AL-BAGHDADI, AHMED, MS. MAY 2017 COMPUTER SCIENCE

COMPUTING TOP-K CLOSENESS CENTRALITY IN UNWEIGHTED UNDIRECTED GRAPHS REVISITED

Thesis Advisor: Feodor Dragan

Centrality indices are essential concepts in network analysis. Closeness centrality is the very popular and most widely used centrality in the analysis of real-world complex networks. Closeness of a vertex is the inverse of the average distance to each other vertex, it shows how important and influential the vertex is. In particular, the current best algorithm of selecting the k most central vertices in unweighted undirected graphs is based on the All-Pairs-Shortest-Path approach (in short, APSP). However, in practice finding the most k central vertices in a dataset is computationally intractable, even for sparse graphs, since it requires a quadratic running time in the worst case. Fortunately, in this problem we are not required to compute the exact closeness centrality for all vertices, to save time, we can cut computation after finding the k vertices with the most closeness value. Bergamini et al. [7] proposed a new algorithm for computing top-k ranking in unweighted graphs. Their work is based on computing a lower bound on total distances of every vertex and stopping the search when k vertices already have lower total distances than the bounds computed for the other vertices. In Bergamini et al. algorithm, the tightness of the bounds dramatically influences the algorithm performance. In this work, first, we propose a new method of computing lower bounds on total distances of vertices to replace the method of computing lower bounds in Bergamini et al. algorithm. We achieve excellent improvements through replacing the method of computing lower bounds in Bergamini et al. algorithm by our method of computing lower bounds. In our experiments on real-world networks we

show that our new method of computing bounds combined with top-k ranking algorithm of Bergamini et al. significantly improves computation of finding the k most closeness vertices over the APSP and Bergamini et al. algorithm. Second, our method of computing lower bounds can be used as an approximation algorithm for finding the median vertex in a graph; the median of a graph is a vertex with the highest closeness centrality. We validate our approximation algorithm experimentally on various datasets from different domains, such as biology, social networks, collaboration networks, etc.

COMPUTING TOP-K CLOSENESS CENTRALITY IN UNWEIGHTED UNDIRECTED GRAPHS REVISITED

A thesis submitted
To Kent State University in partial
Fulfillment of the requirements of the
Degree of Master of Science

by

Ahmed Al-Baghdadi

April, 2017

Thesis written by

Ahmed Al-Baghdadi

B.S., University of Al-Qadisiyah, 2011

M.S., Kent State University, 2017

Approved by

Feodor Dragan, Advisor,

Javed I. Khan, Chair,Department of Computer Science

James L. Blank, Dean, College of Arts and Sciences

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and for the most, I would like to thank and praise Allah for blessing us with the ability and perseverance to complete this thesis.

I would like to express my appreciation to my supervisor Professor Feodor Dragan who helped in every step of the way. Professor Dragan has believed in me and never hesitate to provide relentless support and motivation at all times. His continuous support, guidance, and encouragement will not be forgotten. Without his guidance and constant feedback this thesis would not have been achievable.

I would like to acknowledged all my professors at the Computer Science department at Kent State University. My graduate experience benefitted greatly from the courses and learning environment that the department of Computer science provides.

I would like to thank in advance the committee members Professor Hassan Peyravi and Professor Xiang Lian for the extensive time, effort, and expertise that they devote towards this thesis.

Finally, I would like to acknowledge my sponsor HCED-Iraq for providing this opportunity to study abroad. HCED-Iraq provides all the guidance and financial support to complete my master.

Dedication

I would like to dedicate this work to my loving father Hussein, mother Manahil, and second mother Hoyam who are everything in my life.

I would like also to dedicate this work to my brothers and sisters: Ali, Osamma, Sarah, Larah, and Samah who are beside me all the time. And to my beautiful nephews Fatima, Abdullah, and Zainab who I miss the most.

Finally, for all the laugh and great memories, I would like to dedicate this thesis to my cousins and friends back in Iraq and here in the U.S. who are there whenever I need them. My friend Illia, thank you for all your help and support!

I love you all!

God bless you!

Chapter 1

Introduction

1.1 Introduction

Discovering the most central nodes in a real-world network is a fundamental problem that rises in many research areas, such as biology, computer science, sociology, and psychology [9]. Although many centrality measures have been introduced in literature (see survey [8]), the graph closeness centrality is the most popular and widely used one among those centralities. Informally, the closeness centrality of a node v is the inverse of the average shortest path distance [7]. Intuitively, the most central node is a node that can reach the rest of the graph quickly. The centrality of a node tells how efficient the node is in spreading information to other nodes. Also, nodes with the highest closeness centrality are considered the most influential nodes in the network.

The importance of closeness centrality comes from its real-world applications. One example is facility location: a company planning to open a store in a city might be interested in knowing one or few locations that are close, on average, to a large number of potential customers [22]. In viral marketing, the crucial question is to find a group of people in a network who can trigger the most efficient product adoption [18]. National security, power grid administration, and computer network management are other applications that stated in [21] to benefit from the discovery of nodes with most closeness centrality in a network.

Given an unweighted undirected graph $G = (V, E)$ and an integer k. Finding the k nodes with highest closeness centrality values is related to the All-Pairs-Shortest-Path problem (APSP). It requires the computation of distances for all pairs of nodes. Williams and Vassilevska [24] proposes fast matrix multiplication for solving the top-k closeness centrality problem in $O(n^{2.373})$ time, where n is the number of vertices. However, this approach is not practical due to its memory requirements $O(n^2)$ time. A more practical approach is to run BFS for each node in the network. The complexity of the second approach is $O(n(n + m))$ time, where m is the number of edges. Since real-world networks are often sparse, the BFS approach is more suitable. Unfortunately, the quadratic running time of the second approach makes it prohibitive to be used in real-world networks with few millions of nodes.

Just recently Bergamini et al. [7] propose an algorithm for finding the k most central vertices in an unweighted graph, we call it top-k ranking. The basic idea of their algorithm is to discover a lower bound on total distances for each vertex. The algorithm works by running BFS from vertices to compute total distances and stops computation when k vertices whose total distances values are lower than the lower bounds on total distances of the other nodes. In their experiments on real-world networks Bergamini et al. achieve significant improvements over [9] and [21]. However, in the worst case the three algorithms are not guaranteed to be faster than APSP approach, not surprising since it was proven that the problem of finding vertices with highest closeness centrality is equivalent to APSP problem [7].

The performance of Bergamini et al. top-k ranking algorithm is dramatically influenced by the tightness of lower bounds on total distances of vertices. In our work, we propose a new method of computing a lower bound for each vertex. Through replacing the method of computing lower bounds in Bergamini et al. top-k ranking algorithm by our method of computing lower bounds (LPLB : Layering Partition Lower Bound) we achieve excellent improvements. We refer to Bergamini et al. method of computing lower bounds as Level-

Based Lower Bound, for short LBLB. Our method LPLB of computing lower bounds is based on the layering partition procedure proposed by V. Chepoi and F. Dragan [13]. We decompose graph into a tree like metric graph, Γ tree. Then, we compute exact distances on Γ tree and use them to assign lower bounds to the vertices of the original graph.

Our method of finding lower bounds, Layering Partition Lower Bound, provides bounds to vertices that are very close to exact values of total distances. If the lower bounds produced by LPLB are not close by number to exact total distances values of the graph's vertices, they are close by the order. For example, let us assume that a vertex x is the most central vertex in the graph. x will be assigned one of the best (lowest) lower bounds by LPLB algorithm. This finding encouraged us to propose an approximation algorithm for finding a median vertex in a graph. A median vertex in a graph is a vertex with highest closeness centrality. It is a vertex that is the closest on average to all other vertices in the graph.

Our algorithm LPLB of finding lower bounds finds bounds on total distances for vertices that are always better than the bounds that are found by Bergamini et al. method LPLP. In our experiments on large collection of real-world networks, we show that our algorithm LPLB of finding lower bounds combined with Bergamini et al. top-k ranking algorithm drastically improves the APSP approach. In all the experiments LPLB combined with Bergamini et al. top-k ranking algorithm [7] is mostly better, and never worse, than Bergamini et al. $FastClos_L$ algorithm. However, in general graphs and in the worst case Bergamini et al. and our method are not guarantied to be better than APSP approach. We show also experimentally that our method LPLB of finding lower bounds can help in finding a median vertex of a graph very fast. In our experiment, we show that after the preprocessing we need only to examine 5% of the graph's vertices to find a median of the graph.

1.2 Problem Definition and Notation

Given an unweighted undirected graph $G = (V, E)$ and an integer k, we want to find the k most central vertices in G. G is a strongly connected graph with V, E set of vertices and edges respectively. The group k of most central vertices is the vertices in G that are close, on average, to all other vertices. Let's denote the closeness centrality of a vertex v, $c(v)$ Then

$$c(v) = \frac{n-1}{S(v)}, \tag{1.1}$$

where n is the number of vertices, $n = |V|$, and $S(v)$ is the total distances, farness, from the vertex v to all other vertices of G.

$$S(v) = \sum_{u \in V} d_G(v, u). \tag{1.2}$$

The total distances, farness, from a vertex v is the sum of distances from v to all the other vertices in G. $d_G(v, u)$ denotes the number of edges in a shortest path from v to u. We define $\Gamma = (Q, E')$ is the graph tree resulting from layering partition $(LP(G, s))$ of the graph G with a starting vertex s. Q, E' denote set of nodes and edges of Γ respectively. Table 1.1 summarizes notations.

1.3 Related Work

Closeness centrality is one of the most popular and used centrality in network analysis. Due to its real-world applications, closeness has been studied extensively in literature. It was proven in [1] that the complexity of discovering a node with the best closeness value is related to the APSP problem. However, computing distances for all pairs of vertices in the network may not be even feasible in a network with few millions of vertices. As we discussed, even the proposed algorithm in [24] is, feasibly, prohibited to use in large networks.

Symbol	Definition
$G = (V, E)$	Graph with V, E sets of vertices and edges respectively
n	Number of vertices = $\lvert V\rvert$
m	Number of edges = $\lvert E\rvert$
$\Gamma = (Q, E')$	A tree graph with Q, E' sets of nodes and edges respectively
q	Number of nodes = $\lvert Q\rvert$
q'	Number of edges = $\lvert E'\rvert$
$d_G(v, u)$	The number of edges in a shortest path between u and v in G
$W(v)$	The number of vertices residing in the cluster v
$d_\Gamma(v, u)$	The number of edges in the shortest path between u and v in Γ
$LP(G, s)$	Layering partition of a graph G from a starting point s
$c(v)$	Closeness centrality of v
$S(v)$	The total distances, farness, of v
$\sigma(\Gamma)$	The dismantling ordering of the graph tree Γ
LPLB	Layering partition lower bound
LBLB	Level-based lower bound

Table 1.1: *Summary of Notation*

Some effort has been devoted to approximation algorithms. An approximation algorithm for computing closeness values for all vertices in the network with an adaptive error ϵ proposed by Eppstein et al. [16]. Although approximation algorithms might give a fast solution, they may fail at preserving ranking of vertices since the difference between centrality of some vertices might be very small [9].

