and the unique attributes remains the same as in the default case, i.e., 50% for both.

- *overlap*: The final parameter we can change affects how many attributes are common and how many are unique, within the 12 attributes that correspond to each node. We will refer to the fraction of the node attributes that are common among the graph as *overlap*. As we described, the default value for the overlap is 0.5. Increasing this value means that the nodes are described mostly by the same attributes, i.e., the size of the ground set of attributes A_G is limited.

For all of these changes apart from the *overlap*, we will apply a *multiplier* to the default values that we described above. The value of the multiplier is captured in the x-axis of the relevant figures.

Performance measures

At the same time, there are multiple measurements that we can track, in order to better describe the performance of the algorithm and the changes in the retrieved solution:

- *solution size*: The first measure that captures changes in the algorithms' behavior is the size of the retrieved subgraph. We expect this size to grow, whenever the algorithm can either use more attributes to "pay" for the required edges, or, whenever it can find alternative ways to keep the subgraph connected.

- *number of preflow-push calls*: An important measure of performance is the number of times that our algorithms execute a min-cut computation on the coverage graph.

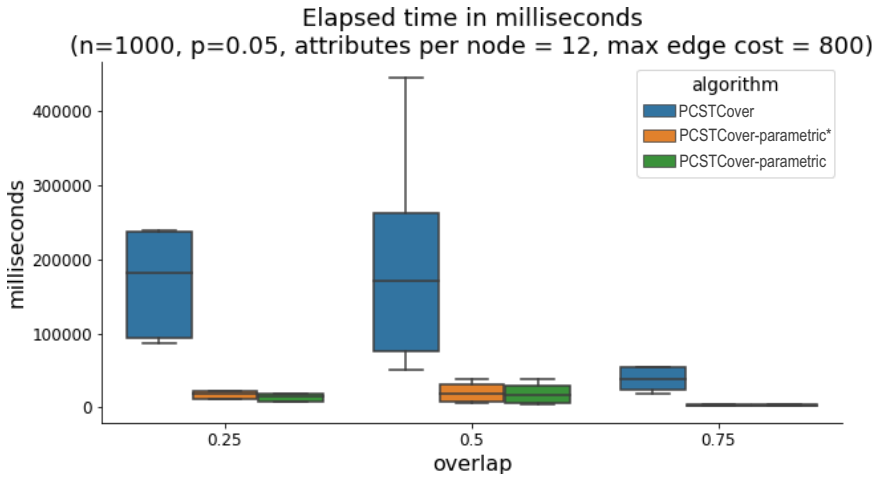
- *number of pushes*: Another important measure of performance is the number of pushes that the preflow-push algorithm is executing. This number is the aggregate sum of across all the min-cut computations.

- *execution time*: Finally, a significant but also mostly indicative measure is the time the each algorithm requires for its execution. This time is expected to change across different computer architecture, hardware, and environments, however it still provides a ball-park estimate of the performance.

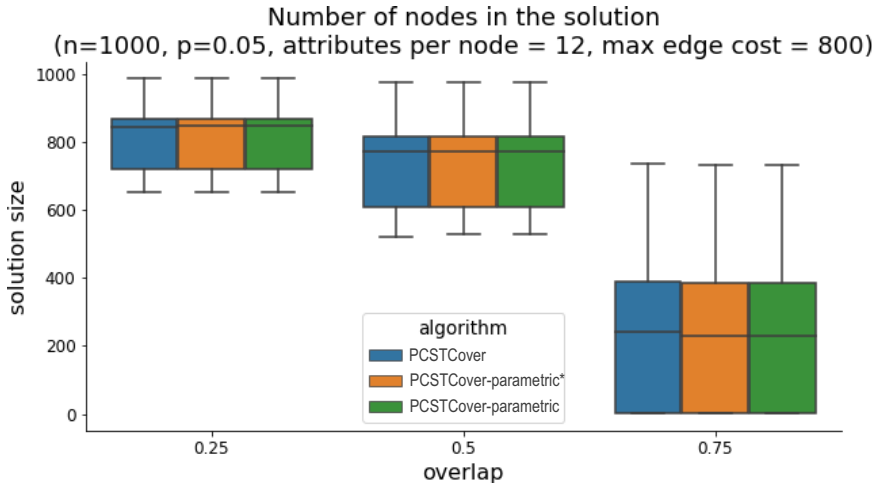
All experiments are repeated at least ten times.

Algorithms

In our experiments, we will be comparing our three algorithms, i.e., PCSTCover, PCSTCover-parametric* and PCSTCover-parametric. As a reminder, the first algorithm is executing a binary-search combined with a regular preflow-push method in order to identify the next cluster deactivation. The other two algorithms



(a) The effect of the overlap on the execution time (in milliseconds) of the algorithms.



(b) The size of the retrieved solution changes as we increase the overlap. The effect is the same across all the algorithms.

Figure 4·1

are using a discrete Newton's type of parametric-search together with a parametric preflow-push method. Finally, PCSTCover-parametric* uses a pessimistic upper bound for the parametric-search, while PCSTCover-parametric is more aggressive. In some experiments, we skip the results from PCSTCover, because its execution was too slow.

Presentation method

The figures present the results of the experiments as boxplots. This kind of plot visualizes the minimum and the maximum value for a set of executions (10 repetitions), as well as the 1st and the 3rt quartile, and the median value.

