

VU Research Portal

Heavy Nodes in a Small Neighborhood

Li, Ling; Verbeek, Hilde; Chen, Huiping; Loukides, Grigorios; Gwadera, Robert; Stougie, Leen; Pissis, Solon P.

published in

IEEE Transactions on Knowledge and Data Engineering
2025

DOI (link to publisher)

[10.1109/TKDE.2024.3515875](https://doi.org/10.1109/TKDE.2024.3515875)

document version

Publisher's PDF, also known as Version of record

document license

Article 25fa Dutch Copyright Act

[Link to publication in VU Research Portal](#)

citation for published version (APA)

Li, L., Verbeek, H., Chen, H., Loukides, G., Gwadera, R., Stougie, L., & Pissis, S. P. (2025). Heavy Nodes in a Small Neighborhood: Exact and Peeling Algorithms With Applications. *IEEE Transactions on Knowledge and Data Engineering*, 37(4), 1853-1870. <https://doi.org/10.1109/TKDE.2024.3515875>

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal






Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:

vuresearchportal.ub@vu.nl

Heavy Nodes in a Small Neighborhood: Exact and Peeling Algorithms With Applications

Ling Li , Hilde Verbeek , Huiping Chen , Grigorios Loukides , *Senior Member, IEEE*, Robert Gwadera, Leen Stougie , and Solon P. Pissis 

Abstract—We introduce a weighted and unconstrained variant of the well-known minimum k union problem: Given a bipartite graph $\mathcal{G}(U, V, E)$ with weights for all nodes in V , find a set $S \subseteq V$ such that the ratio between the total weight of the nodes in S and the number of their *distinct* adjacent nodes in U is maximized. Our problem, which we term *Heavy Nodes in a Small Neighborhood* (HNSN), finds applications in marketing, team formation, and money laundering detection. For example, in the latter application, S represents bank account holders who obtain illicit money from some peers of a criminal and route it through their accounts to a target account belonging to the criminal. We prove that HNSN can be solved exactly in polynomial time via linear programming. We also develop several algorithms offering different effectiveness/efficiency trade-offs: an exact algorithm, based on node contraction, graph decomposition, and linear programming, as well as three peeling algorithms. The first peeling algorithm is a near-linear time approximation algorithm with a tight approximation ratio, the second is an iterative algorithm that converges to an optimal solution in a very small number of iterations in practice, and the third is a near-linear time greedy heuristic. In addition, we formalize a money laundering scenario involving multiple target accounts and show how our algorithms can be extended to deal with it. Our experiments on real and synthetic datasets show that our algorithms find (near-)optimal solutions, outperforming a natural baseline, and that they can detect money laundering more effectively and efficiently than two state-of-the-art methods.

Index Terms—Minimum k union, dense subgraph discovery, combinatorial optimization, money laundering detection.

Received 12 March 2024; revised 28 October 2024; accepted 25 November 2024. Date of publication 11 December 2024; date of current version 7 March 2025. The work of Ling Li was supported by a CSC Scholarship. The work of Hilde Verbeek was supported by CWT's Constance van Eeden PhD fellowship. The work of Leen Stougie is supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation under Grant NETWORKS-024.002.003. Recommended for acceptance by T. Weninger. (*Corresponding author: Huiping Chen.*)

Ling Li and Grigorios Loukides are with the Department of Informatics, King's College London, WC2B 4BG London, U.K. (e-mail: ling.2.li@kcl.ac.uk; grigorios.loukides@kcl.ac.uk).

Hilde Verbeek is with the Centrum Wiskunde & Informatica (CWI), 1098 XG Amsterdam, The Netherlands (e-mail: hilde.verbeek@cwi.nl).

Huiping Chen is with the University of Birmingham, B15 2TT Birmingham, U.K. (e-mail: h.chen.13@bham.ac.uk).

Robert Gwadera is with UBS AG, CH-4051 Basel, Switzerland (e-mail: gwadera@bluewin.ch).

Leen Stougie and Solon P. Pissis are with the CWI and the Vrije Universiteit, 1081 HV Amsterdam, The Netherlands (e-mail: leen.stougie@cwi.nl; solon.pissis@cwi.nl).

This article has supplementary downloadable material available at <https://doi.org/10.1109/TKDE.2024.3515875>, provided by the authors.