The problem of computing the k nodes with the top closeness centralities with an accurate ranking has been studied in [21] and [9]. Just recently, Bergamini et al. [7] proposed an algorithm for extracting the k vertices with top closeness centralities of unweighted graph. In their algorithm, a lower bound on total distances is computed for each vertex. A BFS based algorithm computes exact total distances values of vertices and stops computation when k vertices with total distances lower than the lower bounds for the other vertices is found. In Bergamini et al. work, the computation of lower bounds is based on the fact that the distance between any two nodes u at level i and v at level j and $i \leq j$, is at least $j - 1$. The computation for a very large number of vertices will be skipped if the lower bounds are close to the exact values of total distances.

Experimentally, Bergamini et al. out performs all the current algorithms of finding the k vertices with the best closeness such as [9]. However, the worst case complexity of all proposed algorithms is not better than the worst case complexity of APSP approach.

Chapter 2

Theory Review and Datasets

2.1 Theory Review

In this section we will go through literature reviews for graph definition, representation, and basic graph algorithms, such as BFS.

2.1.1 Graph

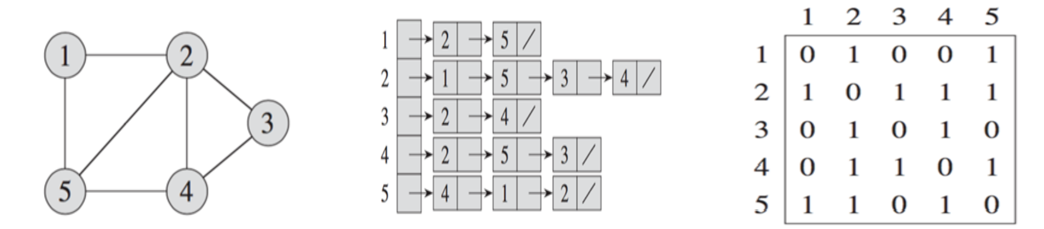
A graph G is a pair (V, E), where V is a set of objects (items) and E is a set of binary relations on V [14]. V is called the *vertex set* of G, the elements of V are called *v*ertices. E is called the *edgeset* of G, the elements of E are called *edges*. A graph G can be directed or undirected. We use notation (u, v) to represent edge that connects the two vertices u, v. In an undirected graph G, we say u, v are *adjacent* if and only if the edge $(u, v) \in E$. The *degree* of a vertex in the undirected graph G is the number of edges incident on it. We denote the degree of a vertex v by $deg(v)$.

A path of length k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\{v_0, v_1, .., v_k\}$ of vertices such that $u = v_0$ and $u' = v_k$, and $(v_{(i-1)}, v_i) \in E$ for $i = 0, 1, .., k$. The number of edges on the path denotes the length of the path [14]. A path P is said to be a simple path if all vertices in P are distinct.

In an undirected graph a path $P = v_0, v_1, ..., v_k$ forms a cycle if $v_0 = v_k$. The cycle is called simple if, $v_0, v_1, ..., v_k$ are distinct. We say that $G' = (V', E')$ is a *subgraph* of the graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. If we are given $V' \subseteq V$, the subgraph induced by V' is the graph $G' = (V', E')$, where $E = \{(u, v) \in E' : u, v \in V'\}$

2.1.2 Representation of Graphs

There are two standard ways to represent a graph $G = (V, E)$: adjacency list or adjacency matrix [14]. The two representations are applicable for directed and undirected graphs. However, the adjacency list is the more compact way to represent sparse graphs (real-world), since real-world graphs are mostly sparse [7]. The graph algorithms in this thesis assume the input graph to be represented as an adjacency list. Figure 2.1 shows the two representations of an undirected graph. 2.1(a) shows an undirected graph with $|V| = 5$ and $|E| = 7$, 2.1(b) the adjacency list representation, and 2.1(c) the adjacency matrix representation of the graph.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

((a)) An undirected graph ((b)) Adjacency list representation ((c)) Adjacency matrix representation

Figure 2.1: The two representations of an undirected graph

2.1.3 Undirected Graph

A graph G is called undirected graph when edges have no orientation [3]. In an undirected graph G the edge (v, u) is identical to the edge (u, v). The maximum number of edges in any undirected graph is $n(n-1)/2$ [3]. The graph algorithms in this thesis assume the input graph to be undirected. Figure 2.2 shows an undirected graph G with 6 vertices and 4 edges.

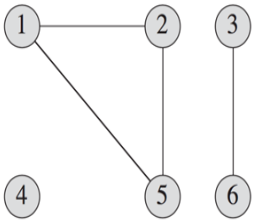


Figure 2.2: An undirected graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$, and $E = \{(1, 2), (2, 5), (1, 5), (3, 6)\}$

2.1.4 Connected Components

A subgraph of an undirected graph G is called a connected component if and only if there is a path that connects any two vertices of that subgraph [14]. Figure 2.3 shows a graph G with three connected components. The graph algorithms in this thesis assume the input graph to be connected, if not, then we extract the largest connected component.

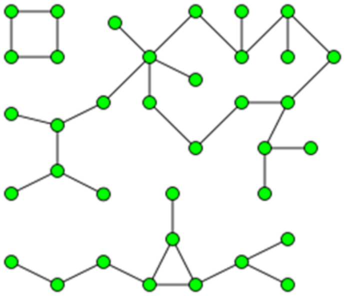


Figure 2.3: A graph G with three connected

2.1.5 Distance Between Vertices

The distance between any two vertices u, v in a graph G is the number of edges in a shortest path connecting them. If there is no path connecting u, v, they belong to different connected components. We denote the distance between any two vertices u, v in G by $d_G(u, v)$. The diameter of a graph G, $dim(G)$, is the greatest distance between any pair of vertices $u, v \in G$. Formally, $dim(G)$ of a set of vertices $Y \subseteq V$ is $max_{(u,v \in Y)}\ d_G(u, v)$.

2.1.6 Breadth-First Search

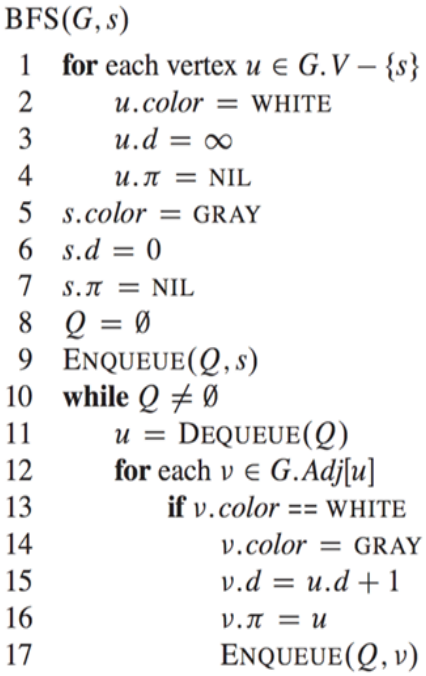
Breadth-First Search (BFS) is one of the fundamental algorithms for exploring a graph. Many important graph algorithms, such as Prim's minimum spanning tree and Dijkstra's single-source shortest-path algorithms, use concepts related to those in Breadth-First Search [14].

Given a graph $G = (V, E)$ and a source vertex s, $BFS(G, s)$ explores the edges of G to discover every vertex that is reachable from the starting vertex s. It computes the shortest path from s to every reachable vertex. $BFS(G, s)$ procedure produces a breadth-first-search tree of the graph G rooted at s. The breadth-first-search tree contains all shortest paths from s to any reachable vertex v.

In order to track the progress, vertices will be colored white, gray, or black. At the beginning all vertices are white. A vertex is colored black or gray while it is discovered for the first time. If $(u, v) \in E$ and vertex u is black, vertex v is either gray or black. All vertices that attached (adjacent) to a black vertex have been discovered. Gray vertex could have white neighbors.

The construction of breadth-first-search tree is started from the root vertex s. Whenever a white vertex v is discovered in the scanning process of the adjacency list of an already visited vertex u, the vertex v and the edge (u, v) is added to the breadth-first-search tree. We say that u is the parent of v in the tree. Every vertex in the breadth-first-search tree has at most one parent. If u is on the simple path in the breadth-first-search tree from s to v, then u is an ancestor of v and v is a descendant of u.

Figure 2.4 shows the pseudocode of breadth-first search algorithm. The input graph $G = (V, E)$ is assumed to be represented using adjacency list. Several additional attributes are added to each vertex in the graph. The color of each vertex is stored at u. color and the parent of u in $u.\pi$. At the beginning the parent of each vertex $u.\pi = NIL$. $u.d$ holds the distance from the source vertex s to u.



```
BFS($G, s$)
 1  for each vertex $u \in G.V - \{s\}$
 2      $u.color$ = WHITE
 3      $u.d = \infty$
 4      $u.\pi$ = NIL
 5  $s.color$ = GRAY
 6  $s.d = 0$
 7  $s.\pi$ = NIL
 8  $Q = \emptyset$
 9  ENQUEUE($Q, s$)
10  while $Q \neq \emptyset$
11      $u$ = DEQUEUE($Q$)
12      for each $v \in G.Adj[u]$
13          if $v.color$ == WHITE
14              $v.color$ = GRAY
15              $v.d = u.d + 1$
16              $v.\pi = u$
17              ENQUEUE($Q, v$)
```

Figure 2.4: Breadth-First Search procedure

The BFS procedure works as follows. In line $1 - 4$ every vertex, except s, is colored white, distance $u.d$ is set to be infinity, and the parent is set to be NIL. In line 5, s is painted gray, since it is already discovered. Line 6 initializes $s.d = 0$, and line 7 sets parent of s to be NIL. Line $9 - 10$ initialize a queue Q. The while loop of lines $10 - 18$ iterates while there remain gray vertices. Line 11 determines and removes the gray vertex u from the head of the queue. Lines $12 - 17$ deal with each vertex v is in the adjacency list of u. In case v is white, it has not been explored, and we will discover it by executing lines $14 - 17$. We paint v gray, and set its distance $v.d = u.d + 1$ and save u as v's parent, $v.\pi = u$. Once all the adjacent vertices of u have been examined, the procedure paints u black. Figure 2.5 shows the process of BFS procedure on a simple example.

The total running time of the breadth-first search (BFS) procedure on a graph $G = (V, E)$ is $O(V + E)$. It runs in linear time in the size of adjacency list of representation of G.