Varying the overlap

First, let us explore how changing the overlap parameter affects the performance of our algorithms. As a reminder, higher overlaps means that the unique attributes of each node are fewer than the common. This, in turn, means that the size of the ground set of attributes A_G is also smaller, since each node does not contribute much to it. Equivalently, this means that the nodes do not afford to "pay" for the more expensive edges any more.

Figures 4·1a and 4·1b capture the change in running time and in solution size respectively. Across all algorithms, we observe that although there is not big change between the values 0.25 and 0.5, increasing the overlap to 0.75 causes the algorithms to return much earlier. Figure 4·1b indicates what we mentioned previously regarding how much each node affords to "pay". The size of the solution drops as we increase the overlap in the attributes.

One thing that becomes clear in Figure 4·1a is the vast difference in running times between PCSTCover and the parametric algorithms. This difference is often more than an order of magnitude. The reason for this will become clear in the next

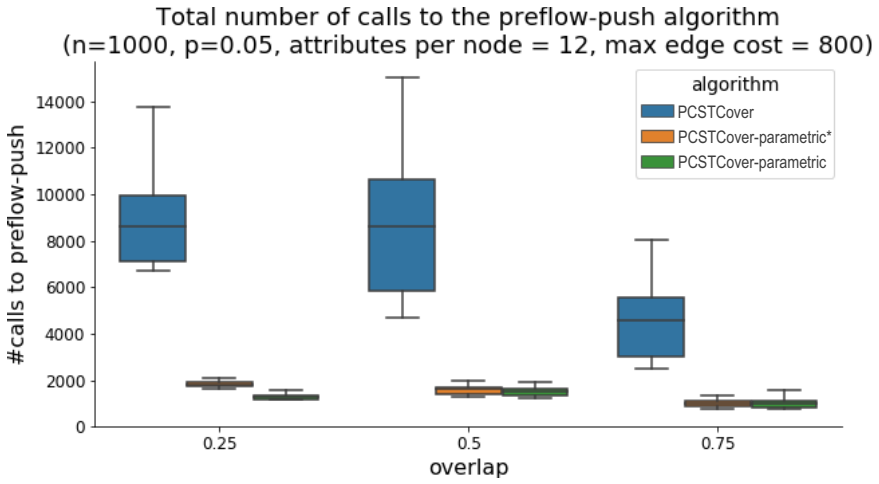
paragraph. However, for this reason, in most of the experiments we skip running PCSTCover because of time constraints.

Figures 4·2a and 4·2b explain both why PCSTCover is much slower than the parametric algorithms, and why the algorithms retrieve a solution faster as overlap grows. First, Figure 4·2a shows the number of calls that each of the algorithms is making to a minimum-cut solver. It is clear that the parametric algorithms make almost an order of magnitude fewer calls than PCSTCover. Computing the minimum-cut on the coverage graph turns out to be the biggest bottleneck of all our algorithms. Furthermore, even if that solver is extremely optimized, to run under 10ms even for large graphs, calling it thousands of times makes a huge impact in the running time. Since the parametric algorithms by design are making fewer calls (by substituting the binary search with a parametric-search, and the simple preflow-push with the parametric preflow-push), their speed advantage is obvious.

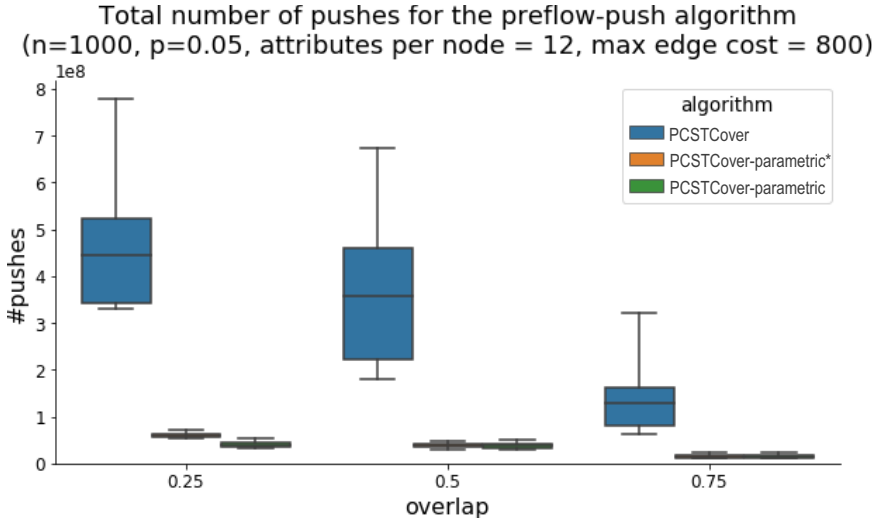
The preflow-push algorithm has two basic operations; *push* and *relabel*. By far, the most common one during the computation is *push*. In Figure 4·2b we plot the total number of pushes per execution. We observe that, as the overlap grows, the algorithm terminates earlier and does not require as many pushes as when the overlap is smaller. At the same time, we can see that there is very strong correlation between the number of calls of the preflow-push algorithm, and the total number of pushes. That is something we expect, as there is a clear trend and small variance in the number of pushes at a per-iteration level, as the algorithm progresses.

Varying the size of the input graph

The next parameter we will experiment with has to do with the size of the input graph, and specifically the number of nodes. In Figure 4·3, we plot how the number of pushes, i.e., a proxy to how long the preflow-push algorithm takes, changes as the input graphs become larger. As expected, the trend is increasing. Notice, that



(a) The total number of calls to the minimum-cut solver.