Digital Object Identifier 10.1109/TKDE.2024.3515875

I. INTRODUCTION

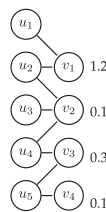
THE minimum k union problem [1], [2] is a well-known combinatorial optimization problem asking to select k sets from a collection of sets that have the minimum union size (i.e., k sets that together contain the fewest distinct elements). It can also be modeled as a small set expansion problem [2], where there is a bipartite graph $\mathcal{G}(U, V, E)$ whose left side U represents elements, right side V represents sets, and there is an edge $(u, v) \in E$ between a node $u \in U$ and a node $v \in V$, if and only if the set corresponding to v contains the element corresponding to u . Then, minimum k union is clearly equivalent to the problem of finding a set $S \subseteq V$ of k nodes, in order to minimize the size of their *neighborhood* $N(S)$ (i.e., the number of *distinct* nodes adjacent to the nodes in S). We introduce a weighted, unconstrained (i.e., S can have any size) variant of this problem, which we term HEAVY NODES IN A SMALL NEIGHBORHOOD (HNSN):

Problem 1 (HNSN): Given a bipartite graph $\mathcal{G}(U, V, E)$, where each $v \in V$ has degree $d(v) > 0$, and a weight function $w : V \rightarrow \mathbb{R}_{\geq 0}$, find a set $S \subseteq V$ of nodes s.t. $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ is maximized, where $N(S) = \{u \mid \exists (u, v) \in E \wedge u \in U, v \in S\}$.

Example 1: The optimal solution of HNSN on the graph below, where the weights of the nodes in V appear on the right, is $S = \{v_1\}$, since $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ is maximized to $\frac{1 \cdot 2}{2}$.

(We assume that $w()$ is evaluated in $O(1)$ time.) The degree requirement ensures that \mathcal{G} has no isolated nodes in V ; these would be added to a solution, as they only increase the objective function. HNSN is motivated by three real-world applications:

A.1 Profitable product set discovery: U represents all *products* for sale, V all previously purchased *bundles* of products, and an edge $(u, v) \in E$ that product u was sold in a bundle v . Each bundle v brought a profit $w(v)$ to a retailer (e.g., the sum of the profit of each product in v [3], or a fraction of that). HNSN outputs a collection of bundles that has the largest ratio between the total profit and the number of distinct products in these bundles. Such bundles can be greatly beneficial to a retailer, as retailers often wish to know a small *set* of products that generate a large profit when sold in specific bundles. This knowledge can be exploited by a retailer to buy such products cheaply in bulk, or transport and store them with lower costs [4].



A.2 Team formation [5]: U represents a set of *workers* and V a set of *jobs*. Each edge $(u, v) \in E$ represents that a worker $u \in U$ has some of the skills required for accomplishing a job $v \in V$. A job v is accomplished when all relevant workers (i.e., the adjacent nodes of v) are hired to do it; this results in a profit $w(v)$ to a company. HNSN outputs a set S of jobs that are performed by a set $N(S)$ of workers and have the largest total profit per hired worker ratio.

A.3 Money laundering detection [6], [7]: Smurfing is a key money laundering technique [8], [9], [10], [111]. It starts with a criminal (detection target) who distributes their illicit money into peers. These peers deposit the money to their bank accounts and perform transactions with other peers who then transfer the money, or a large part of it, into the criminal's bank account. Their goal is to hide the original source of money, so that it appears it came from legitimate sources. HNSN can be used to detect this single-target smurfing attack. U and V correspond to *bank account holders*, and an edge $(u, v) \in E$ represents a transaction performed between $u \in U$ and $v \in V$. The target is represented by a node t , connected to all nodes in V . Note that there are no edges between nodes in V because such edges make it hard for the attack to succeed and easier to track the account holders performing money laundering [8]. The peers who receive money from the criminal are part of U and those who transfer money to the criminal are part of V . The weight $w(v)$ represents how suspicious the money flow through v is (see Section VII). Furthermore, a suspicious peer corresponding to v typically receives money from few peers corresponding to nodes in U [8], [10], as more transactions increase the risk that this peer is caught. Thus, the output of HNSN is a set of peers (corresponding to a subset of V) who are the most suspicious based on their weights and on that they received money from few others (corresponding to a subset of U).

Contributions: We introduce the HNSN problem and design algorithms offering different effectiveness/efficiency trade-offs. Specifically:

1. Unlike minimum k union that is NP-hard to solve exactly [2], we show that HNSN can be solved *exactly in polynomial time*. In particular, we design a Linear Programming (LP) algorithm for HNSN, called LP, and prove that it solves HNSN exactly.

2. We propose ContractDecompose, an exact algorithm for HNSN to construct an optimal solution faster than the LP algorithm when the input graph \mathcal{G} is disconnected, as is often in practice. ContractDecompose is based on two observations we make about any optimal solution of HNSN: (I) the solution is contained in one of the connected components of \mathcal{G} ; and (II) the solution contains either all nodes with the same neighborhood or none of them.

3. We design three *peeling* algorithms¹ for HNSN: (I) an *approximation algorithm* (GAR) which iteratively peels a selected node in U , its neighbors, and the incident edges to these neighbors; (II) an *iterative* algorithm (IP) which peels the entire graph

¹Peeling algorithms [12] typically start from the entire graph and iteratively remove parts of it until it becomes empty. Each remaining part yields a candidate solution and the best among them is returned.