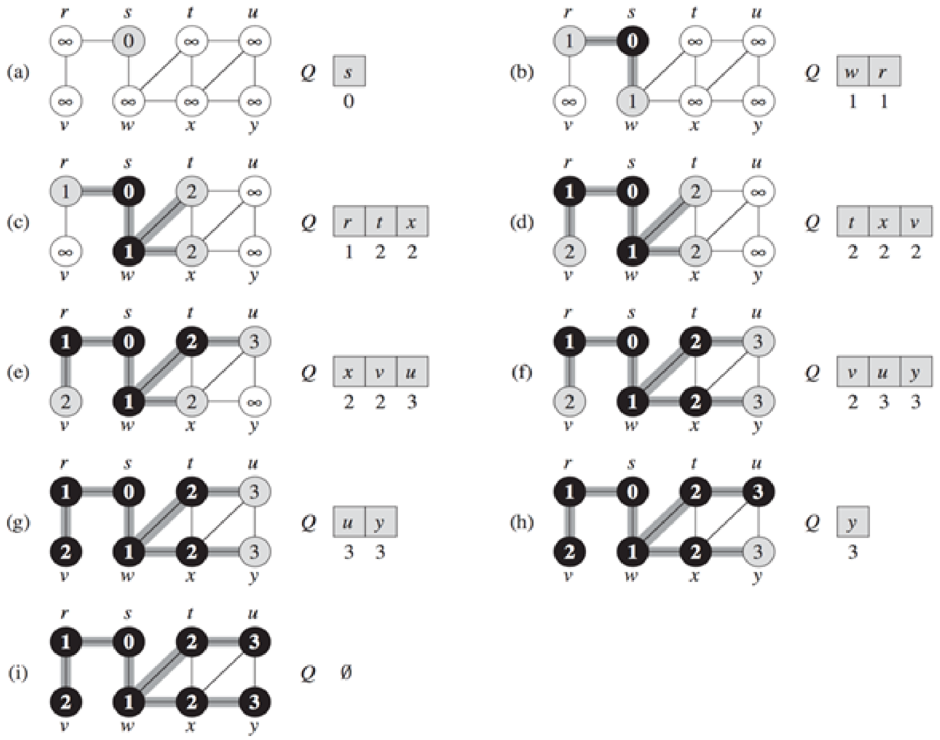


Figure 2.5: BFS procedure works on an undirected graph. Tree edges are shown as they are processed by the BFS

2.1.7 Trees

A tree $T = (V, E)$ is a connected, acyclic graph. Any two nodes $u, v \in V$ are connected by a unique simple path [14]. In case any edge is removed from E, the resulting graph is disconnected. T is also called a free tree, shown in Figure 2.6.

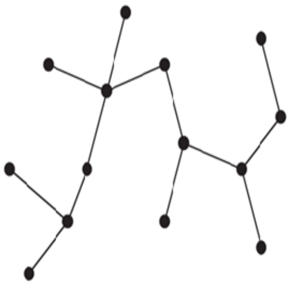


Figure 2.6: A free tree

A rooted tree is a tree in which one of the nodes is distinguished from the others [14]. The distinguished node is called the root of the tree. Consider a rooted tree T at r and a node x. Any node y on the simple path from r to x is called an ancestor of x. In this case x is a descendant of y. When the last edge on the simple path from r to x is (y, x), then y is called the parent of x, and x is called a child of y. The root r of T is the only node with no parent. Siblings are the nodes of T with same parent. A $leafnode$ is the node with no children. Any non-leaf node is called internal node.

The depth of a node x in T is the length of the simple path from the root r to x. All nodes in the same depth in T is in the same level of T. The height of a node x in T is the length of the longest simple path from x to a leaf node. The largest depth of any node in the tree T denotes the height of T. Figure 2.7 illustrates a rooted tree with height 4. The root of the tree is node 7, and its children at depth 1(nodes $3, 10, 4$). Nodes $6, 9, 1, 10, 11, 2$ are leaf nodes.

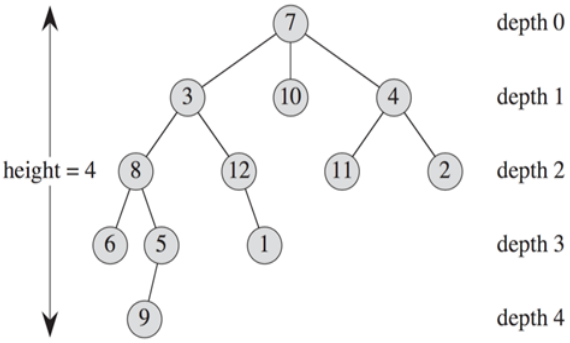


Figure 2.7: A rooted tree T

2.2 Datasets

In this section, we demonstrate our datasets which come from different domains such as biology, web graph, internet measurements, collaboration, and social network datasets. Table 2.1 summarizes basic statistics about our graph datasets in terms of number of vertices n, number of edges m, and diameter of graph $diam$.

Datasets	n	m	$diam$
Euroroad [4]	1,174	1,417	62
PPI [17]	1,458	1,948	19
Yeast [11]	2,224	6,609	11
Facebook [2]	2,888	2,981	9
DutchElite [15]	3,621	4,311	22
EVA [20]	4,475	4,664	18
AS-733 [12]	6,475	13,895	9
Erdös [6]]	6,927	11,850	4
Routeview [23]	10,515	21,455	10
PGP [5]	10,681	24,316	24
CAIDA [19]	26,475	53,381	17

Table 2.1: *Summarization of graph datasets used in this work: n, m is the number of vertices and edges respectively, $diam$ is the diameter of the graph*

2.2.1 Biological Networks

2.2.1.1 PPI

This biological graph demonstrates interactions between proteins in yeast Saccharomyces Cerevisiae. PPI is a protein-protein interaction dataset, where each vertex represents a protein, and each edge represents an interaction between two proteins. In the original dataset, self loops are deleted. PPI has been analyzed and described in depth in [17].

2.2.1.2 Yeast

This biological graph shows interactions between proteins in the budding yeast. Yeast is a protein-protein interaction dataset, where vertices represent proteins, each edge represents an interaction between two proteins. In the original dataset, self loops are eliminated. This dataset is well analyzed and described in [11].

2.2.2 Social and Collaboration Networks

2.2.2.1 DutchElite

This dataset was collected by De Volkskrant and Wouter de Nooy [15] from the administrative elite in Netherland, 2006. Memberships in the administrative and organization bodies were represented in this dataset. The structure of the dataset as follow: persons and organizations are represented by vertices. An edge is drawn between a person vertex and an organization vertex if the person belongs to the organization.

2.2.2.2 EVA

EVA dataset represents interconnections between corporations. Each vertex represents a company, and there is an edge between two vertices (companies) if one of them is the owner of the other one. This graph is well described in [20].

2.2.2.3 Erdös

Erdös is a collaboration network, where each vertex represents an author, and an edge between two vertices (authors) represents a paper co-authorship between them. Erdös is the collaboration network of the mathematician Paul Erdös [6].

2.2.2.4 PGP

List of edges of the giant component of the network of users of the Pretty-Good-Privacy algorithm for secure information interchange. This is the interaction network presented by M. Boguña [5].

2.2.2.5 Facebook

Facebook is a social network graph that is collected from a Facebook user's friend list. In this graph, each vertex represents a user, and each edge represents a friendship. Two users are friends if and only if there is an edge connected their two vertices in the graph. This graph collected by J. McAuley and J. Leskovec [2].

2.2.3 Internet Measurements Networks

2.2.3.1 Routeview

The Routeview graph dataset was obtained from Route-View Project by The University of Oregon [23]. Routeview is an Autonomous System (AS) graph using looking glass data and routing registry. The dataset represented as an undirected graph where each vertex represents an AS, and edges represent physical links between them.

2.2.3.2 CAIDA

CAIDA is an autonomous system (AS) graph dataset of internet relationships. Data is derived from BGP table snapshots taken over a 5 days' period [19]. The obtainable AS relationships are customer-provider, peer-to-peer, and sibling- to-sibling.

2.2.3.3 AS-733

This graph dataset is presented by J. Leskovec, J. Kleinberg and C. Faloutsos [12] as an organization of sub-graphs, to the graph of routers comprising the internet, that is called

Autonomous System (AS). Each AS exchanges traffic flows with some neighbors. This project is also collected from Route-View Project by The University of Oregon. Vertices are autonomous systems (AS), and edges represent communication. The network has loops.

2.2.4 Road Networks

2.2.4.1 2.2.4.1. Euroroad

This dataset is an international E-road network that is located in Europe. Euroroad dataset was collected by Lovro *Š*ubelj and Marko Bajec [4]. Vertices in this graph denote cites and an edge between two nodes represents that they are joined by an E-road.

Chapter 3

Algorithms

In this section we describe our method of computing top-k closeness centrality in a network. Our basic idea of computing the k most central vertices is the following: (1) compute a lower bound, $S'(v)$, of total distance for every vertex v in the graph, (2) run BFS for vertices with lowest lower bounds, $S'(.)$, to compute exact total distance of a vertex, (3) stop computation when the k vertices are found.

We start with describing the method of computing a lower bound for every vertex in the network (graph), layering partition lower bound (LPLB) algorithm, we also refer it as the precomputation step. After we build lower bounds, we use the top-k ranking algorithm by [7] for finding the k vertices with the highest closeness centrality. Furthermore, we propose an approximation algorithm to find the median vertex in the graph. We verify experimentally that our approximate median is no worse that 1.5× the total distance of the optimal median.

3.1 Computing Lower Bound

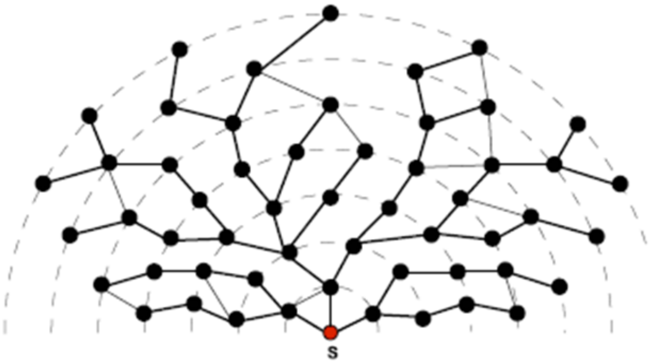
Our main idea of designing a tight lower bound, $S'(v)$, for every $v \in G$ is to decompose the graph G into a tree graph, $\Gamma = (Q, E')$. Each node $w \in Q$ represents a group of vertices of G. We compute farness, total distances, $S_\Gamma(w)$ for each node w. After that, each node w

will assign its farness value as a lower bound on total distances, $S'_G(v)$, to the vertices of G that the node w represents.

Our algorithm of computing lower bounds, layering partition lower bound (LPLB), we also call it the preprocessing, consists of four steps. First step is to decompose the graph metric into a tree. Every node in the tree corresponds to at least one vertex of the original graph. Second step puts nodes of the tree in an order based on the dismantling ordering. The third step computes total distance, farness on the tree, form every node to all other nodes of the tree. The fourth step assigns the total distance of every node on the tree to its corresponding vertices in the graph as a lower bound.

3.1.1 Step 1: Layering Partition

Our first step of designing a tight lower bound, $S'(v)$, for each vertex $v \in V$ comes from [10] and [13]. Layering partition is a procedure that embeds graph's metric into a tree, we denote it as Γ. Γ is very operational in calculating a tight lower bound on distances between vertices, that is the main tool in our strategy. Building $\Gamma = (Q, E')$ tree from a graph $G = (V, E)$ starts by layering G with respect to a starting vertex s. A layering of G starting from any vertex s is the decomposition of V into $r + 1$ layers $L^i = \{u \in V : d_G(s, u) = i\}$, $i = 0, 1, ..., r$. Layering of any graph can be done in linear time $O(n + m)$ by running BFS algorithm starting from s. Figure 3.1 shows the graph G after layering process.



.

Figure 3.1: Layering on Graph G with a starting node s

Partitioning starts with partition each layer L^i into clusters $L^i_1, \ldots, L^i_{(p^i)}$. The method of building these clusters is that any two vertices $u, v \in L^i$ belong to the same cluster L^i_j if and only if they can be connected by a path outside the ball $B_{(i-1)}(s)$ of radius $i - 1$ centered at s. Formally, $LP(G, s) = L^i_1, \ldots, L^i_{(p^i)} : i = 0, 1, \ldots, r$. Figure 3.2 shows partition process. The implementation of the portioning procedure in linear time comes from V. Chepoi and F. Dragan [13]. Just after layering G, we start from the layer $L_r(s)$ of largest radius. We start by finding connected components of the layer r based on the predefined method and contruct each of them into a node. After that we go to the lower layer and find the connected components in the graph induced by $L_{(r-1)}(s)$ and the set of contracted vertices, contract each of them and descend to the lower level, until we will come to the vertex s.

