Exploring Hierarchies in Online Social Networks

Can Lu, Jeffrey Xu Yu, Rong-Hua Li*, Hao Wei

The Chinese University of Hong Kong, Hong Kong, China
**Guangdong Province Key Laboratory of Popular High Performance Computers, Shenzhen University, China*

{lucan,yu,rhli,hwei}@se.cuhk.edu.hk

ABSTRACT

Social hierarchy (i.e., pyramid structure of societies) is a fundamental concept in sociology and social network analysis. The importance of social hierarchy in a social network is that the topological structure of the social hierarchy is essential in both shaping the nature of social interactions between individuals and unfolding the structure of the social networks. The social hierarchy found in a social network can be utilized to improve the accuracy of link prediction, provide better query results, rank web pages, and study information flow and spread in complex networks. In this paper, we model a social network as a directed graph G, and consider the social hierarchy as DAG (directed acyclic graph) of G, denoted as G_D. By DAG, all the vertices in G can be partitioned into different levels, the vertices at the same level represent a disjoint group in the social hierarchy, and all the edges in DAG follow one direction. The main issue we study in this paper is how to find DAG G_D in G. The approach we take is to find G_D by removing all possible cycles from G such that $G = \mathcal{U}(G) \cup G_D$ where $\mathcal{U}(G)$ is a maximum Eulerian subgraph which contains all possible cycles. We give the reasons for doing so, investigate the properties of G_D found, and discuss the applications. In addition, we develop a novel two-phase algorithm, called Greedy-&-Refine, which greedily computes an Eulerian subgraph and then refines this greedy solution to find the maximum Eulerian subgraph. We give a bound between the greedy solution and the optimal. The quality of our greedy approach is high. We conduct comprehensive experimental studies over 14 real-world datasets. The results show that our algorithms are at least two orders of magnitude faster than the baseline algorithm.

1. INTRODUCTION

Social hierarchy refers to the pyramid structure of societies, with minority on the top and majority at the bottom, which is a prevalent and universal feature in organizations. Social hierarchy is also recognized as a fundamental characteristic of social interactions, being well studied in both sociology and psychology [16]. In recent years, social hierarchy has attracted considerable attention and generates profound and lasting influence in various fields, especially social networks. This is because the hierarchical structure of a population is essential in shaping the nature of social interactions between individuals and unfolding the structure of underlying social networks. Gould in [16] develops a formal theoretical model to model the emergence of social hierarchy, which can accurately predict the network structure. By the social status theory in [16], individuals with low status typically follow individuals with high status. Clauset et al. in [9] develop a technique to infer hierarchical structure of a social network based on the degree of relatedness between individuals. They show that the hierarchical structure can explain and reproduce some commonly observed topological properties of networks and can also be utilized to predict missing links in networks. Assuming that underlying hierarchy is the primary factor guiding social interactions, Maiya and Berger-Wolf in [28] infer social hierarchy from undirected weighted social networks based on maximum likelihood. With temporal collaboration networks, Wang et al. in [34] model the advisor-advisee relationship mining problem utilizing a jointly likelihood objective function, which can benefit applications such as visualization of genealogy and expert finding. Dong et al. in [10] study the interaction of social status and social networks in an enterprise and observe the tendency of high-status individuals to be spanned as "structural holes" over their subordinates and unveil the "rich club" effects between high-status individuals in enterprise networks. All the studies imply that social hierarchy is a primary organizing principle of social networks, capable of shedding light on many phenomena. In addition, social hierarchy is also used in many aspects of social network analysis and data mining. For instance, social hierarchy can be utilized to improve the accuracy of link prediction [27], provide better query results [21], rank web pages [17], study information flow and spread in complex networks [1, 3], and relationship categorization [32]. A comprehensive review of related topics can be found in [2].

In this paper, we focus on social networks that can be modeled by directed graphs, because in many social networks (e.g., Google+, Weibo, Twitter), information flow and influence propagate follow certain directions from vertices to vertices. Given a social network as a directed graph G, its social hierarchy can be represented as a directed acyclic graph (DAG). By DAG, all the vertices in G are partitioned into different levels (disjoint groups), and all the edges in the cycle-free DAG follow one direction, as observed in social networks that prestigious users at high levels are followed by users at low levels and the prestigious users typically do not follow their followers. Here, a level in DAG represents the status of a vertex in the hierarchy the DAG represents.

The issue we study in this paper is how to find hierarchy as a DAG in a general directed graph G which represents a social network. Given a graph G, there are many possible ways to obtain a DAG. First, converting graph G into a DAG, by contracting al-

l vertices in a strongly connected component in G as a vertex in DAG, does not serve the purpose, because all vertices in a strongly connected component do not necessarily belong to the same level in a hierarchy. Second, a random DAG does not serve the purpose, because it heavily relies on the way to select the vertices as the start to traverse and the way to traverse. Therefore, two random DAGs can be significantly different in topology. Third, finding the maximum DAG of G is NP-hard and approximating the maximum DAG within a factor better than 1/2 is Unique Games hard [19]. The way we do this is to find the DAG by removing all possible cycles from G following [18]. In [18] Gupte et al. propose a way to decompose a directed graph G into a maximum Eulerian subgraph $\mathcal{U}(G)$ and DAG G_D, s.t. $G = \mathcal{U}(G) \cup G_D$. Here, all possible cycles in G are in $\mathcal{U}(G)$, and all edges in G_D do not appear in $\mathcal{U}(G)$. We take the same approach to find DAG G_D for a graph G by finding the maximum Eulerian subgraph $\mathcal{U}(G)$ of G such that $G = \mathcal{U}(G) \cup G_D$, as given in [18].

Main contributions: We summarize the main contributions of our work as follows. First, unlike [18] which studies a measure between 0 and 1 to indicate how close a given directed graph is to a perfect hierarchy, we focus on the hierarchy (DAG). In addition to the properties investigated in [18], we show that G_D found is representative, exhibits the pyramid rank distribution. In addition, G_D found can be used to study social mobility and recover hidden directions of social relationships. Second, we significantly improve the efficiency of computing the maximum Eulerian subgraph $\mathcal{U}(G)$. Note that the time complexity of the *BF-U* algorithm [18] is $O(nm^2)$, where n and m are the numbers of vertices and edges, respectively. Such an algorithm is impractical, because it can only work on small graphs. We propose a new algorithm with time complexity $O(m^2)$, and propose a novel two-phase algorithm, called Greedy-&-Refine, which greedily computes an Eulerian subgraph in $O(n+m)$ and then refines this greedy solution to find the maximum Eulerian subgraph in $O(cm^2)$ where c is a very small constant less than 1. The quality of our greedy approach is high. Finally, we conduct extensive performance studies using 14 real-world datasets to evaluate our algorithms, and confirm our findings.

Further related works: Ball and Newman [4] analyze directed networks between students with both reciprocated and unreciprocated friendships and develop a maximum-likelihood method to infer ranks between students such that most unreciprocated friendships are from lower-ranked individuals to higher-ranked ones, corresponding to status theory [16]. Leskovec et al. in [25, 24] investigate signed networks and develop an alternate theory of status in place of the balance theory frequently used in undirected and unsigned networks to both explain edge signs observed and predict edge signs unknown. Influence has been widely studied [7], finding social hierarchy provides a new perspective to explore the influence given the existence of a social hierarchy.

Extracting the *MAX ACYCLIC SUBGRAPH* from a given directed graph G is one way to find the social hierarchy behind G, since it is to find an acyclic subgraph with the most edges for a given graph. However, Karp in [20] shows that it is NP-hard. Newman shows that it remains NP-hard on graphs with maximum degree 3 and is NP-hard to approximate within a factor greater than $\frac{65}{66}$ in [30]. Recently, Guruswami et al. [19] prove that it is Unique Games hard to approximate the *MAX ACYCLIC SUBGRAPH* problem within a factor better than $1/2$. Other approaches to obtain DAGs includes [11] which condenses all vertices in a SCC into a supernode and [31] that constructs DAGs with edges/paths to maximize influence propagation probabilities, providing influence probability on each edge.

Eulerian graphs have been well studied in the theory community [12, 13, 6, 8, 26]. For example, in [12], Fleischner gives a comprehensive survey on this topic. In [13], the same author surveys several applications of Eulerian graphs in graph theory. Another closely related concept is super-Eulerian graph, which contains a spanning Eulerian subgraph [6, 8, 26], here a spanning Eulerian subgraph means an Eulerian subgraph that includes all vertices. The problem of determining whether or not a graph is super-Eulerian is NP-complete [8]. Most of these works mainly focus on the properties of Eulerian subgraphs. There are not much related works on computing the maximum Eulerian subgraphs for large graphs. To the best of our knowledge, the only one in the literature is done by Gupte, et al. in [18]. However, the time complexity of their algorithm is $O(nm^2)$, which is clearly impractical for large graphs.

Organization: In Section 2, we focus on the properties of the social hierarchy found after giving some useful concepts on maximum Eulerian subgraph, and discuss the applications. In Section 3, we discuss an existing algorithm *BF-U* [18]. In Section 4, we propose a new algorithm *DS-U* of time complexity $O(m^2)$, and treat it as the baseline algorithm. We present a new two-phase algorithm *GR-U* for finding the maximum Eulerian subgraph, as well as its analysis in Section 5. Extensive experimental studies are reported in Section 6. Finally, we conclude this work in Section 7.

2. THE HIERARCHY

Consider an unweighted directed graph $G = (V, E)$, where $V(G)$ and $E(G)$ denote the sets of vertices and directed edges of G, respectively. We use $n = |V(G)|$ and $m = |E(G)|$ to denote the number of vertices and edges of graph G, respectively. In G, a path $p = (v_1, v_2, \cdots, v_k)$ represents a sequence of edges such that $(v_i, v_{i+1}) \in E(G)$, for each v_i $(1 \leq i < k)$. The length of path p, denoted as $\mathsf{len}(p)$, is the number of edges in p. A simple path is a path $(v_1, v_2, \cdots, v_k)$ with k distinct vertices. A cycle is a path where a same vertex appears more than once, and a simple cycle is a path $(v_1, v_2, \cdots, v_{k-1}, v_k)$ where the first $k-1$ vertices are distinct while $v_k = v_1$. For simplicity, below, we use V and E to denote $V(G)$ and $E(G)$ of G, respectively, when they are obvious. For a vertex $v_i \in V(G)$, the in-neighbors of v_i, denoted as $N_I(v_i)$, are the vertices that link to v_i, i.e., $N_I(v_i) = \{v_j \mid (v_j, v_i) \in E(G)\}$, and the out-neighbors of v_i, denoted as $N_O(v_i)$, are the vertices that v_i links to, i.e., $N_O(v_i) = \{v_j \mid (v_i, v_j) \in E(G)\}$. The in-degree $d_I(v_i)$ and out-degree $d_O(v_i)$ of vertex v_i are the numbers of edges that direct to and from v_i, respectively, i.e., $d_I(v_i) = |N_I(v_i)|$ and $d_O(v_i) = |N_O(v_i)|$.

A strongly connected component (SCC) is a maximal subgraph of a directed graph in which every pair of vertices v_i and v_j are reachable from each other.

A directed graph G is an Eulerian graph (or simply Eulerian) if for every vertex $v_i \in V(G)$, $d_I(v_i) = d_O(v_i)$. An Eulerian graph can be either connected or disconnected. An Eulerian subgraph of a graph G is a subgraph of G, which is Eulerian, denoted as G_U. The maximum Eulerian subgraph of a graph G is an Eulerian subgraph with the maximum number of edges, denoted as $\mathcal{U}(G)$. Given a directed graph G, we focus on the problem of finding its maximum Eulerian subgraph, $\mathcal{U}(G)$, which does not need to be connected. Note that the problem of finding the maximum Eulerian subgraph ($\mathcal{U}(G)$) in a directed graph can be solved in polynomial time, whereas the problem of finding the maximum connected Eulerian subgraph is NP-hard [5]. The following example illustrates the concept of maximum Eulerian subgraph.

Example 2.1: Fig. 1 shows a graph $G = (V, E)$ with 14 ver-

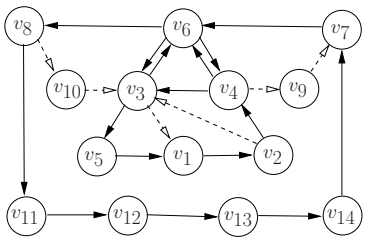


Figure 1: Illustration of the maximum Eulerian subgraph

tices and 22 edges. Its maximum Eulerian subgraph $\mathcal{U}(G)$ is a subgraph of G, where its edges are in solid lines: $E(\mathcal{U}(G)) = \{(v_1, v_2), (v_2, v_4), (v_4, v_3), (v_3, v_5), (v_5, v_1), (v_4, v_6), (v_6, v_4), (v_3, v_6), (v_6, v_3), (v_6, v_8), (v_8, v_{11}), (v_{11}, v_{12}), (v_{12}, v_{13}), (v_{13}, v_{14}), (v_{14}, v_7), (v_7, v_6)\}$, and $V(\mathcal{U}(G))$ is the set of vertices that appear in $E(\mathcal{U}(G))$.

The main issue here is to find a hierarchy of a directed graph G as DAG G_D by finding the maximum Eulerian subgraph $\mathcal{U}(G)$ for a directed graph G. With $\mathcal{U}(G)$ found, G_D can be efficiently found due to $G = \mathcal{U}(G) \cup G_D$, and $E(\mathcal{U}(G)) \cap E(G_D) = \emptyset$. We discuss the properties of the hierarchy G_D and the applications.

The representativeness: The maximum Eulerian subgraph $\mathcal{U}(G)$ for a general graph G is not unique. A natural question is how representative G_D is as the hierarchy. Note that G_D is only unique w.r.t $\mathcal{U}(G)$ found. Below, we show G_D identified by an arbitrary $\mathcal{U}(G)$ is representative based on a notion of *strictly-higher* defined between two vertices in G_D, over a ranking $r(\cdot)$ where $r(u) < r(v)$ for each edge $(u, v) \in G_D$. Here, for two vertices u and v, a larger rank implies a vertex is in a higher status in a follower relationship, and u is *strictly-higher* than v if $r(u) > r(v)$ and u is reachable from v, i.e. there is a directed path from v to u in G_D.

Theorem 2.1: *Let G_{D_1} and G_{D_2} be two DAGs for G such that $G = \mathcal{U}_1(G) \cup G_{D_1} = \mathcal{U}_2(G) \cup G_{D_2}$. There are no vertices u and v such that u is strictly-higher than v in G_{D_1} whereas v is strictly-higher than u in G_{D_2}.*

The proof is given in Appendix.