(b) The total number of pushes that the preflow-push algorithm executes during the minimum-cut computation.

Figure 4·2

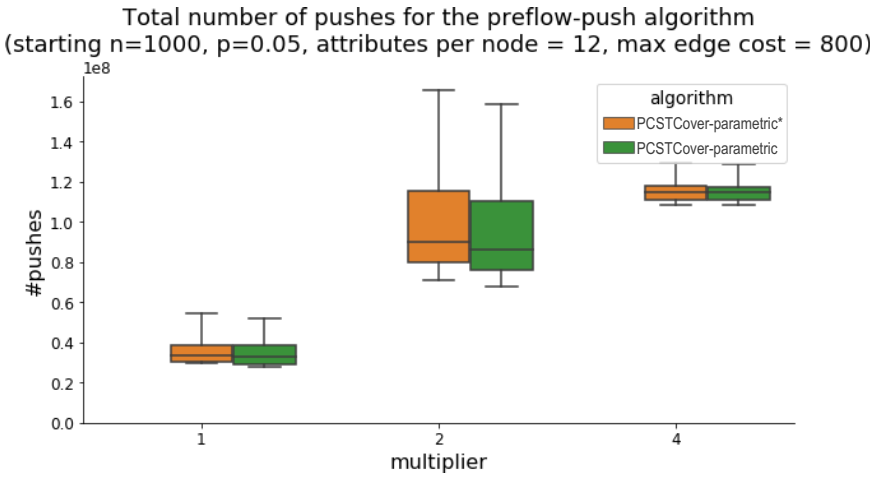


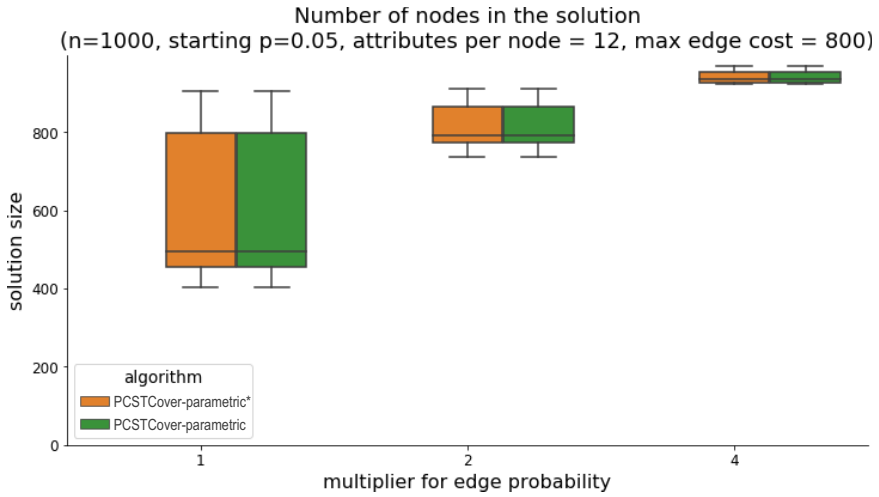
Figure 4·3: The number of pushes of the preflow-push algorithm when we increase the size of the graph. The x-axis indicates the multiplier on the number of nodes.

we only provide data points for the parametric algorithms, as PCSTCover did not scale well in practice. Furthermore, PCSTCover-parametric seems to have a slight advantage over the more naive algorithm, although the difference is not very pronounced in this experiment.

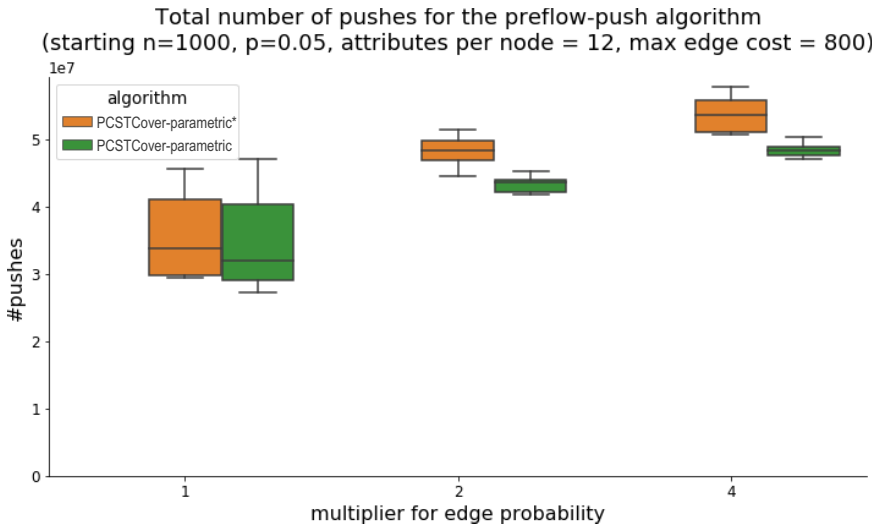
Varying the density of the input graph

The next parameter we experiment with is the density of the graph. We keep the number of nodes fixed, and we only change the number of edges. Figures 4·4a and 4·4b show the results of this experiment. Specifically, Figure 4·4a shows how the size of the retrieved solution changes as the graph becomes denser. As we mentioned before, adding more edges in the graph allows the algorithm to be more flexible in its choices. As a result, the algorithm has more opportunities to grow the solution by using alternative cheaper edges. As expected, the trend here is increasing; denser graphs result in bigger solutions.