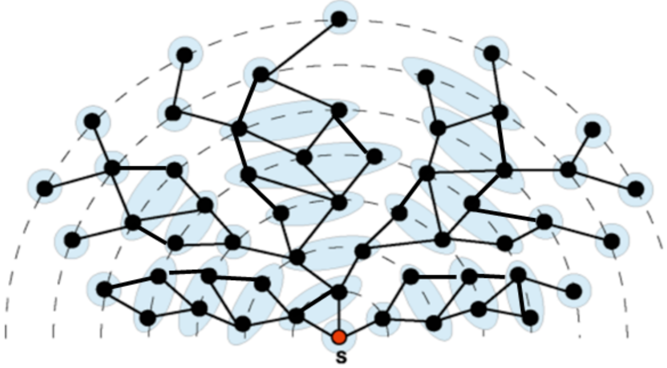


Figure 3.2: Clusters of the layering partition $LP(G, s)$.

$\Gamma = (Q, E')$ is the graph that results from layering partition process, $LP(G, s)$.The graph's Γ nodes Q are the clusters of $LP(G, s)$ and two nodes in Γ are adjacent if and only if there exist vertices $u \in L_j^i$, $v \in L_{j'}^{i'}$ such that there exists an edge $(u, v) \in E$.The resulting graph Γ is proven to be a tree by [13]. Figure 3.3 shows Γ tree.

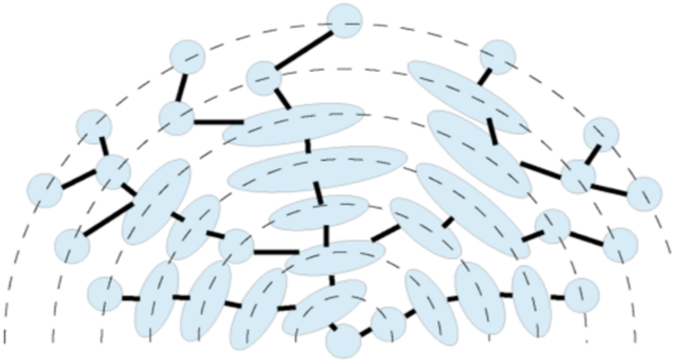


Figure 3.3: Γ tree that is resulted from $LP(G, s)$.

3.1.2 Step 2: Dismantling Ordering on Trees

Our main idea of computing a tight lower bound for each vertex in a graph G is to build a $\Gamma = (Q, E')$ graph from $G = (V, E)$. Next, we compute exact value of total distances, farness, for every node of Γ. Since we know every node in Γ is a cluster that represents group of vertices of G, we assign the total distances value of each node $w \in \Gamma$ as a lower bound to the vertices of G that the node w represents.

Let $S_\Gamma(w)$ represents the total distances, farness, of any node $w \in Q$ then

$$S_\Gamma(w) = \sum_{v \in Q} d_\Gamma(w, v) \times W(v), \tag{3.1}$$

where $d_\Gamma(w, v)$ is the number of edges on the shortest path between w, v in Γ, and $W(v)$ is the number of vertices of G residing in the cluster L_j^i that is represented by the node v.

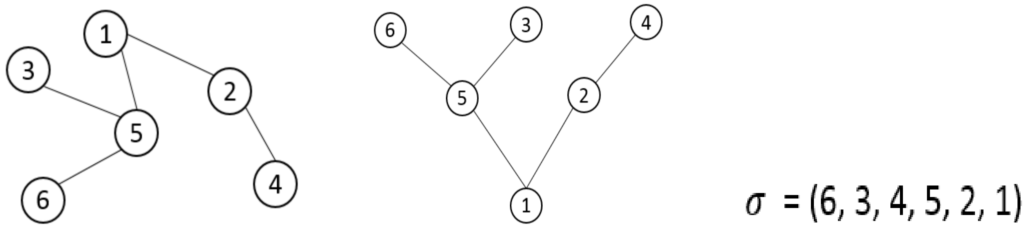
For a fast computation of the distance between any two nodes in Γ tree, a two stages procedure will be followed:

(1) find the dismantling ordering, σ, of Γ;

(2) based on σ we build a $|Q| \times |Q|$ array to compute exact distances.

In this subsection we present the solution to (1). The solution to (2) will be shown in subsection 3.1.3

The dismantling ordering of a tree can be built by finding leaf nodes of the tree at each iteration. The linear time implementation can be done by: (1) run BFS algorithm from a random node; (2) iteratively, cut leaf nodes and put them in the order. Figure 3.4(a) shows an example of a Γ tree, Figure 3.4(b) demonstrates Γ tree after running BFS from a starting node, *node*1. Building $\sigma(\Gamma)$ is done iteratively by cut nodes in the higher layer of the tree shown in Figure 3.4(b). After that, we add to the order, σ, nodes in the lower layer, and so on till we reach the starting node, Figure 3.4(c) shows the resulting order of Γ tree.



((a)) A Γ tree

((b)) Γ tree after running BFS from *node*1

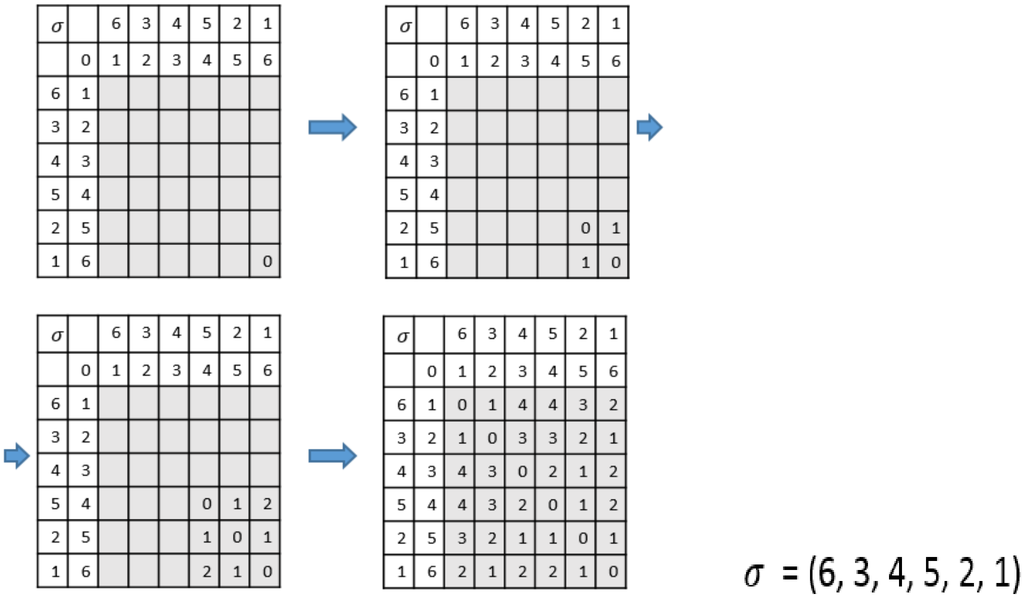
((c)) The dismantling ordering $\sigma(\Gamma)$

Figure 3.4: Building The dismantling ordering of Γ tree.

3.1.3 Step 3: Distance Matrix

Computing distance between any pair of nodes can be done using a two dimensional array, we denote it as A, of size $p \times p$, where $p = |Q|$. Columns and rows of A are occupied by the dismantling ordering, σ. The idea of computing exact distance between any pair of nodes is that: each node will rely on its parent to compute distances to other nodes. Since the distance between every node and its parent is 1, for every node that is reachable from the parent of a node w with distance d, it is also reachable from w with distance $d + 1$. One thing that we need to make sure of is that we should compute distances of a parent before its children. This property is preserved since we compute distances based on the dismantling ordering, σ.

The calculation will start from the bottom right corner and go up diagonally. To make the computation clear, we will rely on the following example: Example 1: Assume we have the perfect elimination ordering σ of Γ tree shown in Figure 3.5(b). Also, every node in σ has a pointer to its parent in Γ tree. We start the computation of exact distances on Γ tree by building matrix A shown in Figure 3.5(a). We start filling the array as follows: we start from the bottom right corner and fill $A(6,6) = 0$ because the distance from a node to itself is 0. We go diagonally up and fill $A(5,5) = 0$. We rely on $node1$ to calculate distances of $node2$, because $node1$ is the parent of $node2$, as follows: we will copy all distances that $node1$ knows so far and add 1 to each one of them and give them to $node2$. We will get $A(5,6) = A(6,6) + 1$, $A(6,5) = A(5,6)$. We continue to calculate distances of $node5$ as follows: $node5$ knows its parents, $node2$, so its distances will be the same as for its parent, $node2$, plus 1 to each of them. The distances of $node5$ will be as follows: $A(4,4) = 0$, $A(4,5) = A(5,5) + 1$, $A(4,6) = A(5,6) + 1$, $A(5,4) = A(4,5)$, $A(4,6) = A(4,6)$. And so on for all the remaining nodes.



σ		6	3	4	5	2	1
	0	1	2	3	4	5	6
6	1						
3	2						
4	3						
5	4						
2	5						
1	6						0

σ		6	3	4	5	2	1
	0	1	2	3	4	5	6
6	1						
3	2						
4	3						
5	4						
2	5					0	1
1	6					1	0

σ		6	3	4	5	2	1
	0	1	2	3	4	5	6
6	1						
3	2						
4	3						
5	4				0	1	2
2	5				1	0	1
1	6				2	1	0

σ		6	3	4	5	2	1
	0	1	2	3	4	5	6
6	1	0	1	4	4	3	2
3	2	1	0	3	3	2	1
4	3	4	3	0	2	1	2
5	4	4	3	2	0	1	2
2	5	3	2	1	1	0	1
1	6	2	1	2	2	1	0

$\sigma = (6, 3, 4, 5, 2, 1)$

((a)) Steps of filling the distances matrix, A

((b)) The dismantling ordering $\sigma(\Gamma)$

Figure 3.5: Computing exact distances of Γ tree.

After building the two dimentional array A, every node in Γ tree will have information about the shortest path to every other node. In order to compute value of total distances, farness, $S_\Gamma(w)$ for every node $w \in \Gamma$, we apply Equation (3.1). From the implementation point of view this step can be done in $O(p^2)$ time, where $p = |Q|$. The running time is linear to the graph G, since $|Q| \leq |V|$, however, in practice Q is way smaller than V.

3.1.4 Step 4: Assignment of Lower Bounds

By the end of Subsection 3.1.3, we have calculated the total value of distances $S_\Gamma(L_j^i)$ for every node L_j^i in Γ tree. Just a reminder that the graph $\Gamma = (Q, E')$ was built from $G = (V, E)$ through clustering vertices of G into groups based on a specific method, defined in Subsection 3.1.1. Every node $L_j^i \in \Gamma$ is a cluster that holds a group of vertices from G in a list we denote it as $l(L_j^i)$. In this subsection, every node L_j^i in Γ tree assigns its value of total distances, $S_\Gamma(L_j^i)$, as a lower bound for all vertices v in its cluster list as $S'(v)$. Formally: $\forall v \in l(L_j^i)$, $S_G^{'}(v) = S_\Gamma(L_j^i)$, where $S_G^{'}(v)$ is the lower bound on total distances for the vertex $v \in V$, and $S_\Gamma(L_j^i)$ is the exact total distances of the node $L_j^i \in Q$.