A case study: With the hierarchy (DAG G_D) found, suppose we assign every vertex u a non-negative rank $r(u)$ such that $r(u) < r(v)$ for any edge $(u, v) \in G_D$, then $r(\cdot)$ is a strictly-higher rank. In this paper, we assign each vertex u a rank $r(u)$ as follows

$$r(u) = \begin{cases} 0 & \text{if } d_I(u) \text{ in } G_D \text{ is } 0, \\ \max\{r(v) + 1 | (v, u) \in G_D\} & \text{otherwise} \end{cases}$$

It is noteworthy that isolated vertices in G_D are assigned to rank 0. In general, in a graph, there are large DAGs, small DAGs, and isolated vertices. Some vertices with $d_O(v) = d_I(v)$ are isolated vertices but not all vertices with $d_O(v) = d_I(v)$ are isolated vertices in DAG G_D found. Fig. 2 illustrates a subgraph extracted from Twitter, where the dashed edges represent the maximum Eulerian subgraph. Note that the graphs in Fig. 2 and Fig. 1 are different. After removing the maximum Eulerian subgraph, all the vertices except v_6 exist in DAG G_D. The number of isolated vertices for the large graphs tested are shown in Table 1 and Table 2, as $|V| - |V(G_D)|$. The numbers are less than 10% of the total number of vertices in G, and such vertices are in the middle/low position in the hierarchy. When the number of isolated vertices is large, $|E(\mathcal{U}(G))|/|E|$ intends to be large (refer to Table 1 and Table 2). This implies that all vertices are in similar rank, and therefore the graph G does not exhibit a social hierarchy as also observed by [13]. In the datasets wiki-vote, Gnutella, web-Google,

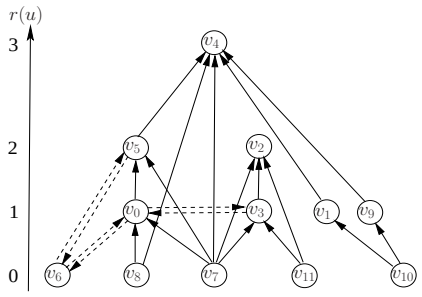
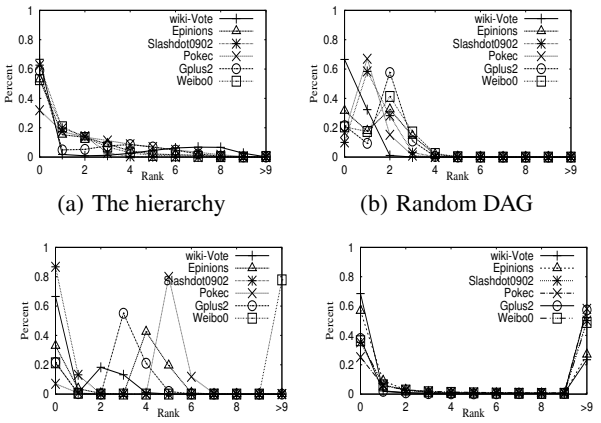


Figure 2: A subgraph extracted from Twitter ($\neq$ Fig. 1)



(a) The hierarchy (b) Random DAG

(c) SCC (d) Approximate Maximum DAG

Figure 3: Rank Distribution

etc. the DAG G_D covers the majority of vertices in $V(G)$, whereas in graphs with high $|E(\mathcal{U}(G))|/|E|$, e.g. Slashdot0811 and Slashdot0902, they do not exhibit explicit hierarchy structures. In such cases, like Slashdot0811 and Slashdot0902, assigning most vertices with the similar ranks is reasonable. Note that other rankings, for instance the ranking proposed in [18], generate similar results with marginal difference.

To show whether such ranking reflects the ground truth, as a case study, we conduct testing using Twitter, where the celebrities are known, for instance, refer to Twitter Top 100 (`http://twittercounter.com/pages/100`). We sample a subgraph among 41.7 million users (vertices) and 1.47 billion relationships (edges) from Twitter social graph **G** crawled in 2009 [22]. In brief, we randomly sample 5 vertices in the celebrity set given in Twitter, and then sample 1,000,000 vertices starting from the 5 vertices as seeds using random walk sampling [23]. We construct an induced subgraph G' of the 1,000,000 vertices sampled from **G**, and we uniformly sample about 10,000,000 edges from G' to obtain the sample graph G, which contains 759,105 vertices and 11,331,061 edges. In G, we label a vertex u as a celebrity, if u is a celebrity and has at least 100,000 followers in **G**. There are 430 celebrities, we manually verified, in G including Britney Spears, Oprah Winfrey, Barack Obama, etc. We compute the hierarchy (G_D) of G using our approach and rank vertices in G_D. The hierarchy reflects the truth: 88% celebrities are in the top 1% vertices and 95% celebrities in the top 2% vertices. In consideration of efficiency, we can approximate the exact hierarchy with a greedy solution obtained by *Greedy* in Section 5. In the approximate hierarchy, 85% celebrities are in the top 1% vertices and 93% celebrities in the top 2% vertices.

The pyramid structure of rank distribution is one of the most

Graph	$\|V\|$	$\|E\|$	$\|V(\mathcal{U}(G))\|$	$\|E(\mathcal{U}(G))\|$	$\|V(G_D)\|$
Gplus0	23,046	115,090	2,833	6,271	22,556
Gplus1	51,181	512,281	14,797	70,537	50,140
Gplus2	84,690	2,867,781	51,605	770,854	82,348
Gplus3	99,630	8,289,203	87,941	3,644,147	94,604
Weibo0	97,906	2,431,525	73,581	850,136	96,765
Weibo1	97,954	2,446,002	73,713	855,131	96,833
Weibo2	98,004	2,463,050	73,911	861,729	96,902
Weibo3	98,057	2,479,140	74,094	868,044	96,969

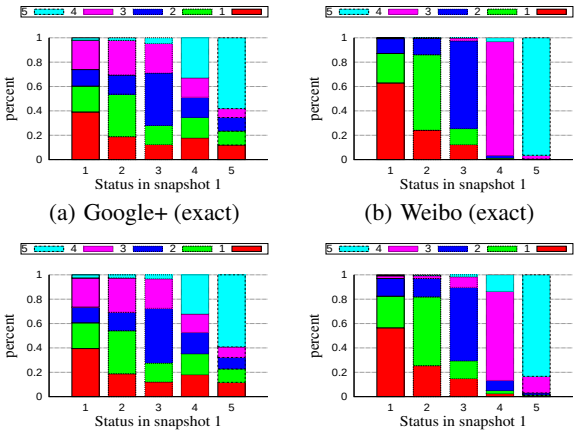
Table 1: To study social mobility

fundamental characteristics of social hierarchy. We test the social networks: wiki-Vote, Epinions, Slashdot0902, Pokec, Google+, Weibo. The details about the datasets are in Table 1 and Table 2. The rank distribution derived from hierarchy G_D, shown in Fig. 3(a), indicates the existence of pyramid structure, while the rank distributions derived from a random DAG (Fig. 3(b)), by contracting SCCs (Fig. 3(c)) and from an $\frac{1}{2}$-approximation maximum DAG (Fig. 3(d)) do not show such properties. Here, the x-axis is the rank where a high rank means a high status, and the y-axis is the percentage in a rank over all vertices. By analyzing the vertices, u, in G over the difference between in-degree and out-degree, i.e. $d_I(u) - d_O(u)$, it reflects the fact that those vertices u with negative $d_I(u) - d_O(u)$ are always at the bottom of G_D, whereas those vertices in the higher rank are typically with large positive $d_I(u) - d_O(u)$ values. Specifically, we divide all vertices into 5 equal groups in terms of rank and degree difference ($d_I(u) - d_O(u)$) respectively, then 93.75% (94.11%) of the top (bottom) 20% in terms of rank appear in the corresponding top (bottom) 20% in terms of degree difference.

The social mobility: With the DAG G_D found, we can further study social mobility over the social hierarchy G_D represents. Here, social mobility is a fundamental concept in sociology, economics and politics, and refers to the movement of individuals from one status to another. It is important to identify individuals who jump from a low status (a level in G_D) to a high status (a level in G_D). We conduct experimental studies using the social network Google+ (`http://plus.google.com`) crawled from Jul. 2011 to Oct. 2011 [15, 14], and Sina Weibo (`http://weibo.com`) crawled from 28 Sep. 2012 to 29 Oct. 2012 [35]. For Google+ and Weibo, we randomly extract 100,000 vertices respectively, and then extract all edges among these vertices in 4 time intervals during the period the datasets are crawled, as shown in Table 1.

We show social mobility in Fig. 4. We compare two snapshots, G_1 and G_2, and investigate the social mobility from G_1 to G_2. For Google+, G_1 and G_2 are Gplus0 and Gplus1, and for Weibo, G_1 and G_2 are Weibo0 and Weibo1. For G_1, we divide all vertices into 5 equal groups in terms of the ranking derived. The top 20% go into group 5, and the second 20% go to group 4, for example. In Fig. 4, the x-axis shows the 5 groups for G_1. Consider the number of vertices in a group as 100%. In Fig. 4, we show the percentage of vertices in one group moves to another group in G_2. Fig. 4(a) and Fig. 4(b) show the results for Google+ and Weibo. Some observations can be made. Google+ is a new social network when crawled since it starts from Jun. 29, 2011, and Weibo is a rather mature social network since it starts from Aug. 14, 2009. From Fig. 4(a), many vertices move from one status to another, whereas from Fig. 4(b), only a very small number of vertices move from one status to another. Similar results can be observed from approximate hierarchies, by our greedy solution *Greedy* given in Section 5, as shown in Fig. 4(c) and Fig. 4(d). Those moved to/from the highest level deserve to be investigated.

Recovering the hidden directions is to identify the direction of



(a) Google+ (exact) (b) Weibo (exact)

(c) Google+ (approx) (d) Weibo (approx)

Figure 4: Social mobility result from hierarchy

Algorithm 1 *BF-U* (G)

Input: A graph $G = (V, E)$
Output: Two subgraphs of G, $\mathcal{U}(G)$ and G_D ($G = \mathcal{U}(G) \cup G_D$)

1: $w(v_i, v_j) \leftarrow -1$ for each edge $(v_i, v_j) \in E$;
2: **while** there is a negative cycle p_c in G **do**
3: **for** every edge (v_i, v_j) in the negative cycle p_c **do**
4: $w(v_i, v_j) \leftarrow -w(v_i, v_j)$;
5: Reverse the direction of the edge (v_i, v_j) to be (v_j, v_i);
6: **end for**
7: **end while**
8: G_D is a subgraph that contains all edges with weight -1;
9: $\mathcal{U}(G)$ is a subgraph containing the reversed edges with weight +1;

an edge if the direction of the edge is unknown [36]. The directionality of edges in social networks being recovered is important in many social analysis tasks. We show that our approach has advantage over the semi-supervised approach (SM-ReDirect) in [36]. Here, the task is using the given 20% directed edges as training data to recover the directions for the remaining edges. In our approach, we construct a graph G from the training data, and identify G_D by $G = \mathcal{U}(G) \cup G_D$. With the ranking $r(\cdot)$ over the vertices, we predict the direction of an edge (u, v) is from u to v if $r(v) > r(u)$. It is worth noting that $r(u)$ for vertex u that are not covered by the training set is randomly assigned according to the rank distribution. Take Slashdot and Epinion datasets used [36], our approach outperforms the matrix-factorization based SM-ReDirect both in terms of accuracy and efficiency. For Slashdot, our prediction accuracy is 0.7759 whereas SM-ReDirect is 0.6529. For Epinion, ours is 0.8285 whereas SM-Redirect is 0.7118. Using approximate hierarchy, our accuracy is 0.7682 for Slashdot and 0.8277 for Epinion, respectively. In addition, we take G_2 as the training set and predict the directions of edges in $G_3 \setminus G_2$, the prediction accuracy is 0.7108 for Google+ and 0.8006 for Weibo, respectively.

3. THE EXISTING ALGORITHM

To find the maximum Eulerian subgraph, Gupte, et al. in [18] propose an iterative algorithm based on the Bellman-Ford algorithm, which we call *BF-U* (Algorithm 1). Let $w(v_i, v_j)$ be a weight assigned to an edge (v_i, v_j) in G. Initially, *BF-U* assigns an edge-weight with a value of -1 to every edge in graph G (Line 1). Let a negative cycle be a cycle with a negative sum of edge weights. In every iteration (Lines 2-7), *BF-U* finds a negative cycle p_c repeatedly until there are no negative cycles. For every edge (v_i, v_j)

Algorithm 2 *DS-U* (G)

Input: A graph $G = (V, E)$
Output: Two subgraphs of G, $\mathcal{U}(G)$ and G_D ($G = \mathcal{U}(G) \cup G_D$)

1: **for** each edge (v_i, v_j) in $E(G)$ **do** $w(v_i, v_j) \leftarrow -1$;
2: **for** each vertex u in $V(G)$ **do** $dst(u) \leftarrow 0$, $relax(u) \leftarrow true$, $pos(u) \leftarrow 0$;
3: **while** there is a vertex $u \in V(G)$ such that $relax(u) = true$ **do**
4: $S_V \leftarrow \emptyset$, $S_E \leftarrow \emptyset$, $NV \leftarrow \emptyset$;
5: **if** *FindNC* (G, u) **then**
6: **while** S_V.top() $\neq NV$ **do**
7: S_V.pop(); $(v_i, v_j) \leftarrow S_E$.pop();
8: $w(v_i, v_j) \leftarrow -w(v_i, v_j)$;
9: Reverse the direction of the edge (v_i, v_j) to be (v_j, v_i);
10: **end while**
11: S_V.pop(); $(v_i, v_j) \leftarrow S_E$.pop();
12: $w(v_i, v_j) \leftarrow -w(v_i, v_j)$;
13: Reverse the direction of the edge (v_i, v_j) to be (v_j, v_i);
14: **end if**
15: **end while**
16: G_D is a subgraph that contains all edges with weight -1;
17: $\mathcal{U}(G)$ is a subgraph containing the reversed edges with weight +1;

Algorithm 3 *FindNC* (G, u)

1: S_V.push(u);
2: **for** each edge (u, v) starting at $pos(u)$ in $E(G)$ **do**
3: $pos(u) \leftarrow pos(u) + 1$;
4: **if** $dst(u) + w(u, v) < dst(v)$ **then**
5: $dst(v) \leftarrow dst(u) + w(u, v)$;
6: $relax(v) \leftarrow true$, $pos(v) \leftarrow 0$;
7: **if** v is not in S_V **then**
8: S_E.push((u, v));
9: **if** *FindNC* (G, v) **then return** $true$; **endif**
10: **else**
11: S_E.push((u, v)); $NV \leftarrow v$; **return** $true$;
12: **end if**
13: **end if**
14: **end for**
15: $relax(u) \leftarrow false$;
16: S_V.pop(); S_E.pop() if S_E is not empty; **return** $false$;

in the negative cycle p_c found, it changes the weight of (v_i, v_j) to be $-w(v_i, v_j)$ and reverses the direction of the edge (Lines 4-5). As a result, it finds a maximum Eulerian subgraph $\mathcal{U}(G)$ and a directed acyclic graph (DAG) of G, denoted as G_D, such that $G = \mathcal{U}(G) \cup G_D$. Since the number of edges with weight +1 increases by at least one during each iteration, there are at most $O(m)$ iterations (Line 2-7). In every iteration it has to invoke the Bellman-Ford algorithm to find a negative cycle (Line 2), or to determine whether there is a negative cycle. The time complexity of Bellman-Ford algorithm is $O(nm)$. Therefore, in the worst case, the total time complexity of *BF-U* is $O(nm^2)$, which is too expensive for real-world graphs.