Figure 4·4b indicates that there is a cost to be paid for adding these edges. The preflow-push algorithm becomes clearly more expensive. Additionally, in this fig-



(a) The size of the retrieved solution, as we increase the number of edges in the input graph.



(b) More dense graphs result in more expensive preflow-push executions.

Figure 4·4

ure we can clearly notice the difference between PCSTCover-parametric and PCSTCover-parametric*. While they both show an increasing trend, the former becomes more expensive more slowly. Our choice to introduce this more aggressive upper bound for λ_0 in the case of PCSTCover-parametric seems to be paying off.

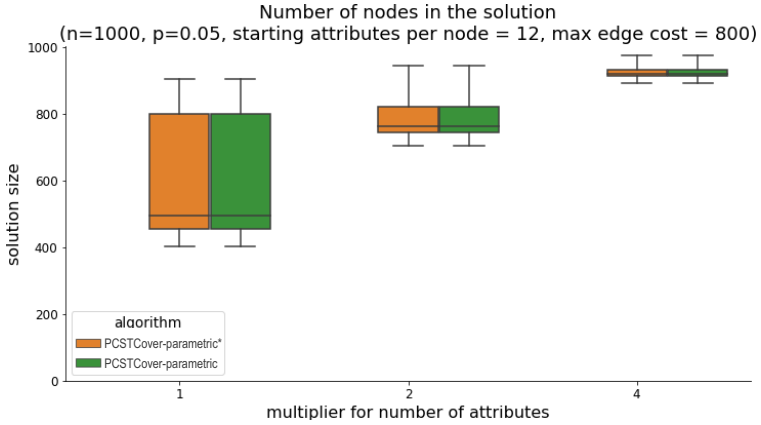
Varying the number of attributes per node

Finally, we experiment with changing the number of attributes that we assign to each node. Originally, we set this value to 12. Figure 4·5a shows that having more attributes leads to larger solutions. This is exactly supporting what we saw in Figure 4·1b, where the shortage of unique attributes resulted in smaller solutions.

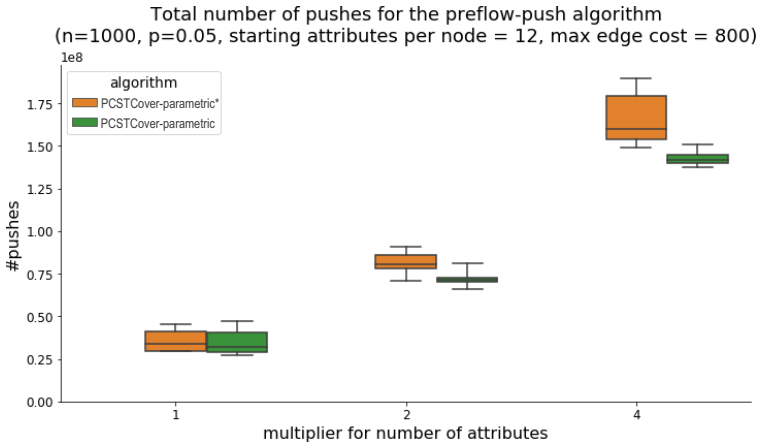
Figure 4·5b plots the number of pushes as the number of attributes per node increases. This increasing trend should be expected; more attributes per node means that the coverage graph will be larger, both in terms of nodes as well as edges. Still, PCSTCover-parametric handles this increase much more elegantly than the naive PCSTCover-parametric*. This effect is also reflected in Figure 4·5c, where we plot the execution time for each different value of the parameter.

4.6.2 Experiments on real data

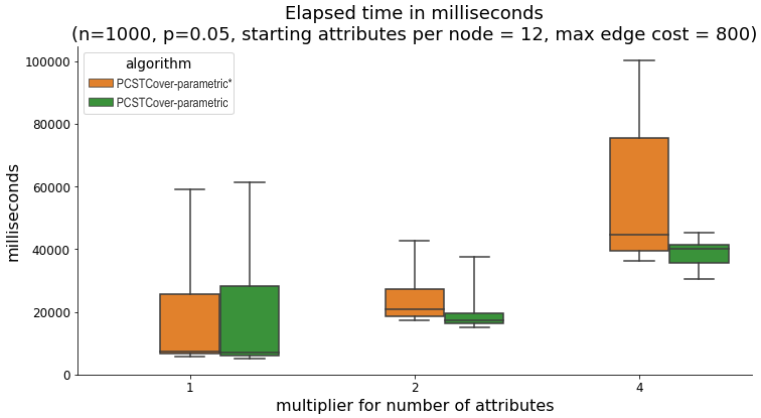
In this section, we switch our focus from the analysis of the algorithmic contributions of our work to its practical implications. More specifically, we apply our novel importance measure on a real world dataset, we show how one can preprocess a dataset in order to leverage this measure appropriately, and we conclude with some interesting insight gained from the particular dataset.



(a) The effect of changing the number of attributes per node on the size of the retrieved solution.



(b) More attributes per node require more pushes.



(c) As the attributes per node grow, the algorithms become slower.

Figure 4·5

Creating the co-authorship network

The data we use is part of the raw XML dump of DBLP[4]. This datasets contains publication records for scientific articles and books. Using this, we build the following co-authorship network. Each author is represented as a node. Whenever two authors have written a paper together, we draw an edge between the two corresponding nodes. That edge is weighted, and its weight reflects the number of times these two authors have collaborated. Furthermore, for each author we compile a list of all the venues[5] she has published in. This list will correspond to the attribute set of that particular node.