A nice observation has been made by Bergamini et al. [7] that can more tighten our lower bound. They observe that in every graph the number of vertices at distance 1 from v is exactly the degree of v. As well as, all other vertices u are at least at 2 distance away from v, except v itself. This observation can tighten our lower bound $S'(v)$. Formally:

$$S'_G(v) := S'_G(v) + \sum_{\forall u \in adj_\Gamma(v)} W(u) + 2 \times W(v) - deg(v) - 2. \tag{3.2}$$

3.2 Computation of Top-*k* Closeness Centrality and Median

Both the computation of the exact top-k closeness centrality and the approximate median rely on the same preprocessing step, which is the computation of the lower bound, LPLB.

3.2.1 Computation of Top-*k* Closeness Centrality

In previous sections we gradually built a very tight lower bound $S'(v)$ for every vertex $v \in V$. In this section, we are going to describe the approach that we use to compute the top-k vertices with maximum closeness centrality. The basic idea of this algorithm is described in [7], however we visit it again.

Let $S(v)$ be the total distances of a vertex v and let $S'(v)$ be its lower bound. If $S(v) \leq S'(u)$, $\forall u \in V$, it is true that $S(v) \leq S(u), \forall u \in V$, thus v is one of the nodes, might be the one, with maximum closeness centrality. Based on this observation, we are not required to compute the total distances $S(u)$ for all remaining vertices. Algorithm 1 is the pseudocode of the idea above.

First, we insert all vertices into an ascending ordered priority queue, ordered by the value of $S'(v)$. At the same time, we set a truth value $exact(v)$ to false. Then, we pop vertices from the queue one after another. For the popped vertex v^*, we compute the exact total distance $S(v)$ and set $exact(v)$ to true and re-enqueue v^* again based on its new priority $S(v)$. If we encounter a vertex with $exact(v) = true$ that means its exact value has been computed and the total distances of it is smaller than the lower bound on total distances of all other vertices in the queue. Therefore, we can append v^* to the list of the most central vertices.

Algorithm 1 Top-k ranking

Input: A graph G=(V, E)
Output: top-k vertices with highest closeness and their closeness values $c(u)$

```
$S' \leftarrow ComputeLowerBounds$
$Q1 \leftarrow \emptyset$
for each $v \in V$ do
    $Q1 \leftarrow enqueue(v, p_v = S'(v))$
    $exact[v] \leftarrow false$
end for
$i \leftarrow 0$
$TopK \leftarrow [\,]$
while $i < k$ do
    $(v^*, S^*) \leftarrow extractMin(Q1)$
    if $exact[v^*]$ then
        $TopK \leftarrow (v^*, S^*)$
        $i \leftarrow i + 1$
    else
        $S(v) \leftarrow \sum_{w \in V} d(v, w)$
        $Q1 \leftarrow enqueue(v, p_v = S(v))$
        $exact[v] \leftarrow enqueue(v, p_v = S(v))$
    end if
end while
return $TopK$
```

3.2.2 The Approximation Algorithm for Computing Median

In this subsection, we describe our approximation algorithm (Approximation Cut, for short *AproxCut*) for computing a median vertex of a network. In many real-world networks the problem of finding a median vertex is critical and challenging. To the best of our knowledge, even for a graph with few millions of nodes, there is no feasible algorithm to compute the exact median on large networks. Since the problem of finding the exact median (top-k closeness centrality, when k = 1) is proven to be equivalent to APSP problem, the attention has been devoted toward the approximation algorithms. The approximation algorithms can give solution that can be correct or close to the correct answer. In this section we propose a new approximation algorithm for finding a median in a graph. Our approximation algorithm is based on a near linear preprocessing (computing lower bounds) and after the preprocessing it examines only 5 % of the graph's vertices to give an approximate median that is no worse than 1.5× the total distances of the optimal median.

AproxCut starts with computing a lower bound for each vertex $v \in V$ using LPLB procedure. The next step is to insert all vertices V into a min-priority queue, $Q1$ based on the lower bound $S'(v)$. After that we extract only 5% of the vertices in $Q1$, compute their exact values of total distances, and insert them into a new priority queue $Q2$ based on their $S(v)$. The approximate median will be in the peek of $Q2$. Algorithm 2 is the pseudocode of the *AproxCut*.

The reason behind the proposal of this algorithm is the following. The lower bound algorithm can give a good prediction of the top-k closeness centrality vertices. We have noticed that LPLB puts the top-k vertices early in the queue, however, to validate them we need to spend some time. The proposed approximation algorithm (*AproxCut*) does not validate the median, however, we show experimentally that the vertex that *AproxCut* provides is a median in many cases, and, if not a median, its value is very close to the optimal. In particular, we verify experimentally, in Chapter 4, that the approximate median is no more than 1.5× the total distances of the optimal median.

Algorithm 2 *ApproxCut*

```
Input: A graph G=(V, E)
Output: Median vertex of $G$ with its closeness value $c(v)$
$S' \leftarrow ComputeLowerBounds$
$Q1 \leftarrow \emptyset$
$Q2 \leftarrow \emptyset$
for each $v \in V$ do
    $Q1 \leftarrow enqueue(v, p_v = S'(v))$
end for
$i \leftarrow 0$
$k \leftarrow G.size * 0.05$
while $i < k$ do
    $(v^*, S^*) \leftarrow extractMin(Q1)$
    $S(v) \leftarrow \sum_{w \in V} d(v, w)$
    $Q2 \leftarrow enqueue(v, p_v = S(v))$
    $i \leftarrow i + 1$
end while
$median \leftarrow extractMin(Q2)$
return $median$
```

Chapter 4

Experimental Evaluation

We denote our algorithm of finding the exact top-k closeness centrality by $BestClos$ and the-state-of-the-art algorithm by $FastClos_L$ [7]. Furthermore, we denote our approximation algorithm for finding median by $AproxCut$. In this section, we experimentally evaluate the performance of the algorithms $BestClos$ and $AproxCut$. $BestClos$ is evaluated against Bergamini et al. algorithm, $FastClos_L$, that is the-state-of-the-art algorithm for finding top-k closeness centrality in unweighted undirected graphs. All three algorithms are implemented in Java 1.8., the machine used is MacBook Pro, 2.6 GHz Intel Core i5 processor, and 8 GB 1600 MHz DDR3 memory.

4.1 Datasets

We use eleven real-world datasets, which come from different domains such as biology, web graph, internet measurements, social and collaboration networks, to evaluate $BestClos$ and $AproxCut$. The networks are taken from SNAP (snap.stanford.edu), KONECT (konect.uni-koblenz.de) and other resources stated in Section 2.2. Table 2.1 summarizes the properties of datasets. Section 2.2 gives a clear explanation to each dataset.

4.2 Preprocessing

Preprocessing, performed by Bergamini et al. algorithm [7] Level-Based Lower Bound (LBLB) and our algorithm Layering Partition Lower Bound (LPLB), is used to compute a lower bound for each vertex $v \in V$. LBLB computes lower bounds by first running BFS from a starting vertex s. BFS will layer the graph into levels based on the distance from the starting vertex s. Let i and j be two levels, and $i \leq j$. Then, the distance between any two vertices v at level i and w at level j must be at least $j - i$. Actually, if $d(v, w)$ was smaller than $j - i$, w would be at level $i + d(v, w) < j$, which contradicts the assumption. LBLB produces lower bounds that are not very close to the exact total distances values of vertices to help Algorithm 1 to terminate early.

On the other hand, our algorithm of computing lower bounds LPLB produces lower bounds to vertices that are very close to the exact total distances. LPLB strategy is to further partition the graph not only to layers, but also to clusters in these layers. The main weakness in LBLB is that they treat all vertices in the same layer similarly. However, LPLB further divides the graph into layers and also clusters in every layer. Dividing the graph helps us to compute better lower bounds. Figures 4.1, 4.2, and 4.3 show direct comparison of LPLB bounds, LBLB bounds, and the exact total distances of every vertex.

In Figures 4.1, 4.2, and 4.3, the performance is shown for every vertex in sorted order based on their exact total distances. Every vertex in charts represented on Figures 4.1, 4.2, and 4.3 contributed three values as follows: (1) its exact total distance, $S(v)$, (2) its lower bound on total distances computed by LPLB, $S'_{LPLB}(v)$, we denote it as LPLB, and (3) its lower bound on total distances computed by LBLB, $S'_{LBLB}(v)$, we denote is as LBLB. We can clearly see that LPLB lower bounds are most of the time better than lower bounds of vertices produced by LBLB. Both LPLB and LBLB produce same lower bound values for some vertices, however, LBLB does not produce lower bounds that are better than that LPLB produces. So, LPLB most of the time produces lower bounds on total distances for vertices that are better than the ones produces by LBLB. It is worth to mention that even though in some cases both LPLB and LBLB produces similar lower bounds for some vertices, $FastClos_L$ fails to confirm the top-k vertices fast enough. $FastClos_L$ performance is mostly behind our algorithm's, $BestClos$, performance. The performance comparison in the next sections clearly demonstrates that.