4. A NEW ALGORITHM

To address the scalability problem of *BF-U*, we propose a new algorithm, called *DS-U*. Different from *BF-U* which starts by finding a negative cycle using the Bellman-Ford algorithm in every iteration, *DS-U* finds a negative cycle only when necessary with condition. In brief, in every iteration, when necessary, *DS-U* invokes an algorithm *FindNC* (short for find a negative cycle) to find a negative cycle while relaxing vertices following *DFS* order. Applying amortized analysis [33], we prove the time complexity of *DS-U*, in the worst case, is $O(m^2)$ to find the maximum Eulerian subgraph $\mathcal{U}(G)$.

The *DS-U* algorithm is outlined in Algorithm 2, which invokes *FindNC* (Algorithm 3) to find a negative cycle. Here, *FindNC* is designed based on the same idea of relaxing edges as used in the Bellman-Ford algorithm. In addition to edge weight $w(v_i, v_j)$, we use three variables for every vertex u, $relax(u)$, $pos(u)$, and $dst(u)$. Here, $relax(u)$ is a Boolean variable indicating whether there are out-going edges from u that may need to relax to find a negative cycle. It will try to relax an edge from u further when $relax(u) = true$. When relaxing from u, $pos(u)$ records the next vertex v in $N_O(u)$ (maintained as an adjacent list) for the edge (u, v) to be relaxed next. It means that all edges from u to any vertex before $pos(u)$ has already been relaxed. $dst(u)$ is an estimation on the vertex u which decreases when relaxing. When $dst(u)$ decreases, $relax(u)$ is reset to be $true$ and $pos(u)$ is reset to be 0, since all its out-going edges can be possibly relaxed again. Initially, in *DS-U*, every edge weight $w(v_i, v_j)$ is initialized to -1, and the three variables, $relax(u)$, $pos(u)$, and $dst(u)$, on every vertex u are initialized to $true$, 0, and 0, respectively. All $w(v_i, v_j)$, $relax(u)$, $pos(u)$, and $dst(u)$ are used in *FindNC* to find a negative cycle following the main idea of Bellman-Ford algorithm in DFS order. A negative cycle, found by *FindNC* while relaxing edges, is maintained using a vertex stack S_V and an edge stack S_E together with a variable NV, where NV maintains the first vertex of a negative cycle. In *DS-U*, by popping vertex/edges from S_v/S_E until encountering the vertex in NV, a negative cycle can be recovered. As shown in Algorithm 2, in the while statement (Lines 3-15), for every vertex u in $V(G)$, only when there is a possible relax ($relax(u) = true$) and there is a negative cycle found by the algorithm *FindNC*, it will reverse the edge direction and update the edge weight, $w(v_i, v_j)$, for each edge (v_i, v_j) in the negative cycle (Lines 6-13).

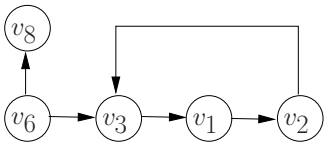


Figure 5: A subgraph G of Fig. 1

Example 4.1: We explain *DS-U* (Algorithm 2) using an example graph G in Fig. 5. For ever vertex u, there is an adjacent list to maintain its out-neighbors. Initially, for every vertex u, $dst(u) = 0, relax(u) = true, pos(u) = 0$; and for every edge (v_i, v_j), $w(v_i, v_j) = -1$. Suppose we process v_6, v_3, v_1, v_2, v_8 in such an order. In the first iteration, $relax(v_6) = true$ and *FindNC* (G, v_6) returns true, which implies a negative cycle is found. Here, for all vertices in G, we have $dst(v_6) = 0, relax(v_6) = true, pos(v_6) = 1$; $dst(v_3) = -4, relax(v_3) = true, pos(v_3) = 0$; $dst(v_1) = -2, relax(v_1) = true, pos(v_1) = 0$; $dst(v_2) = -3, relax(v_2) = true, pos(v_2) = 0$; $dst(v_8) = 0, relax(v_8) = true, pos(v_8) = 0$. In addition, $NV = v_3$, $S_V = \{v_6, v_3, v_1, v_2\}$, $S_E = \{(v_6, v_3), (v_3, v_1), (v_1, v_2), (v_2, v_3)\}$. Following Lines 6–13 in Algorithm 2, we find and reverse negative cycle (v_3, v_1, v_2, v_3) and make $w(v_3, v_2) = w(v_2, v_1) = w(v_1, v_3) = 1$. In the second iteration, the out-neighbors of v_6 are relaxed from $pos(v_6) = 1$ in v_6's adjacent list, i.e. from edge (v_6, v_8). *FindNC* (G, v_6) returns false. We have, $dst(v_6) = 0, relax(v_6) = false, pos(v_6) = 2$, and $dst(v_8) = -1, relax(v_8) = false, pos(v_8) = 1$. In the following iterations (*FindNC* (G, v_3), *FindNC* (G, v_1), and *FindNC* (G, v_2)), all return false. Finally, for vertex v_8, since $relax(v_8) = false$, *FindNC* (G, v_8) is unnecessary, and *DS-U* (G) terminates. It finds the maximum Eulerian subgraph $\mathcal{U}(G) = \{(v_3, v_1), (v_1, v_2), (v_2, v_3)\}$.

Lemma 4.1: *In Algorithm 2, if there is a negative cycle C, $relax(u) = true$ holds for at least one vertex $u \in V(C)$.*

Proof Sketch: Assume the opposite, i.e., there exists a negative cycle C such that for every vertex $u \in V(C)$, $relax(u) = false$. Let $C = (v_0, v_1, \ldots, v_{k-1}, v_k = v_0)$, then $\sum_{i=0}^{k-1} w(v_i, v_{i+1}) < 0$. Since $relax(v_i) = false$ holds for $i = 0, 1, \ldots, k-1$, then $dst(v_{i+1}) \leq dst(v_i) + w(v_i, v_{i+1})$. It leads to a contradiction, if summing both sides from $i = 0$ to $i = k-1$, then $dst(v_0) = dst(v_k) \leq dst(v_0) + \sum_{i=0}^{k-1} w(v_i, v_{i+1}) < dst(v_0)$. □

Theorem 4.1: *Algorithm 2 correctly finds the maximum Eulerian subgraph $\mathcal{U}(G)$ when it terminates.*

Proof Sketch: It can be proved by Lemma 4.1. □

Theorem 4.2: *Time complexity of DS-U (G) is $O(m^2)$.*

The proof sketch is given in Appendix.

5. THE OPTIMAL: GREEDY-&-REFINE

DS-U reduces the time complexity of *BF-U* to $O(m^2)$, but it is still very slow for large graphs. To further reduce the running time of *DS-U*, we propose a new two-phase algorithm which is shown to be two orders of magnitude faster than *DS-U*. Below, we first introduce an important observation which can be used to prune many unpromising edges. Then, we will present our new algorithms as well as theoretical analysis.

Let $\mathcal{S}$ be a set of strongly connected components (SCCs) of G, such that $\mathcal{S} = \{G_1, G_2, \cdots\}$, where G_i is an SCC of G, $G_i \subseteq G$, and $G_i \cap G_j = \emptyset$ for $i \neq j$. We show that for any edge, if it is not included in any SCC G_i of G, then it cannot be contained in the maximum Eulerian subgraph $\mathcal{U}(G)$. Therefore, the problem of finding the maximum Eulerian subgraph of G becomes a problem of finding the maximum Eulerian subgraph of each $G_i \in \mathcal{S}$, since the union of the maximum Eulerian subgraph of $G_i \in \mathcal{S}$, $1 \leq i \leq |\mathcal{S}|$, is the maximum Eulerian subgraph of G.

Lemma 5.1: *An Eulerian graph G can be divided into several edge disjoint simple cycles.*

Theorem 5.1: *Let G be a directed graph, and $\mathcal{S} = \{G_1, G_2, \cdots\}$ be a set of SCCs of G. The maximum Eulerian subgraph of G, $\mathcal{U}(G) = \bigcup_{G_i \in \mathcal{S}} \mathcal{U}(G_i)$.*

The proof of Lemma 5.1 and Theorem 5.1 are given in Appendix.

Below, we discuss how to find the maximum Eulerian subgraph for each strongly connected component (SCC) G_i of G. In the following discussion, we assume that a graph G is an SCC itself.

We can use *DS-U* to find the maximum Eulerian subgraph for an SCC G. However, *DS-U* is still too expensive to deal with large graphs. The key issue is that the number of iterations in *DS-U* (Algorithm 2, Lines 3-15), can be very large when the graph and its maximum Eulerian subgraph are both very large. Since in most iterations, the number of edges with weight +1 increases only by 1, it takes almost $|E(\mathcal{U}(G))|$ iterations to get the optimal number of edges in the maximum Eulerian subgraph $\mathcal{U}(G)$.

In order to reduce the number of iterations, we propose a two-phase Greedy-&-Refine algorithm, abbreviated by *GR-U*. Here, a *Greedy* algorithm computes an Eulerian subgraph of G, denoted as $\widetilde{\mathcal{U}}(G)$, and a *Refine* algorithm refines the greedy solution $\widetilde{\mathcal{U}}(G)$ to get the maximum Eulerian subgraph $\mathcal{U}(G)$, which needs at most $|E(\mathcal{U}(G))| - |E(\widetilde{\mathcal{U}}(G))|$ iterations. The *GR-U* algorithm is given in Algorithm 4, and an overview is shown in Fig. 6. In Algorithm 4, it first computes all SCCs (Line 1). For each SCC G_i, it computes an Eulerian subgraph using *Greedy*, denoted as $\widetilde{\mathcal{U}}(G_i)$ (Line 3). In *Greedy*, in every iteration l ($1 \leq l \leq l_{max}$), it identifies a subgraph by an *l-Subgraph* algorithm, and further deletes/reverses all specific length-l paths called pn-paths which we will discuss in details by *DFS*. Note l_{max} is a small number. After computing $\widetilde{\mathcal{U}}(G_i)$, $G_i - \widetilde{\mathcal{U}}(G_i)$ is near acyclic, and it moves all cycles from $G_i - \widetilde{\mathcal{U}}(G_i)$ to $\widetilde{\mathcal{U}}(G_i)$ (Line 4). Finally, it refines $\widetilde{\mathcal{U}}(G_i)$ to obtain the optimal $\mathcal{U}(G_i)$ by calling *Refine* (Line 5). The union of all $\mathcal{U}(G_i)$ is the maximum Eulerian subgraph for G. Below, we first list some important concepts introduced in the algorithm and analysis parts (refer to Table 6 in Appendix), and then we shall detail the greedy algorithm and refine algorithm, respectively.

Algorithm 4 *GR-U* (G)

1: Compute SCCs of G, $\mathcal{S} = \{G_1, G_2, \cdots\}$;
2: **for** each $G_i \in \mathcal{S}$ **do**
3: $\quad\widetilde{\mathcal{U}}(G_i) \leftarrow$ *Greedy* (G_i);
4: $\quad$Move all cycles found in $G_i - \widetilde{\mathcal{U}}(G_i)$ to $\widetilde{\mathcal{U}}(G_i)$; {Make $G_i - \widetilde{\mathcal{U}}(G_i)$ acyclic}
5: $\quad\mathcal{U}(G_i) \leftarrow$ *Refine* $(\widetilde{\mathcal{U}}(G_i), G_i)$;
6: **end for**
7: **return** $\bigcup_{i=1}^{|\mathcal{S}|} \mathcal{U}(G_i)$;

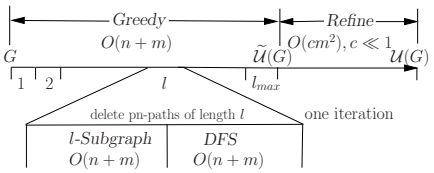


Figure 6: An Overview of Greedy-&-Refine

5.1 The Greedy Algorithms

Given a graph G, we propose two algorithms to obtain an initial Eulerian subgraph $\widetilde{\mathcal{U}}(G)$. The first algorithm is denoted as *Greedy-D* (Algorithm 5), which deletes edges from G to make $d_I(v) = d_O(v)$ for every vertex v in $\widetilde{\mathcal{U}}(G)$. The second algorithm is denoted as *Greedy-R* (Algorithm 8), which reverses edges instead of deletion to the same purpose. We use *Greedy* when we refer to either of these two algorithms. By definition, the resulting $\widetilde{\mathcal{U}}(G)$ is an Eulerian subgraph of G. The more edges we have in $\widetilde{\mathcal{U}}(G)$, the closer the resulting subgraph $\widetilde{\mathcal{U}}(G)$ is to $\mathcal{U}(G)$. We discuss some notations below (refer to Table 6 in Appendix).

The vertex label: For each vertex u in G, we define a vertex label on u, $\mathsf{label}(u) = d_O(u) - d_I(u)$. If $\mathsf{label}(u) = 0$, it means that u can be a vertex in an Eulerian subgraph without any modifications. If $\mathsf{label}(u) \neq 0$, it needs to delete/reverse some adjacent edges to make $\mathsf{label}(u)$ zero.

The pn-path: We also define a positive-start and negative-end path between two vertices, u and v, denoted as pn-path (u, v). Here, pn-path (u, v) is a path $p = (v_1, v_2, \cdots, v_l)$, where $u = v_1$ and $v = v_l$ with the following conditions: $\mathsf{label}(u) > 0$, $\mathsf{label}(v) < 0$, and all $\mathsf{label}(v_i) = 0$ for $1 < i < l$. Clearly, by this definition, if we delete all the edges in pn-path (u, v), then $\mathsf{label}(u)$ decreases by 1, $\mathsf{label}(v)$ increases by 1, and all intermediate vertices in pn-path (u, v) will have their labels as zero. To make all vertex labels being zero, the total number of such pn-paths to be deleted/reversed is $N = \sum_{\mathsf{label}(u)>0} \mathsf{label}(u)$.

The transportation graph G^T: A transportation graph G^T of G is a graph such that $V(G^T) = V(G)$ and $E(G^T) = \{(u, v) \mid (v, u) \in E(G)\}$.

The level **and** rlevel: $\mathsf{level}(v)$ is the shortest distance from any vertex u with a positive label, $\mathsf{label}(u) > 0$, in G. $\mathsf{rlevel}(v)$ is the shortest distance from any vertex u with a positive label,

Algorithm 5 *Greedy-D* (G)

1: $l \leftarrow 1$; $G' \leftarrow G$;
2: **while** some vertex $u \in G'$ with $\mathsf{label}(u) > 0$ **do**
3: $\quad G' \leftarrow$ *PN-path-D* (G', l); $l \leftarrow l + 1$;
4: **end while**
5: **return** G';

Algorithm 6 *PN-path-D* (G, l)

1: $G_l \leftarrow l$-*Subgraph* (G, l);
2: Enqueue all vertices $u \in V(G_l)$ with $\mathsf{label}(u) > 0$ into queue Q;
3: **while** $Q \neq \emptyset$ **do**
4: $\quad u \leftarrow Q$.top();
5: $\quad$ Following *DFS* starting from u over G_l, traverse unvisited edges and mark them "visited"; let the path from u to v be pn-path (u, v), when it reaches the first vertex v in G_l with $\mathsf{level}(v) = l$;
6: $\quad$ **if** pn-path $(u, v) \neq \emptyset$ **then**
7: $\quad\quad$ delete all edges in pn-path (u, v) from G;
8: $\quad\quad$ $\mathsf{label}(u) \leftarrow \mathsf{label}(u) - 1$; $\mathsf{label}(v) \leftarrow \mathsf{label}(v) + 1$;
9: $\quad\quad$ **if** $\mathsf{label}(u) = 0$ **then** Q.dequeue();
10: $\quad$ **else**
11: $\quad\quad$ Q.dequeue();
12: $\quad$ **end if**
13: **end while**
14: **return** G;

$\mathsf{label}(u) > 0$, in G^T. Note $\mathsf{rlevel}(v)$ is the shortest distance to any vertex u with a negative label, $\mathsf{label}(u) < 0$, in G.