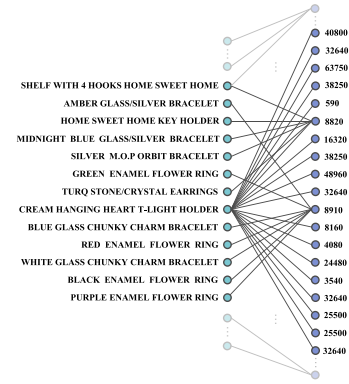


Fig. 1. A solution of our approximation algorithm. Product names (nodes in U) appear on the left and profits of product sets (nodes in V) appear on the right.

TABLE I
SUMMARY OF OUR ALGORITHMS

Algorithm	Type	Comments
LP	Exact	Optimal solutions but slower than other algorithms in general.
CD	Exact	Optimal solutions faster than LP when the graph has multiple connected components and contracted nodes.
IP	Approx.	Converges to an optimal solution after few iterations in practice.
GAR	Approx.	Finds a $\frac{1}{\max_{v \in V} N(\{v\}) }$ approximate solution faster than LP, CD, and IP.
GR	Greedy	No solution quality guarantees but faster than GAR.
FGR	Greedy	Variant of GR that is more efficient but less effective than GR.

\mathcal{G} in each iteration and is based on a non-trivial reduction from Densest Supermodular Subset [13]; and (III) a *heuristic* (GR) which iteratively peels nodes of V , as well as a faster variant of it (FGR). Along the way, we prove that the approximation ratio of the first algorithm is *tight*, that the second algorithm converges to an optimal solution of HNSN, and that the first, third, and fourth algorithms can be implemented in near *linear time*. The example below showcases our first algorithm in application A.1.

Example 2: We applied our approximation algorithm on the *E-Commerce* dataset of Section IX. This dataset is represented by a bipartite graph $\mathcal{G}(U, V, E)$, where U corresponds to products and V to product sets, and $(u, v) \in E$ denotes that a product set v contains product u . Furthermore, each product set v is associated with a profit $w(v)$. We obtained the solution shown in Fig. 1. The solution has a very large profit and its neighborhood is comprised of few semantically similar products which can be bought, transported, or stored with low costs due to economies of scale.

A summary of our algorithms is in Table I. The exact algorithms (LP and CD) are ideal for smaller datasets or when solution quality is critical. The approximation algorithms (IP and GAR) allow dealing with larger data than the exact ones and provide quality guarantees. The greedy heuristics are suitable for large-scale data and perform very well in practice.

4. We examine how smurfing attacks can be detected based on HNSN. Beyond the single-target attack in application A.3, which can be directly tackled by our algorithms, we investigate a multi-target attack, in which there is a third layer in \mathcal{G} containing multiple target accounts. We extend HNSN to a problem for detecting this attack and design adaptations of our algorithms for it.

5. We conducted experiments on 13 real datasets and synthetic ones showing that: (I) our algorithms offer different effectiveness/efficiency trade-offs (our exact algorithms are

reasonably fast, our approximation and iterative algorithms achieve (near)-optimal solutions much faster, and our heuristic trades off a small amount of effectiveness for significantly better efficiency); (II) our algorithms outperform a natural baseline [14], [15] in terms of effectiveness and/or efficiency; (III) ContractDecompose is faster than LP; and (IV) our iterative algorithm converges to an optimal solution in a very small number of iterations. We also show, using a real dataset, that our algorithms can detect money laundering substantially more effectively and efficiently than the two state-of-the-art methods [8], [10] that are closest to our work.

Organization: Section II discusses our LP-based algorithm (LP) as well as its proof of optimality. Section III discusses our ContractDecompose algorithm, which is often faster than LP in practice. Section IV, V, and VI discusses our approximation, iterative, and heuristic algorithm, respectively. Section VII highlights application A.3. Section VIII reviews related work. In Section IX, we present our extensive experimental evaluation. Last, Section X concludes the paper.

A preliminary version *without* Contribution 2, the first two algorithms of Contribution 3, adaptations of the new algorithms and Theorem 11 in Contribution 4, and most of the experiments appeared in the SDM 2023 conference [16].