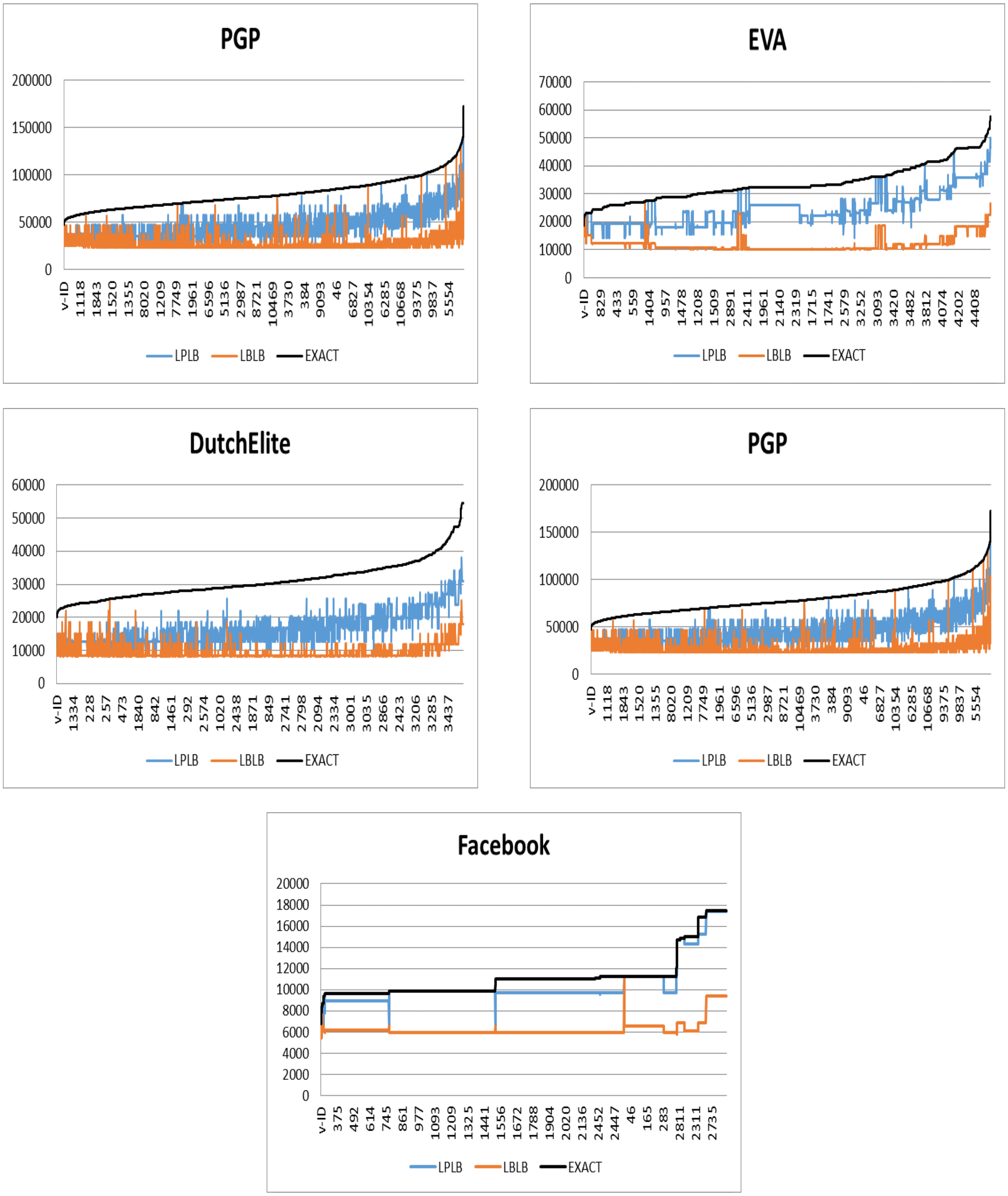


Figure 4.1: Comparison between lower bounds produced by our algorithm LPLB and lower bounds produced by LBLB to the optimal total distances, EXACT. The $x-axis$ is the vertex ID and $y-axis$ represents the three values for every vertex of Social and Collaboration Networks.

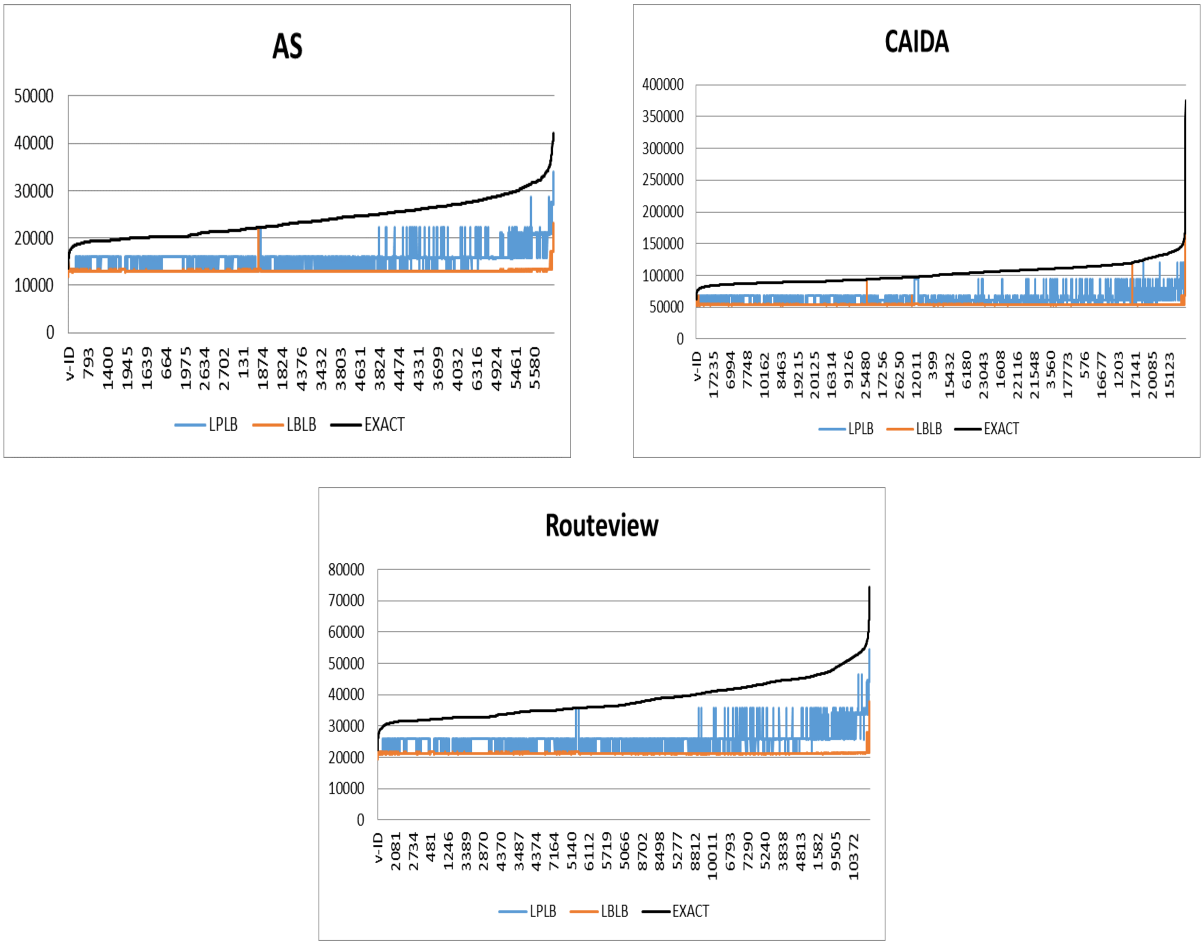
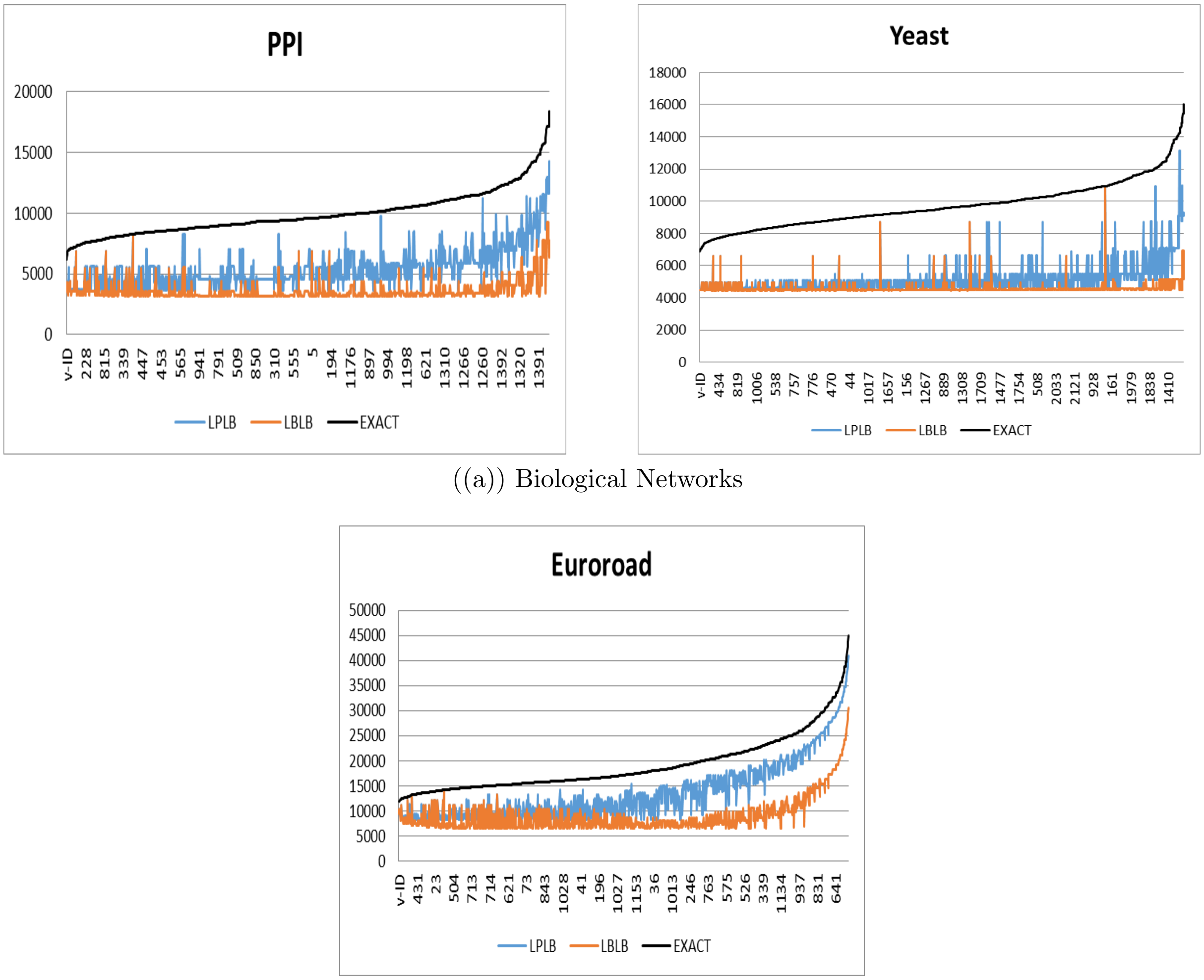


Figure 4.2: Comparison between lower bounds produced by our algorithm LPLB and lower bounds produced by LBLB to the optimal total distances, EXACT. The $x-axis$ is the vertex ID and $y-axis$ represents the three values for every vertex of Internet Measurements Networks.



((a)) Biological Networks

((b)) Road Networks

Figure 4.3: Comparison between lower bounds produced by our algorithm LPLB and lower bounds produced by LBLB to the optimal total distances, EXACT. The $x - axis$ is the vertex ID and $y - axis$ represents the three values for every vertex of Biological Networks and Road Networks.

4.3 Performance Measure

In this test we adopt the measure of performance the *performance ratio* that is introduced by Borassi et al. [9]. Their performance ratio is based on edge traversed, however, we adopt it for vertices. Let $|V_{vis}|$ be the number of vertices visited by the top-k ranking algorithm (Algorithm 1) at line 15. The performance ratio is defined as $|V_{vis}|$ divided by the number of vertices that the exhaustive algorithm (running BFS for all nodes) would use, i.e., $\frac{|V_{vis}|}{|V|}$. This measure depends on the number of operations performed. Furthermore, this measure allows an assessment autonomous of computer architecture. This test is independent in terms that it does not consider the preprocessing step. Table 4.1 shows the performance ratio of our algorithm *BestClos* and Bergamini et al. algorithm $FastClos_L$ [7] with $k = 1, 5$, and 10. From Table 4.1 we can clearly see that our algorithm for small k, $k = 1$ to 10, outperforms the-state-of-the-art algorithm $FastClos_L$.