5.1.1 The Greedy-D Algorithm

Below, we first concentrate on *Greedy-D* (Algorithm 5). Let G' be G (Line 1). In the **while** loop (Lines 2-4), it repeatedly deletes all pn-paths starting from length $l = 1$ by calling an algorithm *PN-path-D* (Algorithm 6) until no vertex u in G' with a positive value ($\mathsf{label}(u) > 0$).

Example 5.1: Consider graph G in Fig. 7. Three vertices, v_2, v_4, and v_8, in double cycles, have a label +1, and three other vertices, v_1, v_3, and v_7, in dashed cycles, have a label -1. Initially, $l = 1$, *Greedy-D* (Algorithm 5) deletes pn-path (v_2, v_3), making $\mathsf{label}(v_2) = \mathsf{label}(v_3) = 0$. When $l = 2$, pn-path $(v_4, v_1) = (v_4, v_3, v_1)$ is deleted. Finally, when $l = 5$, pn-path $(v_8, v_7) = (v_8, v_{11}, v_{12}, v_{13}, v_{14}, v_7)$ will be deleted. In Fig. 7, the graph with solid edges is $\widetilde{\mathcal{U}}(G)$ or the graph G' returned by Algorithm 5. It is worth mentioning that for the same graph G, *DS-U* needs 10 iterations. From the Eulerian subgraph $\widetilde{\mathcal{U}}(G)$ obtain by *Greedy*, it only needs at most 2 additional iterations to get the maximum Eulerian subgraph.

It is worth noting that $\widetilde{\mathcal{U}}(G)$ is not optimal. Some edges in $\widetilde{\mathcal{U}}(G)$ may not be in the maximum Eulerian subgraph, while some edges deleted should appear in $\mathcal{U}(G)$. In next section, we will discuss how to obtain the maximum Eulerian subgraph $\mathcal{U}(G)$ from the greedy solution $\widetilde{\mathcal{U}}(G)$.

Finding all pn-paths with length l: The *PN-path-D* algorithm is shown in Algorithm 6. In brief, for a given graph G, *PN-path-D* first extracts a subgraph $G_l \subseteq G$ which contains all pn-paths of length l that are possible to be deleted from G by calling an algorithm l-*Subgraph* (Algorithm 7) in Line 1. In other words, all edges in $E(G)$ but not in $E(G_l)$ cannot appear in any pn-paths with a length $\leq l$. Based on G_l obtained, *PN-path-D* deletes pn-paths from G (not from G_l) with additional conditions (in Lines 2-13). Let G'_l be a subgraph of G_l that includes all edges appearing in pn-paths of length l to be deleted in *PN-path-D*. *PN-path-D* will return a subgraph $G \setminus G'_l$ as a subgraph of G, which will be used in the next run in *Greedy-D* for deleting pn-paths with length $l + 1$.

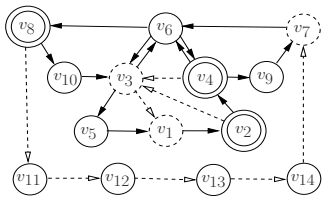
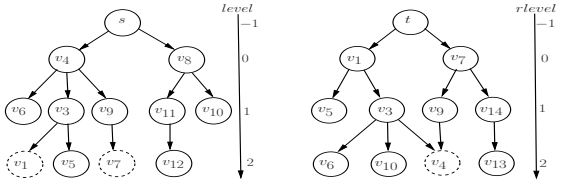


Figure 7: An Eulerian subgraph by *Greedy* for G in Fig. 1



(a) *BFS*-Tree of *BFS* (G, s) (b) *BFS*-Tree of *BFS* (G^T, t)

Figure 8: *BFS*-Trees used for constructing l-*Subgraph*

We discuss the l-*Subgraph* algorithm (Algorithm 7), which extracts G_l from G by *BFS* (breadth-first-search) traversing G twice. In the first *BFS* (Lines 4-6), it adds a virtual vertex s, and adds an edge (s, u) to every vertex u with a positive label ($\mathsf{label}(u) > 0$) in G. Then, it assigns a level to every vertex in G as follows. Let $\mathsf{level}(s)$ be -1. By *BFS*, it assigns $\mathsf{level}(u)$ to be $\mathsf{level}(\mathsf{parent}(u))+1$, where $\mathsf{parent}(u)$ is the parent vertex of u following *BFS*. In the second *BFS* (Lines 7-10), it conceptually considers the transposition graph G^T of G by reversing every edge $(v, u) \in E(G)$ as $(u, v) \in E(G^T)$ (Line 7). Then, it assigns a different rlevel to every vertex in G using the transposition graph G^T. Like the first *BFS*, it adds a virtual vertex t, and adds an edge (t, u) to every vertex u with a negative label ($\mathsf{label}(u) < 0$) in G^T. Then, it assigns rlevel to every vertex in G^T as follows. Let $\mathsf{rlevel}(t)$ be -1. By *BFS*, it assigns $\mathsf{rlevel}(u)$ to be $\mathsf{rlevel}(\mathsf{parent}(u))+1$, where $\mathsf{parent}(u)$ is the parent vertex of u in G^T following *BFS*. The resulting subgraph G_l to be returned from l-*Subgraph* is extracted as follows. Here, $V(G_l)$ contains all vertices u in G if $\mathsf{level}(u)+\mathsf{rlevel}(u) = l$ for the given length l, and $E(G_l)$ contains all edges (u, v) if both u and v appear in $V(G_l)$, (u, v) is an edge in the given graph G, and $\mathsf{level}(u) + 1 = \mathsf{level}(v)$ (Lines 11-13). The following example illustrates how l-*Subgraph* algorithm works.

Example 5.2: Fig. 9 illustrates the G_l returned by l-*Subgraph* (Algorithm 7) when $l = 2$. It is constructed using two *BFS*, i.e., *BFS* (G, s) and *BFS* (G^T, t), and the associated *BFS*-trees with level ≤ 2 and $\mathsf{rlevel} \leq 2$ are shown in Fig. 8(a) and Fig. 8(b), respectively. In Fig. 8(a), vertices v_1 and v_7 are the only vertices with $\mathsf{label} < 0$. In Fig. 8(b), vertex v_4 is the only one with $\mathsf{label} > 0$. Therefore, G_l contains only four edges, in dashed lines, which is much smaller than the original graph G to be handled.

Lemma 5.2: *By l-Subgraph, the resulting subgraph G_l includes all* pn-path*s of length l in G.*

Proof Sketch: Recall that l-*Subgraph* returns a graph G_l where $V(G_l) = \{u \mid \mathsf{level}(u)+\mathsf{rlevel}(u) = l\}$ and $E(G_l) = \{(u, v) \mid u \in V(G_l), v \in V(G_l), (u, v) \in E(G), \mathsf{level}(u) + 1 = \mathsf{level}(v)\}$. It implies the following. All vertices in G_l are on at least one shortest path from a positive label vertex u ($\mathsf{label}(u) > 0$) to a negative label vertex v ($\mathsf{label}(v) < 0$) of length l. All edges are on such shortest paths. No any edge in a pn-path of length l will be excluded from G_l. In other words, there does not exist an edge (u', v') on pn-path (u, v) of length l, which does not appear in $E(G_l)$. □

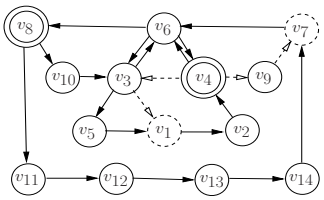


Figure 9: An *l-Subgraph* for length $l = 2$

Algorithm 7 *l-Subgraph* (G, l)

1: **for** each vertex u in $V(G)$ **do**
2: $\mathsf{level}(u) \leftarrow \infty$, $\mathsf{rlevel}(u) \leftarrow \infty$;
3: **end for**
4: Add a virtual vertex s and an edge (s, u) from s to every vertex u in G if $\mathsf{label}(u) > 0$;
5: $\mathsf{level}(s) \leftarrow -1$;
6: $\mathsf{level}(u) \leftarrow \mathsf{level}(\mathsf{parent}(u)) + 1$ for all vertices u in G following *BFS* staring from s;
7: Construct a graph G^T where $V(G^T) = V(G)$ and $E(G^T) = \{(u, v)|(v, u) \in E(G)\}$;
8: Add a virtual vertex t and an edge (t, u) from s to every vertex u in G^T if $\mathsf{label}(u) < 0$;
9: $\mathsf{rlevel}(t) \leftarrow -1$;
10: $\mathsf{rlevel}(u) \leftarrow \mathsf{rlevel}(\mathsf{parent}(u)) + 1$ for all vertices u in G^T following *BFS* staring from t;
11: Extract a subgraph G_l;
12: $V(G_l) = \{u \mid \mathsf{level}(u) + \mathsf{rlevel}(u) = l\}$;
13: $E(G_l) = \{(u, v) \mid u \in V(G_l), v \in V(G_l), (u, v) \in E(G), \mathsf{level}(u) + 1 = \mathsf{level}(v)\}$;
14: **return** G_l;

We explain *PN-path-D* (Algorithm 6). Based on G_l obtained from G using *l-Subgraph* (Algorithm 7), in *PN-path-D*, we delete all possible pn-paths of length l from G (Lines 2-13). The deletion of all pn-paths of length l from the given graph G is done using *DFS* over G_l with a queue Q. It first pushes all vertices u in $V(G_l)$ with a positive label ($\mathsf{label}(u) > 0$) into queue Q, because they are the starting vertices of all pn-paths with length l. We check the vertex u on the top of queue Q. With the vertex u, we do *DFS* starting from u over G_l, traverse unvisited edges in G_l, and mark the edges visited as "visited". Let p be the first pn-path (u, v) with length l along *DFS*. We delete all edges on p, and adjust the labels as to reduce $\mathsf{label}(u)$ by 1 and increase $\mathsf{label}(v)$ by 1. We dequeue u from queue Q until we cannot find any more pn-paths of length l starting from u, i.e. p returned by *DFS* (u) is empty. It is important to note that we only visit each edge at most once. There are two cases. One is that the edges visited will be deleted and there is no need to revisit. The other is that they are marked as "visited" but not included in any pn-paths with length l. For this case, these edges will not appear in any other pn-paths starting from any other vertices.

Lemma 5.3: *By PN-path-D, all* pn-path*s of length l are deleted.*

Proof Sketch: It can be proved based on *DFS* over G_l obtained from *l-Subgraph*.

Lemma 5.4: *By PN-path-D, the resulting G does not include any* pn-path*s of length $\leq l$.*

Proof Sketch: Let G'_i be the resulting graph of *PN-path-D* after deleting all pn-paths of length i from G. It is trivial when $i = 1$. Assume that it holds for G'_i when $i < l$. We prove that G'_i holds when $i = l$. First, there are no pn-paths of length $\leq l - 1$ in graph G'_{l-1} as a result of *PN-path-D* by assumption. Second, $G'_l \subseteq G'_{l-1}$ because G'_l is obtained by deleting pn-paths of length l from G'_{l-1}, as given in the *Greedy-D* algorithm (Algorithm 5).

Algorithm 8 *Greedy-R* (G)

1: $l \leftarrow 1$;
2: Assign an initial value of -1 to the weight $w(v_i, v_j)$ for every edge $(v_i, v_j) \in E$;
3: **while** some vertex $u \in G$ with $\mathsf{label}(u) > 0$ **do**
4: $G \leftarrow$ *PN-path-R* (G, l); {*PN-path-R* is the same as *PN-path-D* (Algorithm 6) except that in Algorithm 6, Line 7 is changed to be "reverse all edges in pn-paths (u, v) in G, both weights and directions"}
5: $l \leftarrow l + 1$;
6: **end while**
7: Remove edges (v_i, v_j) from G if $w(v_i, v_j) = +1$;
8: **return** G;

Furthermore, in *PN-path-D*, every vertex u with $\mathsf{label}(u) = 0$ in G'_{l-1} keeps $\mathsf{label}(u) = 0$ in G'_l. If there is a pn-path (u, v) of length $\leq l - 1$ found in G'_l, then it must be in G'_{l-1}, which contradicts the assumption. Therefore, G'_l does not include any pn-paths of length $\leq l$. □

Theorem 5.2: *The PN-path-D algorithm correctly identifies a subgraph G_l which contains all* pn-path*s of length l and returns a graph includes no* pn-path*s of length $\leq l$.*

Proof Sketch: It can be proved by Lemma 5.2 and Lemma 5.4.

We discuss the time complexity of the *Greedy-D* algorithm. Here, the number of iterations calling *PN-path-D* is equivalent to the maximum length l_{max} in the *Greedy-D* algorithm, which is closely related to the diameter of graph G. With a loose bound, it is $O(\log n)$. But, with the property of small world, $_{max}$ is very small. In our experiments, l_{max} is much less than 100, with the max value of 275. Here, we treat l_{max} as a constant. The time complexity of the *Greedy-D* algorithm is $O(n + m)$. Here, both *PN-path-D* and *l-Subgraph* cost $O(n + m)$, because *l-Subgraph* invokes *BFS* twice and *PN-path-D* performs *DFS* once in addition.

5.1.2 The Greedy-R Algorithm

The *Greedy-R* algorithm is shown in Algorithm 8. Like *Greedy-D*, *Greedy-R* will result in an Eulerian subgraph. Unlike *Greedy-D*, it reverses the edges on pn-paths of length l from $l = 1$ until there does not exist a vertex u in G with $\mathsf{label}(u) > 0$. Initially, *Greedy-R* assigns every edge, (v_i, v_j), in G with a weight $w(v_i, v_j) = -1$. Then, in the while loop, it calls *PN-path-R*. *PN-path-R* is the same as *PN-path-D* (Algorithm 6) except that in Algorithm 6 Line 7 is changed to be "reverse all edges in pn-path (u, v) in G, both weights and directions". As a result, *Greedy-R* identifies an Eulerian subgraph of G, $\widetilde{\mathcal{U}}(G)$. Here, $E(\widetilde{\mathcal{U}}(G))$ contains all edges with a weight $= -1$ and $V(\widetilde{\mathcal{U}}(G))$ contains all the vertices in $E(\widetilde{\mathcal{U}}(G))$. Below, we prove the correctness of *Greedy-R*.