Since this network is very large and noisy, we filter the authors and keep only those that have published in the FOCS[6] conference at least once. We also remove the edges with weight equal to 1, i.e., a collaboration which only happened once. We considered such edges to be noise and not representing a scientific relationship between two researchers. This results in a network with around 2.5k authors and 7k edges. The total number of venues that appear in the union of the authors' attribute sets is around 3.5k.

Dataset statistics

Before we proceed with preprocessing, let us first compile some statistics on the data. This will allow us to build a better plan on how to proceed.

The first question we can ask to better understand the data is "Do authors like variety?". Do answer this, build a histogram on the size of the attribute set for each author. Again, this corresponds to the number of unique venues that an author has published in. We can see the plot in Figure 4·6. The x-axis is cropped at 100,

[4] `https://dblp.uni-trier.de/xml/`
[5] We filter only the `@inproceedings` records. These are mostly conferences and workshops.
[6] IEEE Annual Symposium on Foundations of Computer Science (FOCS)

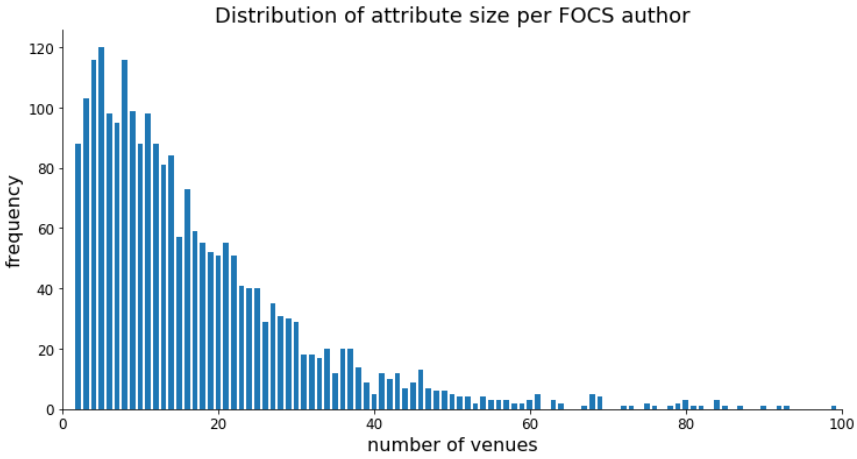


Figure 4·6: A histogram of the number of venues that each author in our network has published in. The corresponds to the size of the attribute set.

however the distribution has a long tail. The mean of this distrubution is equal to 18.7 attributes per author. Notice the shape of the distribution; there is a distinct peak around 10, and later it drops slowly until 60.

Another question one might ask is "How common is each venue?". This translates into the number of nodes that share each single attribute. Again, we can provide an answer to this question by drawing the matching histogram. As Figure 4·7 shows, most venues appear only once. The distribution quickly drops to very low numbers even before 20. In other words, even venues that appear in 20 different authors' lists are quite rare. Once again, the x-axis is cropped at 100.

Finally, one last interesting question we look into is "How frequently do researchers collaborate?". Figure 4·8 provides the answer to this, by plotting the distribution of edge weights. This is a power-law type of distribution, that quickly drops to very small numbers. Once more, the x-axis is cropped at the value 40.

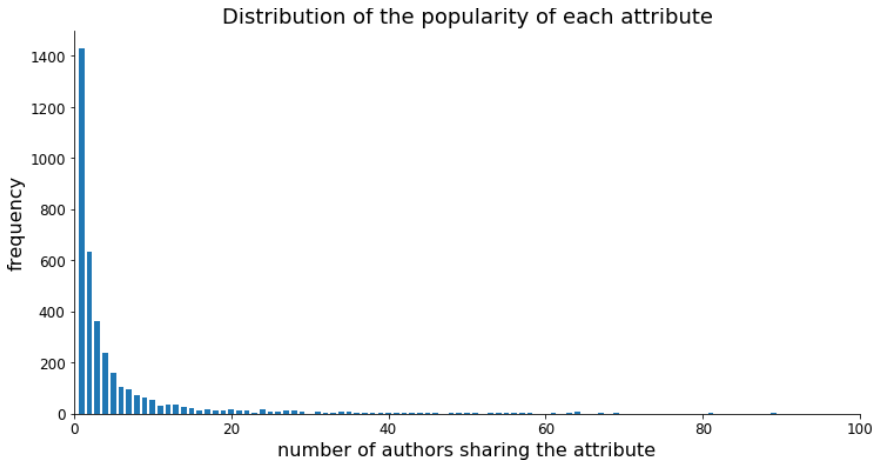


Figure 4·7: A histogram of the number of authors that share a particular venue.

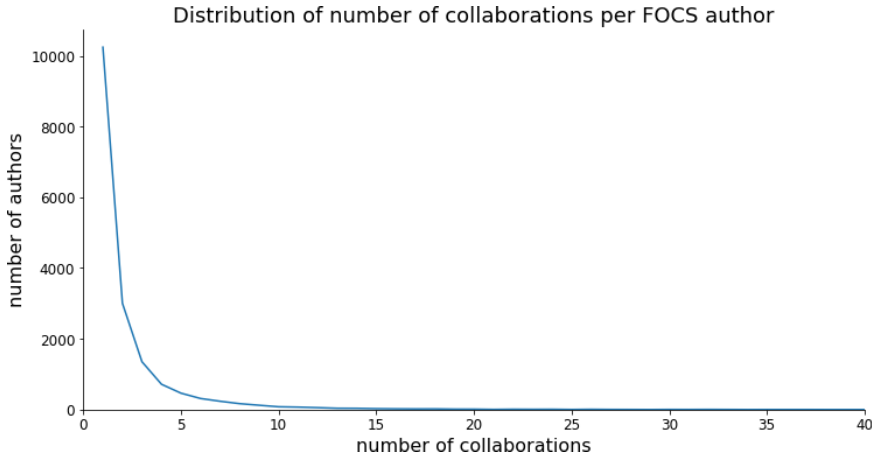


Figure 4·8: A plot of the distribution of the edge weights, i.e., the number of collaborations between two authors.