Dataset	Number of Vertices	*performance ratio*(k=1) *BestClos*	*performance ratio*(k=1) $FastClos_L$	*performance ratio*(k=5) *BestClos*	*performance ratio*(k=5) $FastClos_L$	*performance ratio*(k=10) *BestClos*	*performance ratio*(k=10) $FastClos_L$	
Euroroad	1175	45%	76%	48%	80%	50%	80%	
PPI	1458	73%	97%	85%	98%	86%	98%	
Yeast	2224	95%	99%	95%	99%	97%	99%	
Facebook	2889	0.2%	91%	1%	94%	16%	99%	
DutchElite	3621	83%	99%	85%	99%	85%	99%	
EVA	4475	14%	98%	34%	99%	39%	99%	
AS-733	6475	24%	99%	25%	99%	91%	99%	
Erdös	6927	0.01%	0.01%	98%	99%	98%	99%	
Routeview	10515	24%	99%	24%	99%	26%	99%	
PGP	10681	64%	98%	71%	99%	71%	99%	
CAIDA	26476	71%	99%	71%	99%	71%	99%	

Table 4.1: *Performace evaluation of BestClos against the-state-of-the-art algorithm,* $FastClos_L$*, with k= 1, 5, and 10. Performance ratio is the ratio of the visited vertices by BestClos and* $FastClos_L$ *to the number of vertices visited by the naïve algorithm of finding top-k most central vertices in a network.*

4.4 Full Performance Measure

In this experiment, we monitor the performance of *BestClos* and $FastClos_L$ with all possible values of k. In Subsection 4.3 we restrict k to specific values, but in this experiment we will see the algorithms' performance for all possible values of k. The results will be shown in a figure for each graph dataset where $x - axis$ represents number of vertices visited by the top-k ranking algorithm (Algorithm 1) and $y - axis$ specifies the specific value k. As in previous experiments, we examine the same collection of various datasets that comes from different domains, described in Section 2.2. The reason behind considering graphs from different domains is to draw a conclusion about the performance of our algorithm ($BestClos$) in order to recommend it to be used in specific fields. We strongly believe that our algorithm, $BestClos$, same as most of the proposed algorithms to solve the closeness centrality, performs differently based on characteristics of the network, it is interesting to show that experimentally. Figures 4.4, 4.5, and 4.6 show the performance of $BestClos$ and $FastClos_L$ on different collections of graph datasets.

Figures Figures 4.4, 4.5, and 4.6 show that $BestClos$ has significant improvements over the-state-of-the-art $FastClos_L$. Furthermore, we notice that the performance may differ based on the network. For some social networks, such as Facebook [2] $BestClos$ has superior performance, see Figure 4.4, however, on biological network, such as Yeast [11] it has a poor performance, see Figure 4.6(a). Biological networks are known to have a high degree of vertices and low diameter. These characteristics may affect the method of computing lower bounds. We notice from Figure 4.3(a) that the lower bound of biological network is not very close to the exact total distances values. We do not recommend our method for biological networks. However, we highly recommend it to be used for other networks such as, social and road networks.

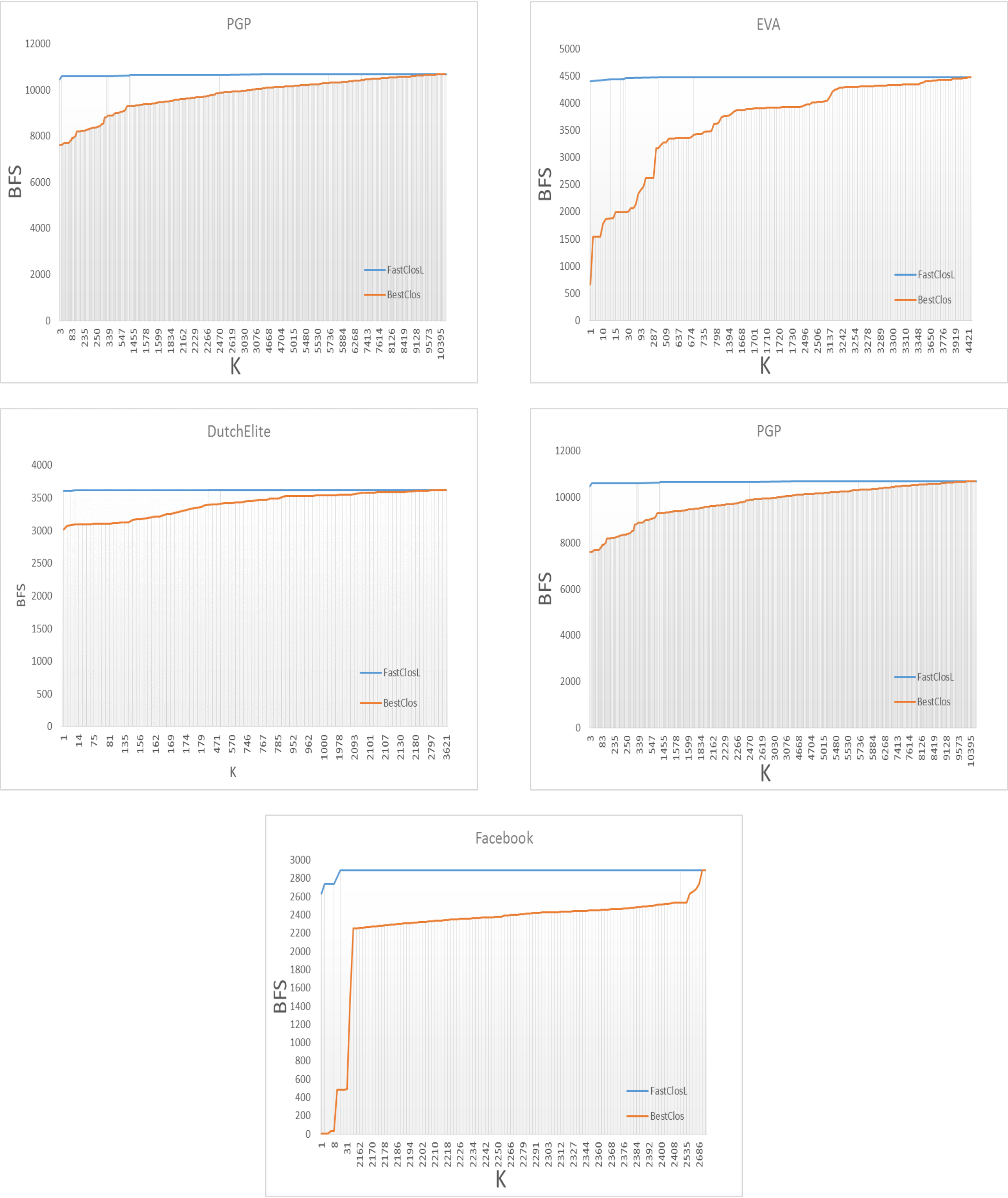


Figure 4.4: Full performance test of $BestClos$ against the-state-of-the-art $FastClos_L$. The $x - axis$ shows the value of k, and the $y - axis$ represents number of BFSs needed by the two algorithms to produce the top-k vertices with best closeness centralities for Social and Collaboration Networks.

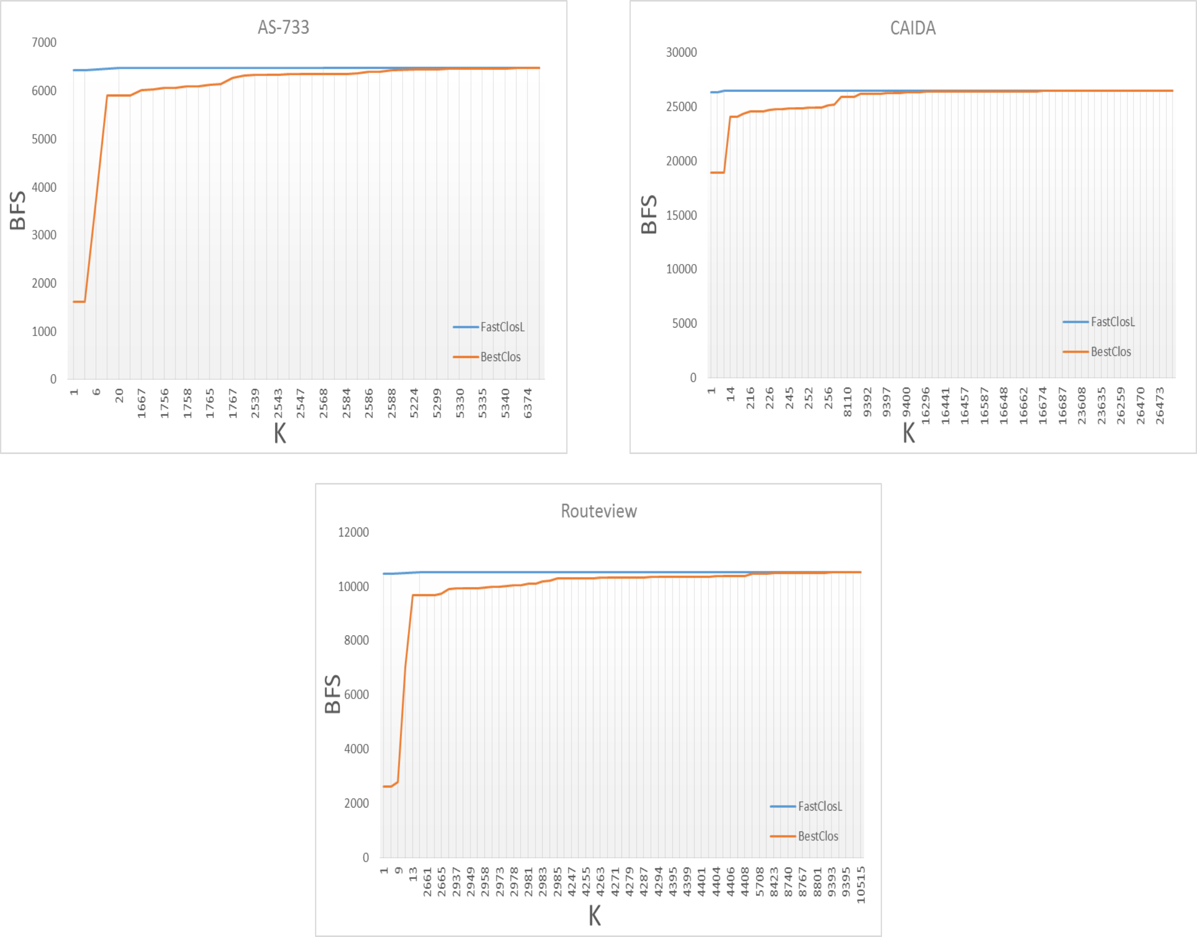
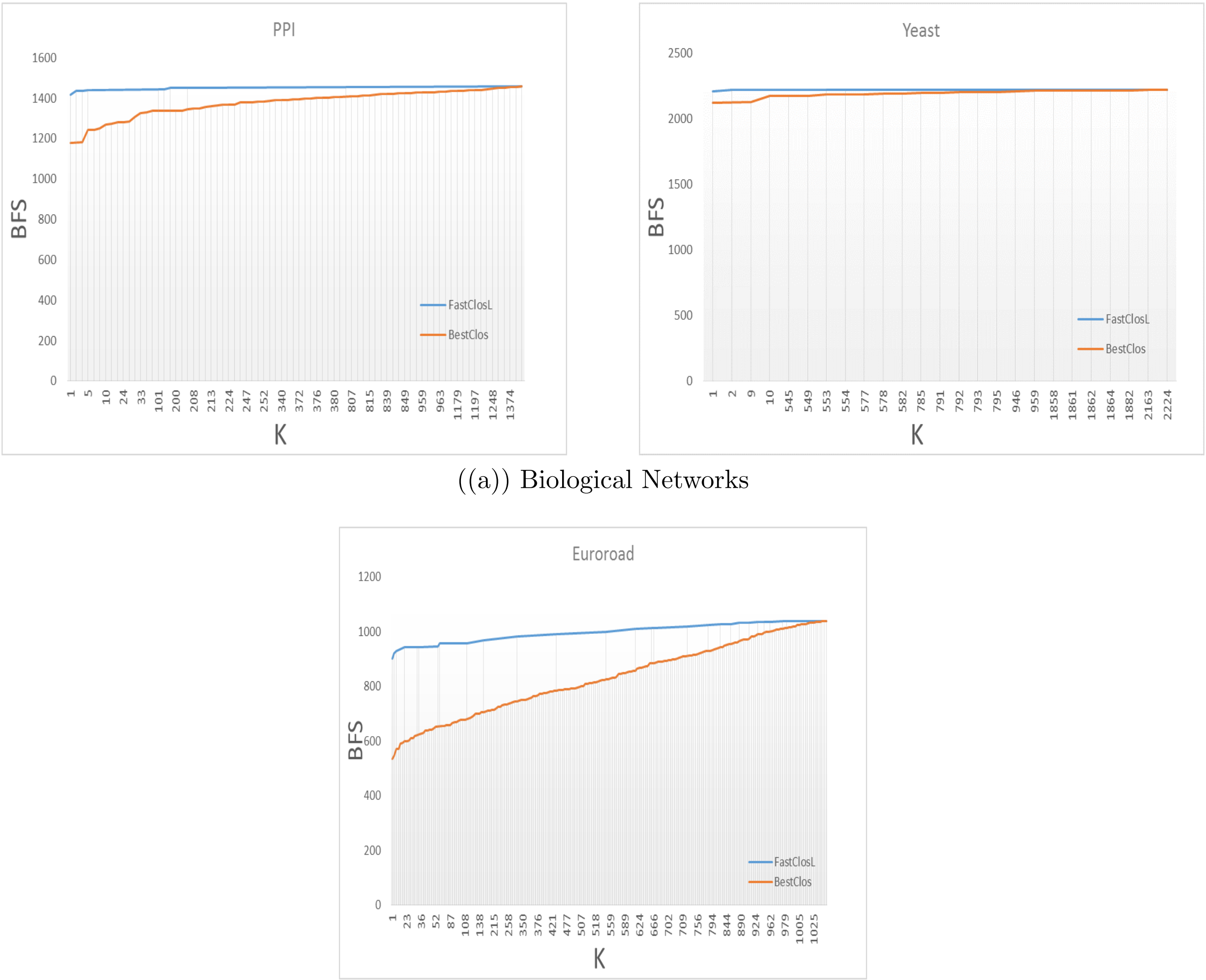