Lemma 5.5: *By PN-path-R, the resulting G does not include any* pn-path*s of length $\leq l$.*

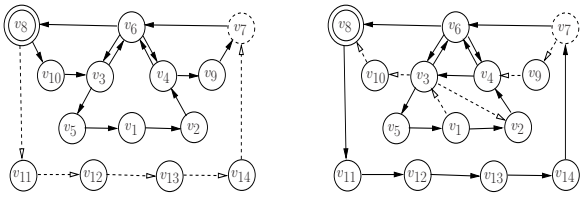
The proof is given in Appendix.

Similar to Theorem. 5.2, *PN-path-R* algorithm correctly identifies a subgraph G_l which contains all pn-paths of length l and returns a graph includes no pn-paths of length $\leq l$.

Theorem 5.3: *The PN-path-R algorithm correctly identifies a subgraph G_l which contains all* pn-path*s of length l and returns a graph includes no* pn-path*s of length $\leq l$.*

We omit the proof of Theorem. 5.3 since it can be proved in a similar manner like Theorem. 5.2 using Lemma 5.5.

It is worth noticing that $\widetilde{\mathcal{U}}(G)$ obtained by *Greedy-R* is at least as good as that obtained by *Greedy-D*. If each edge in $G - \widetilde{\mathcal{U}}(G)$



(a) G' returned by *Greedy-D* (b) G returned by *Greedy-R*

Figure 10: $\widetilde{\mathcal{U}}(G)$ returned by *Greedy-D* and *Greedy-R*

is reversed once, then the $\widetilde{\mathcal{U}}(G)$ obtained by *Greedy-R* is equivalent to that obtained by *Greedy-D*, as each edge appears in at most one pn-path. On the other hand, if there are some edges being reversed more than once, *Greedy-R* performs better. Fig. 10 shows the difference between *Greedy-D* and *Greedy-R*. Since pn-paths of length 1 and 2 are the same, we only show the last deleted/reversed pn-path. In Fig. 10(a), we delete pn-path $(v_8, v_7) = (v_8, v_{11}, v_{12}, v_{13}, v_{14}, v_7)$. On the other hand, in Fig. 10(b), we reverse pn-path $(v_8, v_7) = (v_8, v_{10}, v_3, v_4, v_9, v_7)$. Here edge (v_4, v_3) is reversed twice. $\widetilde{\mathcal{U}}(G)$ returned by *Greedy-R* consists of solid lines, which is better than that returned by *Greedy-D*.

5.2 The Refine Algorithm

Subgraph $\widetilde{\mathcal{U}}(G)$ found by *Greedy* (step 3 in Algorithm 4) is Eulerian since $d_I(u) = d_O(u)$ satisfies for each vertex u in $\widetilde{\mathcal{U}}(G)$. Meanwhile, $\widetilde{G}_D = G - \widetilde{\mathcal{U}}(G)$, the subgraph consist of edges deleted/reversed, is near-acyclic, thus move all cycles from $\widetilde{G}_D$ to $\widetilde{\mathcal{U}}(G)$ (step 4 in Algorithm 4) utilizing *DS-U* is efficient. Such precessing is necessary since $\widetilde{G}_D$ needs to be acyclic to simplify the discussion in Section 5.3 (ensure all cycles found in $\mathcal{G}$ are k-cycles). Besides, $\widetilde{G}_D$ can also be taken as an approximate hierarchy structure for massive graphs.

With the greedy Eulerian subgraph $\widetilde{\mathcal{U}}(G)$ found , we have insight on G because we know $G = \widetilde{\mathcal{U}}(G) \cup \widetilde{G}_D$ where $\widetilde{G}_D$ is a DAG (acyclic), and can design a *Refine* algorithm based on such insight, to reduce the number of times to update $dst(u)$, which reduces the cost of relaxing. The *Refine* algorithm (Algorithm 9) is designed based on the similar idea given in *DS-U* using *FindNC* with two following enhancements.

First, we utilize $G = \widetilde{\mathcal{U}}(G) \cup \widetilde{G}_D$ to initialize the edge weight $w(v_i, v_j)$ for every edge (v_i, v_j) and $dst(u)$ for every vertex u in G. The edge weights are initialized in Line 1-7 in Algorithm 9 based on $\widetilde{\mathcal{U}}(G)$ which is a greedy Eulerian subgraph. We also make use of $\widetilde{G}_D$ to initialize $dst(u)$ based on Eq. (1) in Line 8.

$$dst(u) = \begin{cases} 0 & \text{if } d_I(u) \text{ in } \widetilde{G}_D \text{ is 0,} \\ \min\{dst(v) - 1 | (v,u) \in \widetilde{G}_D\} & u \in \widetilde{G}_D, \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Some comments on the initialization are made below. Following Algorithm 2, $dst(u)$ can be initialized as $dst(u) = 0$. It can be proven that no matter what $dst(v_i)$ is for a vertex v_i $(1 \le i \le k-1)$ in a negative cycle $C = (v_1, v_2, \ldots, v_k = v_1)$, the negative cycle can be identified because there is at least one edge (v_i, v_{i+1}) that can be relaxed. Based on it, if we initialize $dst(u)$ in a way such that $dst(u) \le dst(v) + w(v, u)$, then u cannot be relaxed through (v, u) before updating $dst(v)$. It reduces the number of times to update $dst(u)$, and improves the efficiency. We explain it further. Because for any edge, $(v, u) \in \widetilde{G}_D$, u can never be relaxed through edge (v, u) before $dst(v)$ being updated, *FindNC* (G, u) will relax edges along a path with a few branches to identify a negative-cycle. The variables such as $relax(u)$ and $pos(u)$ are initialized in Line 9 as done in Algorithm 2.

Algorithm 9 *Refine* $(\widetilde{\mathcal{U}}(G), G)$

Input: A graph G, and the Eulerian subgraph obtained by *Greedy*, $\widetilde{\mathcal{U}}(G)$
Output: Two subgraphs of G, $\mathcal{U}(G)$ and G_D $(G = \mathcal{U}(G) \cup G_D)$

1: **for** each edge (v_i, v_j) in $E(G)$ **do**
2: **if** $(v_i, v_j) \in \widetilde{\mathcal{U}}(G)$ **then**
3: reverse the edge to be (v_j, v_i) in G; $w(v_j, v_i) \leftarrow +1$;
4: **else**
5: $w(v_i, v_j) \leftarrow -1$;
6: **end if**
7: **end for**
8: Assign $dst(u)$ for every $u \in V(G)$ based on Eq. (1);
9: **for** each vertex u in $V(G)$ **do** $relax(u) \leftarrow true$, $pos(u) \leftarrow 0$;
10: Enqueue every vertex u in $V(G)$ into a queue $\mathcal{Q}$;
11: $u \leftarrow \mathcal{Q}$.front();
12: **while** $\mathcal{Q} \neq \emptyset$ **do**
13: $S_V \leftarrow \emptyset$, $S_E \leftarrow \emptyset$, $NV \leftarrow \emptyset$;
14: **if** $relax(u) = true$ and *FindNC* (G, u) **then**
15: Reverse negative cycle and change the edge weights using S_V and S_E (refer to Algorithm 2);
16: $\mathcal{Q} \leftarrow \mathcal{Q} \cup S_V$;
17: **else**
18: $\mathcal{Q}$.pop(); $u \leftarrow \mathcal{Q}$.front();
19: **end if**
20: **end while**
21: G_D is a subgraph that contains all edges with a weight of -1;
22: $\mathcal{U}(G)$ is a subgraph that contains the edges reversed for all edges with a weight of +1;

Second, we use a queue $\mathcal{Q}$ to maintain candidate vertices, u, from which there may exist negative-cycles, if $relax(u) = true$. Initially, all vertices are enqueued into $\mathcal{Q}$. In each iteration, when invoking *FindNC* (G, v), let V' be the set of vertices relaxed. Among V', for any vertex $w \in S_V \setminus \{v\}$, $dst(w)$ has been updated and it has only relaxed partial out-neighbors when finding the negative cycle. On the other hand, for any vertex $w \in V' \setminus S_V$, all of the out-neighbors of w have been relaxed and cannot be relaxed before updating $dst(w)$. We exclude $w \in V' \setminus S_V$ from $\mathcal{Q}$ implicitly by setting $relax(w) = false$ in *FindNC* (G, w).

Example 5.3: Suppose we have a greedy Eulerian subgraph $\widetilde{\mathcal{U}}(G)$ (Fig. 7) of G (Fig. 1) by *Greedy-D*, and will refine it to the optimal $\mathcal{U}(G)$ using *Refine*. Initially, all edges (solid lines) in $\widetilde{\mathcal{U}}(G)$ are reversed with initial +1 edge weight, and all remaining edges in $\widetilde{G}_D$ are initialized with -1 edge weight. $dst(v_1) = -2, dst(v_3) = -1, dst(v_7) = -5, dst(v_{11}) = -1, dst(v_{12}) = -2, dst(v_{13}) = -3, dst(v_{14}) = -4$, and other vertices u have $dst(u) = 0$. In the while loop, *FindNC* (G, v_1) relaxes $dst(v_5) = -1$ and returns false. This makes $relax(v_1) = relax(v_5) = false$ by which v_1 and v_5 are dequeued from $\mathcal{Q}$. Afterwards, none of v_2, v_3, v_4, v_6 can relax any out-neighbors, and all are dequeued from $\mathcal{Q}$. *FindNC* (G, v_7) relaxes all vertices, finds a negative cycle $(v_7, v_9, v_4, v_2, v_3, v_{10}, v_8, v_{11}, v_{12}, v_{13}, v_{14}, v_7)$, and adds v_2, v_3, v_4 into $\mathcal{Q}$ as new candidates. Then, no vertices from v_8 to v_{14} can relax any out-neighbors until *FindNC* (G, v_2) finds the last negative cycle (v_2, v_4, v_3, v_2). For most cases, *FindNC* (G, u) relaxes a few of u's out-neighbors.

We discuss the time complexity of *Refine*. The initialization (Lines 1–9) is $O(n + m)$. Since $\widetilde{\mathcal{U}}(G)$ approximates $\mathcal{U}(G)$, the number of negative-cycles found by *Refine* will be no more than $|E(\mathcal{U}(G))| - |E(\widetilde{\mathcal{U}}(G))|$, and vertices u will have $dst(u)$ updated less than $|E(\mathcal{U}(G))| - |E(\widetilde{\mathcal{U}}(G))|$ times. This implies the while loop costs $O(|E(\mathcal{U}(G))| - |E(\widetilde{\mathcal{U}}(G))| \cdot m)$. Time complexity of *Refine* is $O(cm^2)$, where $c \ll 1$, as confirmed in our testing.

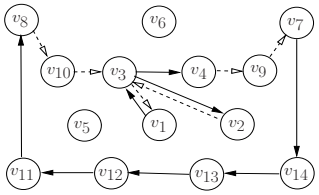


Figure 11: $\mathcal{G} = \overline{G_P} \oplus G_N$ **where** $G_P = G \ominus \widetilde{\mathcal{U}}(G)$ **and** $G_N = G \ominus \mathcal{U}(G)$**.**

5.3 The Bound between Greedy and Optimal

We discuss the bound between $\widetilde{\mathcal{U}}(G)$ obtained by *Greedy* and the maximum Eulerian subgraph $\mathcal{U}(G)$. To simplify our discussion, below, a graph G is a graph with multiple edges between two vertices but without self loops, and every edge (v_i, v_j) is associated with a weight $w(v_i, v_j)$, which is initialized to be -1. Given a graph G, we use $\overline{G}$ to represent the reversed graph of G such that $V(\overline{G}) = V(G)$ and $E(\overline{G})$ contains every edge (v_j, v_i) if $(v_i, v_j) \in E(G)$, and $w(v_j, v_i) = -w(v_i, v_j)$. In addition, we use two operations, $\oplus$ and $\ominus$, for two graphs G_i and G_j. Here, $G_{ij} = G_i \oplus G_j$ is an operation that constructs a new graph G_{ij} by union of two graphs, G_i and G_j, such that $V(G_{ij}) = V(G_i) \cup V(G_j)$, and $E(G_{ij}) = E(G_i) \cup E(G_j)$. And $G' = G_i \ominus G_j$ is an operation that constructs a new graph G' by removing a subgraph G_j from G_i ($G_j \subseteq G_i$) such that $V(G') = V(G_i)$ and $E(G') = E(G_i) \setminus E(G_j)$. Given two Eulerian subgraphs, G_i and G_j, it is easy to show that $G_i \oplus G_j$ and $G_i \ominus G_j$ are still Eulerian graphs. Given any graph G, $G \oplus \overline{G}$ is an Eulerian graph. Note that assume that there is a cycle with two edges, (v_i, v_j) and (v_j, v_i), between two vertices, v_i and v_j, in G. there will be four edges in $G \oplus \overline{G}$, i.e., two edges are from G and two corresponding reversed edges from $\overline{G}$.

We discuss the bound using an Eulerian graph $\mathcal{G} = \overline{G_P} \oplus G_N$, where $G_P = G \ominus \widetilde{\mathcal{U}}(G)$ and $G_N = G \ominus \mathcal{U}(G)$. We call every edge in G_N a negative edge (n-edge), and a path in G_N a negative path (n-path). We also call every edge in $\overline{G_P}$ a positive edge (p-edge), and a path in $\overline{G_P}$ a positive path (p-path). It is important to note that p-edges are given for $\overline{G_P}$ but not for G_P, all n-edges are with a weight of -1, while all p-edges are with a weight of +1, because they are the reversed edges in G_P. Here, $\mathcal{G}$ is a graph with multiple edges between a pair of vertices.

Example 5.4: In Fig. 11, the solid edges represent the p-edges from $\overline{G_P}$, and the dashed edges represent the n-edges from G_N.

Since $\mathcal{G}$ is Eulerian, it can be divided into several edge disjoint simple cycles as given by Lemma 5.1. Among these cycles, there are no cycles in $\mathcal{G}$ with only n-edges, because they must be in $\mathcal{U}(G)$ if they exist. And there are no cycles in $\mathcal{G}$ with only p-edges, because all such cycles have been moved into $\widetilde{\mathcal{U}}(G_i)$ in *GR-U* (Algorithm 4, Line 4).

Next, let a cycle be a positive-cycle if the total weight of the edges in this cycle > 0, and let it be a negative-cycle if its total weight of edges < 0. We show there are no negative-cycles in $\mathcal{G}$.

Lemma 5.6: *There does not exist a* negative-cycle *in* $\mathcal{G}$.