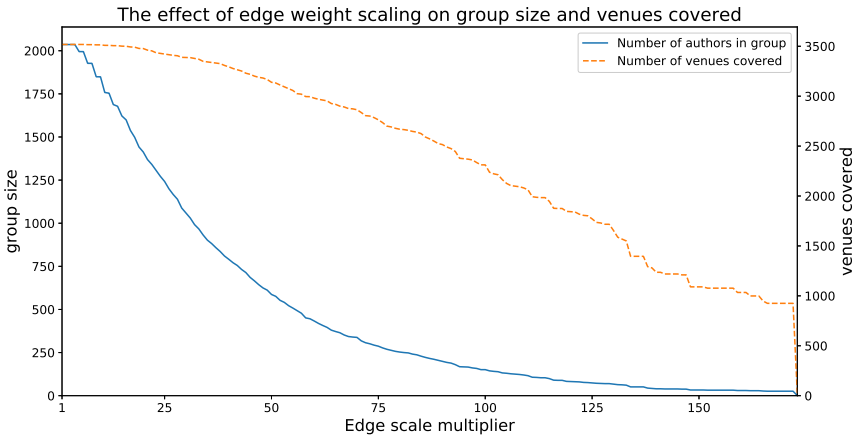


Figure 4·9: A plot of how the size of the solution and the number of covered venues in the solution change, as we change the scaling multiplier of the edge weights. The root node is Alina Ene.

Preprocessing the data

Looking back at how we defined our importance measure, we declared that a *strong and meaningful connection* between two nodes in the graph is captured by a *small edge weight*. In the network we just generated, the inverse actually holds; frequent collaborations between researchers result in large edge weight. To fix this, we subtract each edge weight from the maximum, i.e.,

$$c(e)' = \max_{(u,v)\in E} c((u,v)) - c(e) + 1 \tag{4.23}$$

Another issue that we need to deal with is that the value prize function, i.e., the number of attributes, and the cost function, i.e., the number of collaborations, are not on the same scale. One needs to decide how to rescale these values, such that the number of attributes that need to be used to "pay" for an edge of weight equal to 1 makes sense. In our case, we rescale the edge weights to $(0, 1]$ and multiply them with a predefined scaling multiplier. We experiment with how to set this value in Figure 4·9.

Picking a scaling value for the edge weights

Let us know explore how the different choices for the edge scaling multiplier affect the retrieved solution. To this end, for each different choice of multiplier, we will execute PCSTCover-parametric on that particular instance, and report the size of the group and the number of venues that the group collectively covers. We present the result of this experiment in Figure 4·9. As expected, for small multiplier values, the size of the group is large. Before dropping to zero, we get solutions of small size, i.e., less than 50. Even for these "small" groups, the number of covered attributes is fairly large. The data analyst can look at such a plot and decide what scaling she wants to use.

Comparison with PCST

As a baseline, one could only keep the number of attributes at each node and, instead of using the coverage function, formulate this problem as an instance of PCST. The question is how good would the final group be in terms of attribute coverage. To answer this, we present Figure 4·10. Again, we experiment with different values of the scaling parameter, and report the group size and the number of venues covered by the solution. For PCST, we use the PCSTFast algorithm. We observe that, although the two lines are close to each other, the line corresponding to PCSTCover-parametric is always higher. This means that for the same group size, this algorithm is returning groups that are covering more venues. This should be expected, since PCSTCover-parametric is built for this particular problem. Still, we have to note that the size of the gap between the two lines is heavily dependent on the type of dataset.

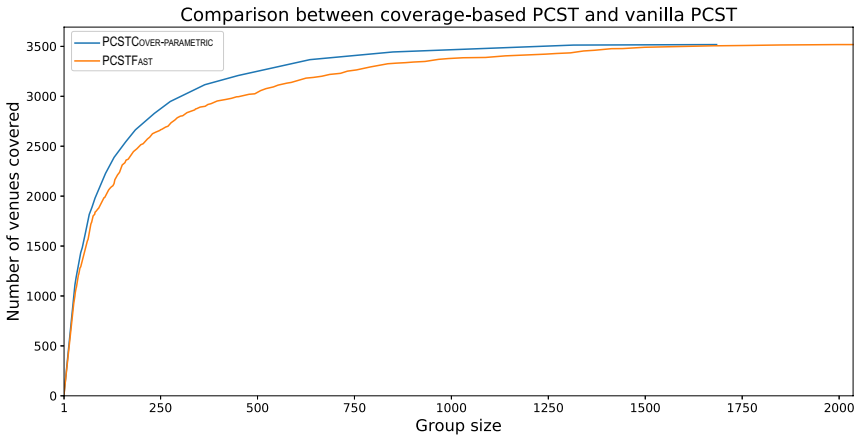


Figure 4·10: A comparison between the PCSTFast and PCSTCover-parametric in terms of number of attributes covered for a particular group size. Here, we vary the edge scaling multiplier. The root node is Alina Ene.