Figure 4.5: Full performance test of $BestClos$ against the-state-of-the-art $FastClos_L$. The $x - axis$ shows the value of k, and the $y - axis$ represents number of BFSs needed by the two algorithms to produce the top-k vertices with best closeness centralities for Internet Measurements Networks.



((a)) Biological Networks

((b)) Road Networks

Figure 4.6: Full performance test of $BestClos$ against the-state-of-the-art $FastClos_L$. The $x-axis$ shows the value of k, and the $y-axis$ represents number of BFSs needed by the two algorithms to produce the top-k vertices with best closeness centralities for Biological Networks and Road Networks.

4.5 Median Test

In this experiment, we test how good is our approximate median that results from the proposed approximation algorithm $AproxCut$ (Algorithm 2). We do the test by comparing results of the proposed algorithm of finding approximate median, $AproxCut$, with the optimal median for various datasets. In Subsection 3.2.2, we mentioned that in order to find the approximate median we do preprocessing, LPLB, and after that, we examine only 5% of the dataset to give the approximate median with confidence. In this test, we show that examining only 5% of the graph after the preprocessing step is enough to give a result that is very close to the optimal solution, if it does not give the optimal. We show that by comparing the result of examining 5% of the graph, *AproxCut*, with the optimal, exact, median. The metric of this experiment is the ratio between total distances, $S(.)$, of the approximate median found by *AproxCut* and total distances of the optimal median of the dataset, $\frac{S(v)}{S(u)}$, where $S(v)$ is the value of total distances of the approximate median v, and $S(u)$ is the value of total distances of the optimal median u. If the ratio is one, we have the optimal median. The higher the ratio is the far *AproxCut* solution is from the optimal median.

To make sure that *AproxCut* algorithm gives a high confidence median by examining only 5% of the dataset, in this experiment we go even beyond examining only 5% to 3% and even 1% to see how good the approximate median that *AproxCut* provides. We test *AproxCut* on eleven datasets from various domains. Our datasets are described in depth in Section 2.2 . Table 4.2 shows that ratio between the total distances of the approximate median provided by *AproxCut* and the total distances of the optimal median.

The result of this experiment shows that for the eleven datasets from various domains that we have examined, *AproxCut* algorithm gives the optimal median in many cases. If *AproxCut* does not give the optimal median, it gives a vertex that is very close to the optimal. We can say with confidence that the approximate median provided by *AproxCut* is no worse than 1.5× the total distances of the optimal median. It is clear that *AproxCut* can give a very good answers not only by examining 5% of the graph, but also 3% and even 1%. To sum up, experiments that we have used in this subsection validate *AproxCut* proposed in Subsection 3.2.2.

Dataset	Farness of *AproxCut* median / Farness of optimal median, only 5% of vertices examined	Farness of *AproxCut* median/ Farness of optimal median only 3% of vertices examined	Farness of *AproxCut* median / Farness of optimal median only 1% of vertices examined
Euroroad	1.05	1.05	1.05
PPI	1.12	1.12	1.14
Yeast	1	1	1
Facebook	1	1	1
DutchElite	1.02	1.02	1.02
EVA	1.06	1.06	1.23
AS-733	1	1	1
Erdös	1	1	1
Routeview	1	1	1
PGP	1.08	1.08	1.08
CAIDA	1	1	1

Table 4.2: *Total distances, farness, of the median proposed by the approximation algorithm (AproxCut) to the total distances, farness, of optimal median.*

Chapter 5

Conclusion

In this work we tackled an important problem in network analysis, closeness centrality. We proposed a practical solution to the problem of finding the top-k most central vertices in large networks. The most recent work on finding most central vertices are based on computing a lower bound on total distances for vertices and stop computation when the k most central vertices are found. However, their method of computing lower bounds is not very effective. We present a new method of finding lower bounds for each vertex in the network. We show that our method significantly improves the-state-of-the-art method of computing lower bounds [7]. As a result we can efficiently find the top-k closeness centrality faster than all the known methods.

Our method of finding lower bounds helps to find approximate median of a network very fast. Median of a network is a vertex that is the closest, on average, to all other vertices in the network. We present an efficient algorithm for finding an approximate median that is no worse than 1.5× total distances of optimal median.

To verify our algorithms, we tested them on eleven datasets from various domains such as biology, social networks, collaboration networks, and road networks. We have drawn into a conclusion that our method outperforms the state-of-the-art methods on all domains of networks that we have tested. Our method gives its best performance on social and road networks where the diameter is high.

For future work, we may examine different starting points because we highly believe that choosing a good vertex to start with the layering partition step may give even better results. We plan to propose an upper bound on total distances to prune many vertices. Furthermore, we plan to adopt our method of computing lower bounds to weighted directed graphs. Also, we want to use our method to suggest a top-k approximation algorithm of computing closeness centrality. Furthermore, it would be interesting to adapt our method of computing lower bounds for another centrality measure such as betweenness centrality.

Bibliography

[1] Amir Abboud, Fabrizio Grandoni, and Virginia Vassilevska Williams. Subcubic equivalences between graph centrality problems, apsp and diameter. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1681–1697. Society for Industrial and Applied Mathematics, 2015.

[2] Richa Agarwala, Vineet Bafna, Martin Farach, Mike Paterson, and Mikkel Thorup. On the approximability of numerical taxonomy (fitting distances by tree metrics). *SIAM Journal on Computing*, 28(3):1073–1085, 1998.

[3] Muslem Muhamed Mahdi Al-Saidi. Balanced disk separators and hierarchical tree decomposition of real-life networks. Master's thesis, Kent State University, 2015.

[4] Mihai Bădoiu, Erik D Demaine, MohammadTaghi Hajiaghayi, Anastasios Sidiropoulos, and Morteza Zadimoghaddam. Ordinal embedding: approximation algorithms and dimensionality reduction. In *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pages 21–34. Springer, 2008.

[5] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[6] Vladimir Batagelj and Andrej Mrvar. Some analyses of Erdös collaboration graph. *Social networks*, 22(2):173–186, 2000.

[7] Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke. Computing top-k closeness centrality faster in unweighted graphs. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 68–80. SIAM, 2016.

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

[9] Michele Borassi, Pierluigi Crescenzi, and Andrea Marino. Fast and simple computation of top-k closeness centralities. *arXiv preprint arXiv:1507.01490*, 2015.

[10] Andreas Brandstädt, Victor Chepoi, and Feodor Dragan. Distance approximating trees for chordal and dually chordal graphs. *Journal of Algorithms*, 30(1):166–184, 1999.

[11] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, et al. Topological structure analysis of the protein–protein interaction network in budding yeast. *Nucleic acids research*, 31(9):2443–2450, 2003.

[12] Mihai Bdoiu, Piotr Indyk, and Anastasios Sidiropoulos. Approximation algorithms for embedding general metrics into trees. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 512–521. Society for Industrial and Applied Mathematics, 2007.

[13] Victor Chepoi and Feodor Dragan. A note on distance approximating trees in graphs. *European Journal of Combinatorics*, 21(6):761–766, 2000.

[14] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[15] W de Nooy. The network data on the administrative elite in the netherlands in april-june 2006.

[16] David Eppstein and Joseph Wang. Fast approximation of centrality. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 228–229. Society for Industrial and Applied Mathematics, 2001.

[17] Hawoong Jeong, Sean P Mason, A-L Barabási, and Zoltan N Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, 2001.

[18] Christine Kiss and Martin Bichler. Identification of influencersmeasuring influence in customer networks. *Decision Support Systems*, 46(1):233–253, 2008.

[19] Olawale Abiodun Martins. *Affecting IP traceback with recent Internet topology maps.* PhD thesis, Iowa State University, 2005.

[20] Kim Norlen, Gabriel Lucas, Mike Gebbie, and John Chuang. Eva: Extraction, visualization and analysis of the telecommunications and media ownership network. In *Proceedings of International Telecommunications Society 14th Biennial Conference (ITS2002), Seoul Korea*. Citeseer, 2002.

[21] Paul W Olsen, Alan G Labouseur, and Jeong-Hyon Hwang. Efficient top-k closeness centrality search. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 196–207. IEEE, 2014.

[22] Sergio Porta, Emanuele Strano, Valentino Iacoviello, Roberto Messora, Vito Latora, Alessio Cardillo, Fahui Wang, and Salvatore Scellato. Street centrality and densities of retail and services in Bologna, Italy. *Environment and Planning B: Planning and design*, 36(3):450–465, 2009.

[23] Steve Uhlig, Bruno Quoitin, Jean Lepropre, and Simon Balon. Providing public intradomain traffic matrices to the research community. *ACM SIGCOMM Computer Communication Review*, 36(1):83–86, 2006.

[24] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 887–898. ACM, 2012.