Proof Sketch: Assume there is a negative-cycle in $\mathcal{G}$, denoted as G_{cyc}. Since there are no cycle with only p-edges or n-edges, there are p-edges and n-edges in G_{cyc}. We divide G_{cyc} into two subgraphs, G_p and G_n. Here G_p consists of all p-edges, where each p-edge is with a +1 weight, and G_n consists of all n-edges, where each n-edge is with a -1 weight. Clearly, $|E(G_p)| < |E(G_n)|$, since it assumes that G_{cyc} is a negative-cycle. Note that $\mathcal{U}(G) \ominus G_p \oplus G_n$, which is equivalent to $\mathcal{U}(G) \oplus G_{cyc} \ominus (G_p \oplus G_p)$, is Eulerian, and it contains more edges than $\mathcal{U}(G)$, resulting in a contradiction. Therefore, there does not exist a negative-cycle in $\mathcal{G}$. □

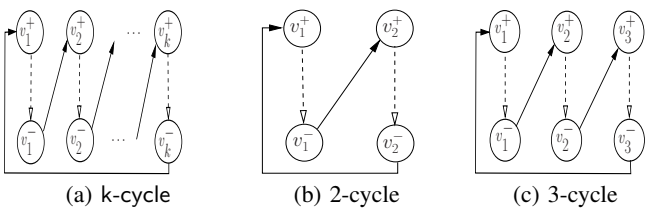


Figure 12: k-cycle

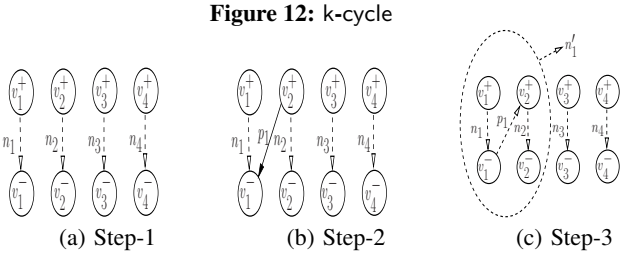


Figure 13: k-cycle **generated by** *Greedy-R*

Lemma 5.6 shows all cycles in $\mathcal{G}$ are non-negative. Since there are no cycles with only p-edges or n-edges, each cycle in $\mathcal{G}$ can be partitioned into an alternating sequence of k p-paths and k n-paths, and represented as $(v_1^+, v_1^-, v_2^+, \ldots, v_k^+, v_k^-, v_1^+)$, where (v_i^+, v_i^-), for $i = 1, 2, \ldots, k$, are n-paths, and (v_i^-, v_{i+1}^+), for $i = 2, \ldots, k-1, k$, plus (v_k^-, v_1^+) are p-paths. We call such cycle a k-cycle. Fig. 12(a) shows an example of k-cycle, and an arrow presents a path. p-paths are in solid lines while n-paths are in dashed lines.

The difference $|E(\mathcal{U}(G))| - |E(\widetilde{\mathcal{U}}(G))|$ is equal to $|E(G \ominus \widetilde{\mathcal{U}}(G))| - |E(G \ominus \mathcal{U}(G))| = |E(G_P)| - |E(G_N)| = |E(\overline{G_P})| - |E(G_N)|$, becomes the total number of edges in G_P minus the total number of edges in G_N. On the other hand, the difference $|E(\mathcal{U}(G))| - |E(\widetilde{\mathcal{U}}(G))|$ can be considered as the total weight of all k-cycles in $\mathcal{G}$. Recall that all edges in G are with weight -1 and the edges in $\overline{G}$ are with weight +1 by our definition. Assume that $\mathcal{G} = \{C_1, C_2, \cdots\}$, where C_i is a k-cycle. The total weight of $\mathcal{G}$ regarding all k-cycles is $w(\mathcal{G}) = \sum_i w(C_i)$. Below, we bound $|E(\mathcal{U}(G))| - |E(\widetilde{\mathcal{U}}(G))|$ using k-cycles.

Consider $\mathcal{G}$ in Fig. 11, there are 3 k-cycles. $C_1 = (v_3, v_1, v_3)$ and $C_2 = (v_3, v_2, v_3)$ with weight 0, and $C_3 = (v_8, v_{11}, v_{12}, v_{13}, v_{14}, v_7, v_9, v_4, v_3, v_{10}, v_8)$ with weight 2. This means that it needs at most 2 more iterations to get the maximum Eulerian subgraph from the greedy solution.

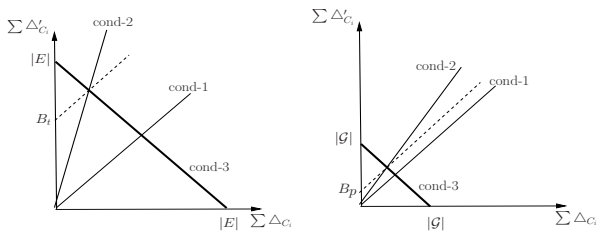
For a k-cycle $(v_1^+, v_1^-, v_2^+, \ldots, v_k^+, v_k^-, v_1^+)$, we use $\triangle_k$ and $\triangle'_k$ to represent the total weight of n-edges[1] and p-edges, i.e. $\triangle_k = \sum_{i=1,\ldots,k} w(v_i^+, v_i^-)$ and $\triangle'_k = \sum_{i=1,\ldots,k-1} w(v_i^-, v_{i+1}^+) + w(v_k^-, v_1^+)$. Because $\triangle_k$ is determined by the optimal in $|E(G_N)| = |E(G \ominus \mathcal{U}(G))|$, the bound is obtained when getting the maximum of $\triangle'_k$.

Theorem 5.4: *The upper bound of the total weight of* p-edge*s in a* k-cycle *with specific* k *is* k *times that of* n-edge*s, i.e.,* $\triangle'_k \leq k \cdot \triangle_k$

The proof is given in Appendix.

Let $\triangle'_{C_i}$ and $\triangle_{C_i}$ denote the total weight of p-edges and n-edges in a k-cycle C_i. Bounding $|E(\mathcal{U}(G))|$- $|E(\widetilde{\mathcal{U}}(G))|$ can be

[1]For n-edges, we take the absolute value of total weight.



(a) Theoretical Upper Bound (b) Empirical Upper Bound

Figure 14: Upper Bounds

formulated as an LP (linear programming) problem.

$$\max \quad \sum_{C_i}(\triangle'_{C_i} - \triangle_{C_i})$$

$$s.t. \quad \text{(Cond-1) } \triangle'_{C_i} > \triangle_{C_i}, \quad \forall i,$$

$$\text{(Cond-2) } \triangle'_{C_i} \leq k_i \cdot \triangle_{C_i}, \text{for k-cycle } C_i \text{ with } k\text{-value } k_i,$$

$$\text{(Cond-3) } \sum_{C_i}(\triangle'_{C_i} + \triangle_{C_i}) \leq |E(\mathcal{G})| \leq |E|$$

In Fig. 14(a), B_t at y-axis illustrates the theoretical upper bound of $|E(\mathcal{U}(G))| - |E(\widetilde{\mathcal{U}}(G))| = \frac{K-1}{K+1}|E|$ by solving the LP problem, where the three solid lines represent the three conditions in the above LP problem, respectively. Here, K is the maximum among all k values. The theoretical upper bound is far from tight. First, $|E(\mathcal{G})| \ll |E|$, which is a tighter upper bound of $\sum_{C_i}(\triangle'_{C_i} + \triangle_{C_i})$, moving Cond-3 towards the origin. Second, for most k-cycles, $\triangle'_k = (1+\epsilon)\cdot\triangle_k$, $0 < \epsilon < 1$, since most p-paths in a k-cycle are far from the upper bound it can get. This leads Cond-2 moving towards x-axis. Therefore, a tighter empirical upper bound is B_p at y-axis in Fig. 14(b). We will show it in the experiments.

We have proved Theorem 5.4 for the case p-paths and n-paths are pn-paths, which shows that each p-path in a k-cycle has an implicit upper bound. In general, there are a small number of cases where p-paths are not pn-paths. For the cases when a p-path in a k-cycle is not a pn-path, we use w_p and w_u to denote its practical weight and the theoretical upper bound it can reach when itself is a pn-path, respectively. Since we concentrate on weight of p-paths, we treat such a p-path as a pn-path with weight w_p if $w_p < w_u$, and treat it as a pn-path with weight w_u if $w_p > w_u$ and add the difference $w_p - w_u$ to a global variable W. We will show in Section 6 that W is very small compared with $|E(\mathcal{U}(G))|$.

Time complexity: Revisit *GR-U* (Algorithm 4), it includes four parts: SCC decomposition (Line 1), *Greedy* (Line 3), cycle moving (Line 4) and *Refine* (Line 5). SCC decomposition can be accomplished in 2 *DFS*, in time $O(n+m)$. As analyzed in Section 5, *Greedy* invokes l_{max} times *PN-path*, and each *PN-path* needs 2 *BF-S* (*l-Subgraph*) and 1 *DFS* (remove/reverse pn-paths). Since l_{max} is small (< 100 in our extensive experiments), the time complexity of *Greedy* is $O(n+m)$. Regarding moving cycles from $G_i - \widetilde{\mathcal{U}}(G_i)$ to $\widetilde{\mathcal{U}}(G_i)$, it is equivalent to moving cycles from non-trivial SCCs of $G_i - \widetilde{\mathcal{U}}(G_i)$ to $\widetilde{\mathcal{U}}(G_i)$. Based on the fact that $G_i - \widetilde{\mathcal{U}}(G_i)$ is near acyclic, there are a few cycles in $G_i - \widetilde{\mathcal{U}}(G_i)$, cycle moving is in $O(n+m)$. The time complexity of *Refine*, as given in Section 5.2 is $O(cm^2)$, because most *FindNC* (G,u) relax edges along a path with a few branches and vertices u will have $dst(u)$ updated less than $|E(\mathcal{U}(G))| - |E(\widetilde{\mathcal{U}}(G))|$ times.

6. PERFORMANCE STUDIES

We conduct extensive experiments to evaluate two proposed *GR-U* algorithms. One is *GR-U-D* using *Greedy-D* (Algorithm 5) and *Refine* (Algorithm 9), and the other is *GR-U-R* using *Greedy-R* (Algorithm 8) and *Refine* (Algorithm 9). We do not compare our algorithms with *BF-U* in [18], because *BF-U* is in $O(nm^2)$ and is too slow. We use our *DS-U* as the baseline algorithm, which is $O(m^2)$. We show that *Greedy* produces an answer which is very close the the exact answer. In order to confirm *Greedy* is of time complexity $O(n+m)$, we show the largest iteration l_{max} used in *Greedy* is a small constant by showing that the longest pn-path (the same as l_{max}) deleted/reversed by *Greedy* is small. In addition, we confirm the constant c of $O(c \cdot m^2)$ for *Refine* is very small by showing statistics of $\mathcal{G}$, W, and k-cycles. We also confirm the scalability of *GR-U* as well as *Greedy* and *Refine*.

All these algorithms are implemented in C++ and complied by gcc 4.8.2, and tested on machine with 3.40GHz Intel Core i7-4770 CPU, 32GB RAM and running Linux. The time unit used is second.

Graph	$\|V\|$	$\|E\|$	$\|V(\mathcal{U}(G))\|$	$\|E(\mathcal{U}(G))\|$	$\|V(G_D)\|$
wiki-Vote	7,115	103,689	1,286	17,676	7,114
Gnutella	62,586	147,892	11,952	18,964	62,519
Epinions	75,879	508,837	33,673	264,995	67,803
Slashdot0811	77,360	828,159	70,849	734,021	33,682
Slashdot0902	82,168	870,159	71,833	748,580	44,611
web-NotreDame	325,729	1,469,679	99,120	783,788	318,964
web-Stanford	281,903	2,312,497	211,883	691,521	252,983
amazon	403,394	3,387,388	399,702	1,973,965	372,309
Wiki-Talk	2,394,385	5,021,410	112,030	1,083,509	2,368,590
web-Google	875,713	5,105,039	461,381	1,841,215	837,275
web-BerkStan	685,230	7,600,595	478,774	2,068,081	620,881
Youtube	1,138,499	4,945,382	534,668	3,954,923	1,016,342
Flickr	1,715,255	22,613,980	1,401,648	15,882,577	919,309
Pokec	1,632,803	30,622,560	1,297,362	20,911,934	1,562,696

Table 2: Summary of real Datasets

Graph	*Refine*	*GR-U-D*	*Refine*	*GR-U-R*	*DS-U*	c
wiki-Vote	0.1	0.1	0.1	0.1	1.0	0.100
Gnutella	0.5	0.5	0.4	0.4	1.6	0.250
Epinions	15.9	16.1	15.2	15.4	414.4	0.037
Slashdot0811	80.6	80.8	70.9	71.0	12,748.6	0.006
Slashdot0902	87.3	87.5	76.6	76.8	14,324.5	0.005
web-NotreDame	2.6	3.0	2.4	2.7	370.4	0.007
web-Stanford	21.5	25.7	16.7	24.9	2,780.0	0.009
amazon	126.5	133.5	124.8	130.5	44,865.0	0.003
Wiki-Talk	504.3	504.9	487.3	487.9	9,120.1	0.053
web-Google	100.2	110.3	78.6	84.6	35,271.7	0.002
web-BerkStan	129.7	137.7	67.8	75.9	7,853.9	0.010
Youtube	4,617.1	4,620.3	4,073.4	4,076.1	-	-
Flickr	76,984.3	76,996.4	66,875.0	66,888.1	-	-
Pokec	30,954.5	30,983.7	30,120.4	30,140.5	-	-
Gplus2	363.5	364.2	360.5	361.2	39,083.8	0.009
Weibo0	206.5	207.3	202.4	203.3	8,004.6	0.025

Table 3: Efficiency of *GR-U-D*, *GR-U-R* and *DS-U*

Datasets: We use 16 real datasets. Among the datasets, wiki-Vote, Epinions, Slashdot0811, Slashdot0902, Pokec, Google+, Weibo, Youtube and Flickr are social networks; web-NotreDame, web-Stanford, web-Google, and web-BerkStan are web graphs; Gnutella is a peer-to-peer network; amazon is a product co-purchasing network; and Wiki-Talk is a communication network. All the datasets are downloaded from Stanford large network dataset collection (`http://snap.stanford.edu/data`) except for Google+, Weibo, Youtube and Flickr. Youtube and Flickr are from [29]. The detailed information of the datasets are summarized in Table 1 and Table 2. In the tables, for each graph, the 2nd and 3rd columns show the numbers of vertices and edges[2], respectively, and the 4th and 5th columns show the numbers of vertices and edges of its maximum Eulerian subgraph, respectively, and the 6th column shows the number of vertices in G_D.

Efficiency: Table 3 shows the efficiency of these three algorithms, i.e., *GR-U-D*, *GR-U-R*, and *DS-U*, over 16 real datasets. For

[2]for each dataset, we delete all self-loops if exist.