Case study

Finally, for a better insight into what kind of groups our algorithm retrieves, we run the following case study. We pick a scaling multiplier for the edges such that the mean of the edge weights matches the mean of Figure 4·6. We then set Aline Ene as the root node, i.e., we indicate that we want her to be a member in the retrieved group. The final solution, which can be seen in Table 4.1, selects *29 authors*, who in total cover *1110 venues* out of the 3.5k that exist in the dataset. This shows that our algorithm can indeed retrieve a small group that still exhibits big diversity in its attributes.

András A. Benczúr (51)	Thomas A. Henzinger (110)	Scott Shenker (59)
Chandra Chekuri (24)	David R. Karger (58)	Alistair Sinclair (11)
Ning Chen (84)	Phokion G. Kolaitis (35)	Daniel A. Spielman (16)
Xi Chen (242)	Orna Kupferman (51)	Shang-Hua Teng (45)
Costas Courcoubetis (45)	Rasmus Kyng (5)	Moshe Y. Vardi (111)
Xiaotie Deng (76)	Nicola Leone (49)	Mihalis Yannakakis (43)
Alina Ene (18)	Liang Li (78)	Yitong Yin (9)
Georg Gottlob (99)	Pinyan Lu (23)	Peng Zhang (170)
Rachid Guerraoui (84)	Fabio Martinelli (117)	Yuan Zhou (85)
David Harel (111)	Christos H. Papadimitriou (69)	

Table 4.1: The members of the group with Alina Ene as the root node. The number next to each name corresponds to the number of venues each author has published in. The total number of unique venues covered from this group is 1110.

Chapter 5

Conclusion

In this thesis, we studied the problem of evaluating the importance of nodes and edges in graphs. This research problem has received a lot of attention, especially due to the popularity of graphs as a means for data representation. We presented a classification of these measures into two big categories, depending on their goal; some measures target nodes and entities as singleton entities, while others are tailored to evaluate groups of entities. We then focused on three importance measures, two of which were first introduced in this dissertation. Furthermore, we provided novel, efficient and theoretically-sound algorithms, which aim at making the computation of these measures practical for real-world scale datasets.

The first importance measure we discussed is the *spanning edge centrality*. This falls under the category of ranking-based measures, which evaluate graph entities as singletons. We proposed a novel algorithmic framework that allows the data analyst to find the most important edges in large graphs realistically fast. We also showed how this framework can be used to enable the fast computation of other related measures as well. Finally, we provided evidence indicating that the *spanning edge centrality* has a direct connection with information propagation processes on a network, and can be used to strategically stop such processes early on.

Then we introduced the *absorbing random walk centrality*, a novel selection-based measure, that gauges the importance of groups of nodes. In fact, this measure allows the analyst to select a set of query nodes and finds a group of nodes, which are central

with respect to the query. We proposed an approximation algorithm that retrieves such important groups and showed how it can be used in practical settings.

Finally, we focus on a particular sub-category of selection-based measure: those that are parameter-free. Instead of specifically predetermining the size of the retrieved group, these measures leverage a prize function to guide this decision. We assume that the nodes in our graphs are also assigned a set of attributes, and propose a novel importance measure that captures the following intuition: an important group needs to be compact and to contain as many attributes as possible. We show how this measure can be connected to the problem of the prize-collecting Steiner tree, and design efficient algorithms with theoretical guarantees. Last, we showcase how this measure behaves on both synthetic and real datasets, and how it can prove useful to the data analyst.

References

Agrawal, R., Gollapudi, S., Halverson, A., and Ieong, S. (2009). Diversifying search results. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 5–14. ACM.

Angel, A. and Koudas, N. (2011). *Efficient diversity-aware search*. ACM.

Anthonisse, J. M. (1971). The rush in a directed graph. *Mathematische Besliskunde*, (BN 9/71).

Bader, D. A., Kintali, S., Madduri, K., and Mihail, M. (2007). Approximating betweenness centrality. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 124–137. Springer.

Balas, E. (1989). The prize collecting traveling salesman problem. *Networks*, 19(6):621–636.

Batagelj, V. and Zaversnik, M. (2003). An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*.

Bavelas, A. (1948). A mathematical model for group structure. *Human Organizations*, 7.

Bienstock, D., Goemans, M. X., Simchi-Levi, D., and Williamson, D. (1993). A note on the prize collecting traveling salesman problem. *Mathematical programming*, 59(1-3):413–420.

Boldi, P. and Vigna, S. (2014). Axioms for centrality. *Internet Mathematics*.

Bollobas, B. (1998). *Modern Graph Theory*. Springer.

Borgatti, S. P. (2005). Centrality and network flow. *Social Networks*, 27.

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

Brandes, U. and Fleischer, D. (2005). Centrality measures based on current flow. In *Annual symposium on theoretical aspects of computer science*, pages 533–544. Springer.

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

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30.

Cherkassky, B. V. and Goldberg, A. V. (1997). On implementing the pushrelabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410.

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

Doyle, P. G. and Snell, J. L. (1984). *Random walks and electric networks*, volume 22. Mathematical Association of America.

Everett, M. G. and Borgatti, S. P. (1999). The centrality of groups and classes. *The Journal of mathematical sociology*, 23(3):181–201.