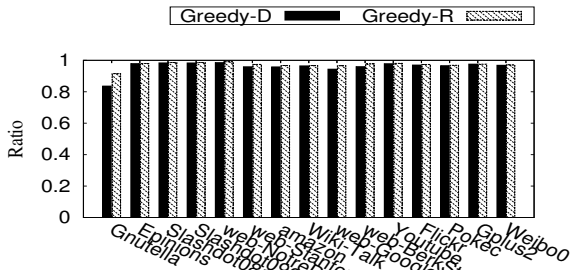


Figure 15: $|E(\widetilde{\mathcal{U}}(G))|/|E(\mathcal{U}(G))|$

Graph	IRD	ISD %	IRR	ISR %	IR_DSU
wiki-Vote	659	95.4	629	95.6	14,361
Gnutella	2,504	69.5	1,410	82.8	8,202
Epinions	5,466	97.4	5,334	97.4	207,124
Slashdot0811	11,464	97.9	9,990	98.2	541,970
Slashdot0902	12,036	97.8	10,426	98.1	554,163
web-NotreDame	9,030	98.1	6,119	98.7	486,240
web-Stanford	23,427	94.8	15,721	96.5	448,960
amazon	75,104	94.1	61,818	95.2	1,282,326
Wiki-Talk	37,662	95.7	36,139	95.9	871,020
web-Google	90,375	92.4	59,387	95.0	1,196,616
web-BerkStan	69,078	95.2	41,703	97.1	1,437,188
Youtube	79,798	-	71,768	-	-
Flickr	467,557	-	409533	-	-
Pokec	686,765	-	635,286	-	-
Gplus2	18,766	96.9	18,721	96.9	613,008
Weibo0	25,991	96.2	24,550	96.4	686,765

Table 4: The numbers of Iterations

GR-U-D, the 2nd column shows the running time of *Refine* and the 3rd column shows the total running time of *GR-U-D*. As can be seen, for *GR-U-D*, the running time of *Refine* dominates that of *Greedy-D*. The 4th and 5th columns show the running time of *Refine* and the total running time of *GR-U-R*, respectively. Likewise, the *Refine* algorithm is the most time-consuming procedure in *GR-U-R*. It is important to note that both *GR-U-D* and *GR-U-R* significantly outperform *DS-U*. In most large datasets, *GR-U-D* and *GR-U-R* are two orders of magnitude faster than *DS-U*. For instance, in web-Stanford dataset, *GR-U-R* takes 25 seconds to find the maximum Eulerian subgraph, while *DS-U* takes 2,780 seconds, which is more than 100 times slower. In addition, it is worth mentioning that in YouTube, Flickr and Pokec dataset, *DS-U* cannot get a solution in 24 hours. In the 6th column, c is the c value in *Refine*'s time complexity $O(cm^2)$, by comparing running time of *GR-U-R* and *DS-U*. In all graphs, $c \ll 1$. Note *BF-U* is very slow, for example, *BF-U* takes more than 30,000 seconds to handle the smallest dataset wiki-Vote, while our *GR-U* takes only 0.1 second.

Effectiveness of *Greedy*: To evaluate the effectiveness of the greedy algorithms, we first study the size of Eulerian subgraph obtained by *Greedy-D* and *Greedy-R*. Fig. 15 depicts the results. In Fig. 15, $|E(\widetilde{\mathcal{U}}(G))|$ denotes the size of Eulerian subgraph obtained by the greedy algorithms, $|E(\mathcal{U}(G))|$ denotes the size of the maximum Eulerian subgraph, and $|E(\widetilde{\mathcal{U}}(G))|/|E(\mathcal{U}(G))|$ denotes the ratio between them. The ratios obtained by both *Greedy-D* and *Greedy-R* are very close to 1 in most datasets. That is to say, both *Greedy-D* and *Greedy-R* can get a near-maximum Eulerian subgraph, indicating that both *Greedy-D* and *Greedy-R* are very effective. The performance of *Greedy-R* is slightly better than that of *Greedy-D*, which supports our analysis. The ratio of Gnutella dataset using *Greedy-D* is slightly lower than others. One possible reason is that Gnutella is much sparser than other datasets, thus some inappropriate pn-path deletions may result in enlarging other pn-paths, and this situation can be largely relieved in *Greedy-R*.

Graph	$\lvert E(\mathcal{U}(G))\rvert$	$\lvert E(\mathcal{G})\rvert$	W
wiki-Vote	17,676	3,214	20
Gnutella	18,964	6,906	10
Epinions	264,995	30,997	129
Slashdot0811	734,021	45,315	118
Slashdot0902	748,580	46,830	145
web-NotreDame	783,788	10,439	3,963
web-Stanford	691,521	35,402	6,168
amazon	1,973,965	202,513	12,994
Wiki-Talk	1,083,509	158,848	331
web-Google	1,841,215	149,425	22,361
web-BerkStan	2,068,081	105,569	16,991
Youtube	3,954,923	288,798	17,947
Flickr	15,882,577	1,758,090	15,384
Pokec	20,911,934	3,003,797	8,964
Gplus2	770,854	117,641	80
weibo0	850,136	124,395	384

Table 5: Statistics of $|\mathcal{G}|$ and W

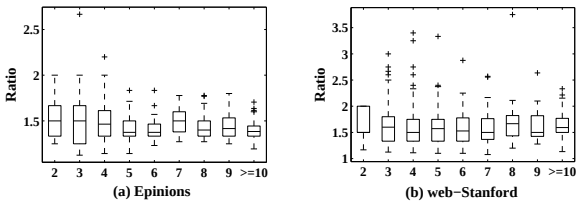
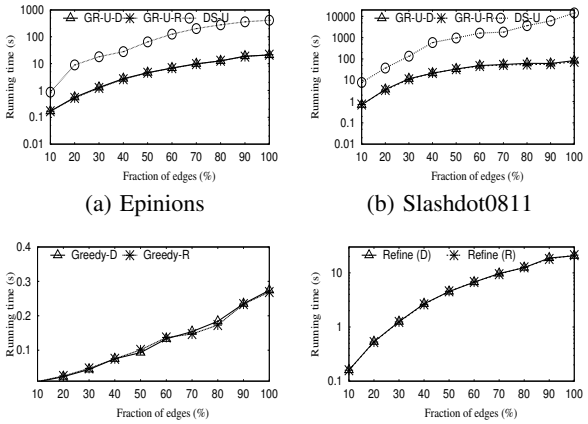


Figure 16: Distributions of k-cycles for each k

Second, we investigate the numbers of iterations used in *GR-U-D*, *GR-U-R*, and *DS-U*. Table 4 reports the results. In Table 4, the 2nd and 4th columns 'IRD' and 'IRR' denote the numbers of iterations used in the refinement procedure (i.e., *Refine*, Algorithm 9) of *GR-U-D* and *GR-U-R*, respectively. The last column 'IR_DSU' reports the total number of iterations used in *DS-U*. From these columns, we can see that in large graphs (e.g., web-NotreDame dataset), the numbers of iterations used in *Refine* of *GR-U-D* and *GR-U-R* are at least two orders of magnitude smaller than those used in *DS-U*. In addition, it is worth mentioning that in Pokec dataset, *DS-U* cannot get a solution in two weeks. The 3rd and 5th columns report the percentages of iterations saved by *GR-U-D* and *GR-U-R*, respectively. Both *Greedy-D* and *Greedy-R* can reduce at least 95% iterations in most datasets. Similarly, the results obtained by *GR-U-R* are slightly better than those obtained by *GR-U-D*.

The largest iteration l_{max}: We show the largest iteration l_{max} in *Greedy* by showing the longest pn-paths deleted/reversed, which is the numbers of *PN-path-D*/*PN-path-R* invoked by *Greedy-D*/*Greedy-R* using the real datasets. Below, the first/second number is the longest pn-paths deleted/reversed. wiki-Vote (9/9), Gnutella (29/22), Epinions (12/10), Slashdot0811 (6/6), Slashdot0902 (8/8), web-NotreDame (96/41), web-Stanford (275/221), amazon (57/37), Wiki-Talk (9/7), web-Google (93/37), web-BerkStan (123/85), Youtube (12/12), Flickr (17/17), Pokec (14/13), Gplus2 (9/8), and Weibo0 (12/10). The longest pn-paths deleted or reversed are always of small sizes, especially compared with $|E|$. Therefore, the time complexity of *Greedy* can be regarded as $O(n+m)$.

The support to a small c: We show the support that c given in $O(cm^2)$ for *Refine* is small by giving statistics of $\mathcal{G}$, W, and k-cycles. We first show the statistics of $\mathcal{G}$ $(= \overline{G_P} \oplus G_N)$ and W discussed in Section 5.3. Table 5 reports the results. From Table 5, we can find that for each graph, $|E(\mathcal{G})|$ and W are small compared with $|E(\mathcal{U}(G))|$. These results confirm our theoretical analysis in Section 5.3. Second, we study the statistics of k-cycles. The results of Epinions and web-Stanford datasets are depicted in Fig. 16, and



(a) Epinions (b) Slashdot0811

(c) *Greedy* of (a) (d) *Refine* of (a)

Figure 17: Scalability: Epinions and Slashdot0811

similar results can be observed from other datasets. In Fig. 16, y-axis denotes the ratio between the total weights of p-edges and the total weights of n-edges (i.e., Δ'_k/Δ_k defined in Section 5.3), and the x-axis denotes k for k-cycles, where $k = 2, 3, \cdots, >= 10$. As can be seen, for all k-cycles, the ratios are always smaller than 2 in both Epinions and web-Stanford datasets. These results confirm our analysis in Section 5.3.

Scalability: We test the scalability for *GR-U-R*, *GR-U-D*, and *DS-U*. We report the results for Epinions and Slashdot0811 in Fig. 17. Similar results are observed for other real datasets. To test the scalability, we sample 10 subgraphs starting from 10% of edges, up to 100% by 10% increments. Fig. 17(a) and Fig. 17(b) show both *GR-U-R* and *GR-U-D* scale well. For Epinions, we further show the performance of *Greedy* and *Refine* in Fig. 17(c) and Fig. 17(d). In Fig. 17(c), *Greedy* seems to be not really linear. We explain the reason below. Revisit Algorithm 4, the efficiency of *Greedy* is mainly determined by two factors, the graph size (or more precisely the size of the largest SCC) and the number of times invoking *PN-path* (i.e. l_{max}). When a subgraph is sparse, both SCC size and l_{max} tend to be small (the smallest sample graph with 10% edges contains a largest SCC with 1,155 vertices and 4,317 edges, and $l_{max} = 30/16$ for *Greedy-D*/*Greedy-R*), whereas, both the size of the largest SCC and l_{max} tend to be large in dense subgraphs (the entire graph contains a largest SCC with 53,968 vertices and 296,228 edges, and $l_{max} = 96/41$ for *Greedy-D*/*Greedy-R*).

7. CONCLUSION

In this paper, we study social hierarchy computing to find a social hierarchy G_D as DAG from a social network represented as a directed graph G. To find G_D, we study how to find a maximum Eulerian subgraph $\mathcal{U}(G)$ of G such that $G = \mathcal{U}(G) \cup G_D$. We justify our approach, and give the properties of G_D and the applications. The key is how to compute $\mathcal{U}(G)$. We propose a *DS-U* algorithm to compute $\mathcal{U}(G)$, and develop a novel two-phase Greedy-&-Refine algorithm, which greedily computes an Eulerian subgraph and then refines this greedy solution to find the maximum Eulerian subgraph. The quality of our greedy approach is high which can be used to support social mobility and recover the hidden directions. We conduct extensive experiments to confirm the efficiency of our Greedy-&-Refine approach.

8. REFERENCES

[1] D. Acemoglu, A. Ozdaglar, and A. ParandehGheibi. Spread of (mis) information in social networks. *Games and Economic Behavior*, 70(2), 2010.

[2] C. C. Aggarwal, editor. *Social Network Data Analytics*. Springer, 2011.

[3] J. A. Almendral, L. López, and M. A. Sanjuán. Information flow in generalized hierarchical networks. *Physica A: Statistical Mechanics and its Applications*, 324(1), 2003.

[4] B. Ball and M. E. Newman. Friendship networks and social status. *Network Science*, 1(01), 2013.

[5] L. Cai and B. Yang. Parameterized complexity of evenodd subgraph problems. *Journal of Discrete Algorithms*, 9(3), 2011.

[6] P. A. Catlin. Supereulerian graphs: a survey. *Journal of Graph theory*, 16(2), 1992.

[7] W. Chen, L. V. Lakshmanan, and C. Castillo. *Information and Influence Propagation in Social Networks*. Morgan & Claypool, 2013.

[8] Z. Chen and H. Lai. Reduction techniques for supereulerian graphs and related topics: a survey. *Combinatorics and graph theory*, 95, 1995.

[9] A. Clauset, C. Moore, and M. E. Newman. Hierarchical structure and the prediction of missing links in networks. *Nature*, 453(7191), 2008.

[10] Y. Dong, J. Tang, N. V. Chawla, T. Lou, Y. Yang, and B. Wang. Inferring social status and rich club effects in enterprise communication networks. *PLoS ONE*, 10, 2015.

[11] P. Doreian, V. Batagelj, and A. Ferligoj. Symmetric-acyclic decompositions of networks. *Journal of Classification*, 17(1), 2000.

[12] H. Fleischner. *Eulerian graphs and related topics*, volume 1. North Holland, 1990.

[13] H. Fleischner. (some of) the many uses of eulerian graphs in graph theory (plus some applications). *Discrete Mathematics*, 230(1), 2001.

[14] N. Z. Gong, A. Talwalkar, L. Mackey, L. Huang, E. C. R. Shin, E. Stefanov, D. Song, et al. Jointly predicting links and inferring attributes using a social-attribute network (SAN). *ACM Workshop on Social Network Mining and Analysis*, 2011.

[15] N. Z. Gong, W. Xu, L. Huang, P. Mittal, E. Stefanov, V. Sekar, and D. Song. Evolution of social-attribute networks: Measurements, modeling, and implications using google+. *ACM/USENIX Internet Measurement Conference*, 2012.

[16] R. V. Gould. The origins of status hierarchies: A formal theory and empirical test1. *American journal of sociology*, 107(5), 2002.

[17] M. S. Granovetter. The strength of weak ties. *American journal of sociology*, 1973.

[18] M. Gupte, P. Shankar, J. Li, S. Muthukrishnan, and L. Iftode. Finding hierarchy in directed online social networks. In *Proc. of WWW'11*, 2011.

[19] V. Guruswami, R. Manokaran, and P. Raghavendra. Beating the random ordering is hard: Inapproximability of maximum acyclic subgraph. In *Proc. of FOCS'08*, 2008.

[20] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, 1972.

[21] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5), 1999.

[22] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proc. of WWW'10*, 2010.

[23] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *Proc. of KDD'06*, 2006.

[24] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. In *Proc. of WWW'10*, 2010.

[25] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Signed networks in social media. In *Proc. of CHI'10*, 2010.

[26] D. Li, D. Li, and J. Mao. On maximum number of edges in a spanning eulerian subgraph. *Discrete mathematics*, 274(1), 2004.

[27] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7), 2007.

[28] A. S. Maiya and T. Y. Berger-Wolf. Inferring the maximum likelihood hierarchy in social networks. In *Proc. of CSE'09*, 2009.

[29] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the flickr social network. In *Proceedings of the 1st ACM SIGCOMM Workshop on Social Networks (WOSN'08)*, 2008.

[30] A. Newman. *Approximating the maximum acyclic subgraph*. PhD thesis, Massachusetts Institute of Technology, 2000.

[31] H. Nguyen and R. Zheng. Influence spread in large-scale social networks - A belief propagation approach. In *Proc. of ECML/PKDD*, 2012.

[32] R. M. Nosofsky. Attention, similarity, and the identification–categorization relationship. *Journal of experimental psychology: General*, 115(1), 1986.

[33] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2), 1985.

[34] C. Wang, J. Han, Y. Jia, J. Tang, D. Zhang, Y. Yu, and J. Guo. Mining advisor-advisee relationships from research publication networks. In *Proc. of KDD'10*, 2010.

[35] J. Zhang, B. Liu, J. Tang, T. Chen, and J. Li. Social influence locality for modeling retweeting behaviors. In *Proc. of IJCAI'13*, 2013.

[36] J. Zhang, C. Wang, and J. Wang. Who proposed the relationship?: recovering the hidden directions of undirected social networks. In *Proc. of WWW'14*, 2014.

Appendix

Proof Sketch: of Theorem 2.1: Assume the opposite. We can construct an auxiliary graph $G' = G \cup \{(u, v)\}$. Then finding the maximum Eulerian subgraph for G' can be done in two steps. In the first step, find the maximum Eulerian subgraph $\mathcal{U}(G)$, and in the second step, find the maximum Eulerian subgraph for G plus the additional edge (u, v). Since $G = \mathcal{U}_1(G) \cup G_{D_1} = \mathcal{U}_2(G) \cup G_{D_2}$, there are supposed to be at least two corresponding relaxing orders, when the first phase terminates, namely, identifying $\mathcal{U}_1(G)$ and $\mathcal{U}_2(G)$. For one relaxing order, we can show that the added edge (u, v) can be relaxed, which results in finding $\mathcal{U}(G')$ such that $|E(\mathcal{U}(G'))| > |E(\mathcal{U}(G))|$. For the other relaxing order, we can also show that the added edge (u, v) cannot be relaxed and $\mathcal{U}(G') = \mathcal{U}(G)$. It leads to a contradiction, because it can find two different maximum Eulerian subgraphs for G' with different sizes.

Alternatively, let the ranking in G_{D_1} and G_{D_2} be $r_1(\cdot)$ and $r_2(\cdot)$. Assume there are two vertices u and v such that u is *strictly-higher* than v by r_1 whereas v is *strictly-higher* than u by r_2. We prove this cannot achieve based on the finding in [18]. In [18], it gives a total score on G which measures how G is different from DAG G_D based on a ranking $r(\cdot)$. The total score, denoted as $A(G, r)$, is obtained by summing up the weights assigned to edges, $max\{r(u)-r(v)+1, 0\}$ for edge (u, v). The finding in [18] is that the minimum total score equals to the number of edges in the maximum Eulerian subgraph, $min_r\{A(G, r)\} = |E(\mathcal{U}(G))|$. Choose r_1 and r_2 satisfying that $A(G, r_1) = |E(\mathcal{U}_1(G))|$ and $A(G, r_2) = |E(\mathcal{U}_2(G))|$. Since u is *strictly-higher* than v in G_{D_1}, there is a directed path from v to u in G_{D_1}. We can construct an auxiliary graph $G' = G \cup \{(u, v)\}$, then $|E(\mathcal{U}(G'))| > |E(\mathcal{U}_1(G))|$. On the other hand, over the same G', since v is *strictly-higher* than u in G_{D_2}, we can show $|E(\mathcal{U}(G'))| \leq A(G', r_2) = A(G, r_2)$, which leads to a contradiction. □

In order to prove Theorem 4.2, we first prove three lemmas: Lemma 8.1, Lemma 8.2, and Lemma 8.3.

Lemma 8.1: *Given an Eulerian graph G, when DS-U (G) terminates, for each vertex u, $dst(u) \in [-2m, 0]$, where $m = |E(G)|$.*

Proof Sketch: We do mathematical induction on the maximum number of cycles the Eulerian graph G contains.

1. If G contains only one cycle, i.e. G is a simple cycle itself, it is easy to see that for each vertex u, $dst(u) \geq -m \in [-2m, 0]$.
2. Assume Lemma 8.1 holds when G contains no more than k cycles, we prove it also holds when G contains at most $k + 1$ cycles. We first decompose G into a simple cycle C which is the last negative cycle found during *DS-U* (G) and the remaining is an Eulerian graph G' containing at most k cycles. We explain the validation of this decomposition as follows. If the last negative cycle found contains some positive edges, then the resulting maximum Eulerian subgraph $\mathcal{U}(G)$ will contain some negative edges, it is against the fact that G itself is given as Eulerian. Next, we decompose *DS-U* (G) into two phases, it finds G' as an Eulerian subgraph in the first phase while cycle C is identified in the second phase. According to the assumption, when the first phase completes, for each vertex $u \in G', dst(u) \in [-2|E(G')| - |E(C)|, 0]$, where $-2|E(G')|$ is by *DS-U* (G') and $-|E(C)|$ is the result by relaxing C. There are two cases for the second phase.
   (a) If $V(C) \bigcap V(G') = \emptyset$, the two phases are independent. Therefore, when the second phase terminates, for each vertex $u \in V(C)$, $dst(u) \in [-2|E(C)|, 0]$, and for each vertex $u \in V(G')$, $dst(u) \in [-2|E(G')|, 0]$, Lemma 8.1 holds.
   (b) If $V(C) \bigcap V(G') \neq \emptyset$, suppose $w \in V(C) \bigcap V(G')$, then $dst(w) \in [-2|E(G')| - |E(C)|, 0]$ when the first phase completes. During the second phase, $dst(w)$ decreases by $|E(C)|$, then $dst(w) \geq -2|E(G')|-2|E(C)| \in [-2m, 0]$. For any vertex $v \in V(G') \setminus V(C)$, $dst(v)$ can only change along a path $p = (w_0, w_1, \ldots, w_{k-1}, w_k = v)$, where $w_0 \in V(C)$ and $(w_i, w_{i-1}) \in E(G')$, for $i = 1, \ldots, k$. Then $d(v) = d(w_0) + \sum_{i=0}^{k-1} w(w_i, w_{i+1}) > d(w_0) \geq -2m \in [-2m, 0]$. Therefore, Lemma 8.1 holds. □

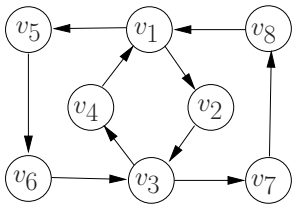


Figure 18: Example graph to explain 2(b) in Lemma 8.1

Example 8.1: We explain the proof of 2(b) in Lemma 8.1 using Fig. 18. Fig. 18 shows an Eulerian graph G containing simple cycles. Suppose the first negative cycle found is $C_1 = (v_1, v_2, v_3, v_4, v_1)$, with the resulting $dst(v_1) = -4, dst(v_2) = -1, dst(v_3) = -2, dst(v_4) = -3$. In a similar way, suppose the second negative cycle found is $C_2 = (v_1, v_5, v_6, v_3, v_2, v_1)$ by relaxing $dst(v_1) = -5, dst(v_5) = -5, dst(v_6) = -6, dst(v_3) = -7, dst(v_2) = -6$, and the third negative cycle is $C_3 = (v_3, v_7, v_8, v_1, v_4, v_3)$ with $dst(v_1) = -10, dst(v_4) = -9, dst(v_3) = -8, dst(v_7) = -8, dst(v_8) = -9$. By reversing these three negative cycles, we have a cycle $C = (v_1, v_2, v_3, v_4)$, and a graph G' which is a simple cycle with $(v_1, v_5, v_6, v_3, v_7, v_8, v_1)$. As can be seen, the current $\min\{dst(u)\} = dst(v_1) = -10$ is in the range of $[-2|E(G')| - |E(C)|, 0]$. When *DS-U* (G) terminates, $\min\{dst(u)|u \in V(C)\} = dst(v_1) = -14$ is in the rage of $[-2|E(G)|, 0]$, and $\min\{dst(u)| u \in V(G')\setminus V(C)\} = dst(v_8) = -13$ is in the range of $[-2|E(G)|, 0]$. It shows that Lemma 8.1 holds for this example.

Used-In	Symbol	Meaning
Greedy	label(u)	label$(u) = d_O(u) - d_I(u)$
	pn-path (u, v)	path$(u = v_1, v_2, \cdots, v_l = v)$, label$(u) > 0$, label$(v) < 0$, and label$(v_i) = 0$ for $1 < i < l$
	G_l	(l-Subgraph) subgraph of G contains all pn-paths of length l
	G^T	$V(G^T) = V(G), E(G^T) = \{(u, v) \mid (v, u) \in E(G)\}$
	level(v)	the shortest distance from any vertex u with a positive label, label$(u) > 0$, in G
	rlevel(v)	the shortest distance to any vertex u with a negative label, label$(u) < 0$, in G
Refine Analysis	$\overline{G}$	$V(\overline{G}) = V(G), \forall(v_i, v_j) \in E(G), (v_j, v_i) \in E(\overline{G})$, and $w(v_j, v_i) = -w(v_i, v_j)$
	p-path/n-path	a path where every edge is with a positive/negative weight
	k-cycle	$(v_1^+, v_1^-, v_2^+, \ldots, v_k^+, v_k^-, v_1^+)$, where (v_i^+, v_i^-) are n-paths, and (v_i^-, v_{i+1}^+) plus (v_k^-, v_1^+) are p-paths
	$\triangle_k$	the total weight of n-edges for a k-cycle ($\triangle_k = \sum_{i=1,\ldots,k} w(v_i^+, v_i^-)$)
	$\triangle'_k$	the total weight of p-edges for a k-cycle ($\triangle'_k = \sum_{i=1,\ldots,k-1} w(v_i^-, v_{i+1}^+) + w(v_k^-, v_1^+)$)
	$\mathcal{G}$	$\mathcal{G} = \overline{G_P} \oplus G_N, G_P = G \ominus \mathcal{U}(G)$ and $G_N = G \ominus \mathcal{U}(G)$

Table 6: Notations

Lemma 8.2: *Given a general graph G, when DS-U(G) terminates, for each vertex u, $dst(u) \in [-4m, 0]$, where $m = |E(G)|$.*

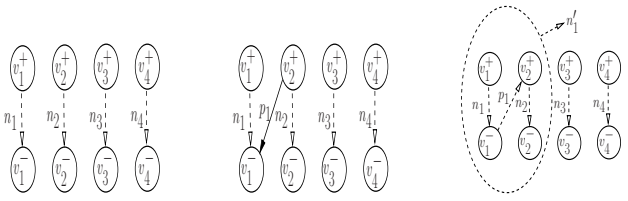
Proof Sketch: For a general graph G, we can add edges (u, v) from vertices u with $d_I(u) > d_O(u)$ to vertices v with $d_I(v) < d_O(v)$ and $(u, v) \notin E(G)$. Obviously, the resulting augment graph G^A has at most $2m$ edges.

Based on Lemma 8.1, when *DS-U* (G^A) terminates, each vertex u satisfies $d(u) \in [-4m, 0]$. On the other hand, *DS-U* (G^A) can be decomposed into two phases, *DS-U* (G) and further relaxations exploiting $E(G^A) \setminus E(G)$, implying that for each vertex u, $dst(u) \in [-4m, 0]$ holds when *DS-U* (G) terminates. □

Lemma 8.3: *For each value of $dst(u)$ of every vertex u, the out-neighbors of u, i.e. $N_O(u)$, are relaxed at most once.*

Proof Sketch: As shown in Algorithm. 3, $dst(u)$ is monotone decreasing, and $pos(u)$ is monotone increasing for a particular $dst(u)$ value. So Lemma 8.3 holds. □

Proof Sketch of Theorem 4.2: Given Lemma 8.1, Lemma 8.2, and Lemma 8.3, since every edge (u, v) is checked at most $|dst(u)| + |dst(v)| \leq 8m$ times for relaxations. By applying amortized analysis [33], the time complexity of *DS-U* (G) is $O(m^2)$. □



(a) Step-1 (b) Step-2 (c) Step-3

Figure 19: k-cycle **generated by** *Greedy-R*

Proof of Theorem 5.4: The proof is based on the way k-cycles constructed by *Greedy-R*. For simplicity, we first assume that each n-path and p-path is a pn-path itself, and we will deal with general cases later. Based on *Greedy-R*, a k-cycle is constructed as shown in Fig. 19, which is a 4-cycle. Initially, there are 4 n-paths, $n_i = (v_i^+, v_i^-)$, $i = 1, 2, \ldots, 4$, as Fig. 19(a) shows. *Greedy-R* deals with pn-paths of length l from a small l to a large l. First, *Greedy-R* finds a path p_1 = pn-path (v_2^+, v_1^-), and combines p_1 with two separated n-paths, n_1 and n_2 into a new n-path n'_1. Here, len(p_1) is no larger than any len(n_i). *Greedy-R* will repeat this process to add all p-paths, p_i into k-cycle in an ascending order of their lengths. The last p-path (v_k^-, v_1^+) to be added to k-cycle should be the longest one among all p-paths. Then its upper bound is $\sum_{i=1,\ldots,k} w(v_i^+, v_i^-) + \sum_{i=1,\ldots,k-1} w(v_i^-, v_{i+1}^+)$. Otherwise, its upper bound should be $max\{w(v_i^-, v_{i+1}^+)\}$. Below, we prove Theorem. 5.4.

- For 2-cycle (Fig. 12(b)): Since $w(v_1^-, v_2^+) \leq w(v_1^+, v_1^-)$, $w(v_1^-, v_2^+) \leq w(v_2^+, v_2^-)$, and $w(v_2^-, v_1^+) \leq w(v_1^+, v_1^-) + w(v_1^-, v_2^+) + w(v_2^+, v_2^-)$ we have,

$$\begin{aligned} \triangle'_2 &= w(v_1^-, v_2^+) + w(v_2^-, v_1^+) \\ &\leq w(v_1^+, v_1^-) + 2 \cdot w(v_1^-, v_2^+) + w(v_2^+, v_2^-) \\ &\leq 2 \cdot \triangle_2 \end{aligned}$$

- For 3-cycle (Fig. 12(c)): Since

$$\begin{aligned} w(v_1^-, v_2^+) &\leq w(v_1^+, v_1^-), w(v_2^+, v_2^-), w(v_3^+, v_3^-) \\ w(v_2^-, v_3^+) &\leq w(v_1^+, v_1^-) + w(v_1^-, v_2^+) + w(v_2^+, v_2^-) \\ w(v_2^-, v_3^+) &\leq w(v_3^+, v_3^-) \\ w(v_3^-, v_1^+) &\leq w(v_1^+, v_1^-) + w(v_1^-, v_2^+) + w(v_2^+, v_2^-) + \\ &\quad w(v_2^-, v_3^+) + w(v_3^+, v_3^-) \end{aligned}$$

we have,

$$\begin{aligned} \triangle'_3 &= w(v_1^-, v_2^+) + w(v_2^-, v_3^+) + w(v_3^-, v_1^+) \\ &\leq \triangle_3 + 2 \cdot (w(v_1^-, v_2^+) + w(v_2^-, v_3^+)) \\ &\leq 2 \cdot \triangle_3 + 3 \cdot w(v_1^-, v_2^+) \leq 3 \cdot \triangle_3 \end{aligned}$$

- Assume that it holds for k-cycles when $k < l$, we prove that it also holds when $k = l$. Suppose that the shortest p-path is (v_1^-, v_2^+), combine (v_1^+, v_1^-), (v_1^-, v_2^+) and (v_2^+, v_2^-) into a single p-path (v_1^+, v_2^-), then we get a k-cycle as $k = l - 1$. As a result, $\triangle'_k \leq (k-1) \cdot (\triangle_k + w(v_1^-, v_2^+)) + w(v_1^-, v_2^+) \leq k \cdot \triangle_k$. □