Feofiloff, P., Fernandes, C. G., Ferreira, C. E., and De Pina, J. C. (2010). A note on johnson, minkoff and phillips' algorithm for the prize-collecting steiner tree problem. *arXiv preprint arXiv:1004.1437*.

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41.

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

Gallo, G., Grigoriadis, M. D., and Tarjan, R. E. (1989). A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55.

Garey, M. and Johnson, D. (1990). *Computers and intractability; A guide to the theory of NP-completeness*. W. H. Freeman & Co.

Geisberger, R., Sanders, P., and Schultes, D. (2008). Better approximation of betweenness centrality. In *Proceedings of the Meeting on Algorithm Engineering & Experiments*, pages 90–100. Society for Industrial and Applied Mathematics.

Goemans, M. X. and Williamson, D. P. (1995). A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317.

Goldberg, A. V. and Tarjan, R. E. (1988). A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940.

Golub, G. H. and Van Loan, C. F. (2012). *Matrix computations*, volume 3. JHU Press.

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

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

Harris, J. M., Hirst, J. L., and Mossinghoff, M. J. (2008). *Combinatorics and Graph Theory*. Undergraduate Texts in Mathematics, Springer.

Hegde, C., Indyk, P., and Schmidt, L. (2014). A fast, adaptive variant of the goemans-williamson scheme for the prize-collecting steiner tree problem. In *Workshop of the 11th Discrete Mathematics and Theoretical Computer Science (DIMACS) Implementation Challenge*.

Hegde, C., Indyk, P., and Schmidt, L. (2015). A nearly-linear time framework for graph-structured sparsity. In *International Conference on Machine Learning*, pages 928–937.

Huitema, C. (2000). *Routing in the Internet*. Prentice Hall.

Ishakian, V., Erdös, D., Terzi, E., and Bestavros, A. (2012). A framework for the evaluation and management of network centrality. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 427–438. SIAM.

Johnson, D. S., Minkoff, M., and Phillips, S. (2000). The prize collecting steiner tree problem: theory and practice. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, volume 10, page 4.

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

Kang, U., Papadimitriou, S., Sun, J., and Tong, H. (2011). Centralities in large networks: Algorithms and observations. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pages 119–130. SIAM.

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer.

Katz, L. (1953). A New Status Index Derived from Sociometric Index. *Psychometrika*.

Kempe, D., Kleinberg, J., and Tardos, É. (2003). Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146. ACM.

Kolaczyk, E. D., Chua, D. B., and Barthélemy, M. (2009). Group betweenness and co-betweenness: Inter-related notions of coalition centrality. *Social Networks*, 31(3):190–203.

Koutis, I., Miller, G. L., and Peng, R. (2011a). A nearly-m log n time solver for sdd linear systems. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 590–598. IEEE.

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

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

Langville, A. N. and Meyer, C. D. (2005). A survey of eigenvector methods for web information retrieval. *SIAM review*, 47(1):135–161.

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

Leavitt, H. J. (1951). Some effects of certain communication patterns on group performance. *The Journal of Abnormal and Social Psychology*.

Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J., and Glance, N. (2007). Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 420–429. ACM.

Mavroforakis, C., Garcia-Lebron, R., Koutis, I., and Terzi, E. (2015a). Spanning edge centrality: Large-scale computation and applications. In *Proceedings of the 24th international conference on world wide web*, pages 732–742. International World Wide Web Conferences Steering Committee.

Mavroforakis, C., Mathioudakis, M., and Gionis, A. (2015b). Absorbing random-walk centrality: Theory and algorithms. In *IEEE International Conference on Data Mining (ICDM)*, pages 901–906. IEEE.

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

Nikolakaki, S. M., Mavroforakis, C., Ene, A., and Terzi, E. (2018). Mining tours and paths in activity networks. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web (WWW)*, pages 459–468. International World Wide Web Conferences Steering Committee.

Noh, J. D. and Rieger, H. (2004). Random walks on complex networks. *Physical Review Letters*, 92(11):118701.

Perlman, R. J. (1985). An algorithm for distributed computation of a spanning tree in an extended lan. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

Royle, G. and Godsil, C. (1997). *Algebraic Graph Theory*. Graduate Texts in Mathematics. Springer Verlag.

Rozenshtein, P., Anagnostopoulos, A., Gionis, A., and Tatti, N. (2014). Event detection in activity networks. In *Proceedings of the 20th ACM International Conference on Knowledge Discovery and Data mining (KDD)*, pages 1176–1185. ACM.

Sabidussi, G. (1966). The centrality index of a graph. *Psychometrika*, 31.

Shi, J. and Malik, J. (2000). Normalized cuts and image segmentation. *IEEE transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905.

Shimbel, A. (1953). Structural parameters of communication networks. *Bulletin of Mathematical Biology*, 15.

Spielman, D. A. and Srivastava, N. (2011). Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926.

Tarjan, R. E. (1974). A note on finding the bridges of a graph. *Information Processing Letters*, 2(6):160–161.

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

Tutte, W. (2001). *Graph Theory*. Cambridge University Press.

Vieira, M. R., Razente, H. L., Barioni, M. C. N., Hadjieleftheriou, M., Srivastava, D., Traina, A. J. M., and Tsotras, V. J. (2011). On query result diversification. *2011 IEEE International Conference on Data Engineering*, pages 1163–1174.

Von Luxburg, U. (2007). A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416.

Zhu, X., Goldberg, A., Van Gael, J., and Andrzejewski, D. (2007). Improving diversity in ranking using absorbing random walks. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 97–104.

Curriculum Vitae