II. EXACT LP ALGORITHM

LP for HNSN. We assign a binary variable x_i to each node $v_i \in V$ of \mathcal{G} such that $x_i = 1$, if $v_i \in S$ and $x_i = 0$ otherwise. Also, we assign a binary variable y_j to each node $u_j \in U$ such that $y_j = 1$ if $u_j \in N(S)$ (i.e., u_j is adjacent to any node in a solution $S \subseteq V$) and $y_j = 0$ otherwise. Furthermore, we consider an arbitrary ordering of nodes in V and assign a (non-negative) weight $w_i = w(v_i)$ to every node $v_i \in V$. Then, to solve HNSN, we need to solve the following Linear-Fractional Integer Program:

$$\max \left(\sum_{i=1}^{|V|} w_i \cdot x_i \right) / \sum_{j=1}^{|U|} y_j \quad (1a)$$

$$\text{s.t.} \quad \sum_{j=1}^{|U|} y_j > 0 \quad (1b)$$

$$x_i \leq y_j, \quad (u_j, v_i) \in E \quad (1c)$$

$$x_i \in \{0, 1\}, \quad i \in [1, |V|] \quad (1d)$$

$$y_j \in \{0, 1\}, \quad j \in [1, |U|]. \quad (1e)$$

Constraint 1b is added due to the degree requirement $d(v) > 0$, for each $v \in V$, in Problem 1, which implies $|N(S)| > 0$, for any $S \subseteq V$. Constraint 1c is added because, if a node v_i is contained in S , all its adjacent nodes are contained in $N(S)$ and, if v_i is not contained in S , its adjacent nodes may still be contained in $N(S)$. So far, it is not obvious that we can solve the program in (1) in polynomial time.

To solve the program in polynomial time, we transform it as follows. First, we linearize its objective function based on the Charnes-Cooper transformation [17]. That is, we introduce a

variable $y_0 = \frac{1}{\sum_{j=1}^{|U|} y_j} > 0$, which can be defined due to Constraint 1b, and using y_0 we rewrite the program as:

$$\max \quad y_0 \cdot \sum_{i=1}^{|V|} w_i \cdot x_i \quad (2a)$$

$$\text{s.t.} \quad y_0 \cdot \sum_{j=1}^{|U|} y_j = 1 \quad (2b)$$

$$\sum_{j=1}^{|U|} y_j > 0 \quad (2c)$$

$$x_i \leq y_j, \quad (u_j, v_i) \in E \quad (2d)$$

$$x_i \in \{0, 1\}, \quad i \in [1, |V|] \quad (2e)$$

$$y_j \in \{0, 1\}, \quad j \in [1, |U|] \quad (2f)$$

$$y_0 \in \mathbb{R}_{>0}. \quad (2g)$$

Second, we observe that Constraint 2b implies Constraint 2c and that $y_0 > 0$. Thus, we can remove Constraint 2c and require $y_0 \geq 0$. We also relax the integrality requirement for all variables x_i and y_j , writing the program in (2) as the following non-linear program:

$$\max \quad y_0 \cdot \sum_{i=1}^{|V|} w_i \cdot x_i \quad (3a)$$

$$\text{s.t.} \quad y_0 \cdot \sum_{j=1}^{|U|} y_j = 1 \quad (3b)$$

$$x_i \leq y_j, \quad (u_j, v_i) \in E \quad (3c)$$

$$x_i \in [0, 1], \quad i \in [1, |V|] \quad (3d)$$

$$y_j \in [0, 1], \quad j \in [1, |U|] \quad (3e)$$

$$y_0 \geq 0. \quad (3f)$$

Last, by setting $z_i = y_0 \cdot x_i$ and $q_j = y_0 \cdot y_j$ in the program of (3), we get the Linear Program below. We obtain Constraint 4c by multiplying both parts of Constraint 3c by y_0 and substituting with z_i and q_j .

$$\max \quad \sum_{i=1}^{|V|} w_i \cdot z_i \quad (4a)$$

$$\text{s.t.} \quad \sum_{j=1}^{|U|} q_j = 1 \quad (4b)$$

$$z_i \leq q_j, \quad (u_j, v_i) \in E \quad (4c)$$

$$z_i \in [0, 1], \quad i \in [1, |V|] \quad (4d)$$

$$q_j \in [0, 1], \quad j \in [1, |U|]. \quad (4e)$$

LP Optimality and Complexity: While LP relaxations do not generally lead to optimal solutions, we prove that the LP in (4) yields an optimal solution to HNSN, by showing that the value of this LP is lower bounded and upper bounded by the objective value of HNSN. Our proof uses some ideas from [12] but our problem and its LP formulation are quite different.

Lemma 1 (Lower bound): For any set $S \subseteq V$, the value of the program in (4) is at least $\frac{\sum_{v \in S} w(v)}{|N(S)|}$.

Proof: We will show that, for a set $S \subseteq V$, there exists a feasible solution (\bar{z}, \bar{q}) with value $\frac{\sum_{v \in S} w(v)}{|N(S)|}$, where $\bar{z} = \{\bar{z}_i \mid v_i \in V\}$ and $\bar{q} = \{\bar{q}_j \mid u_j \in U\}$. For each node $v_i \in S$, set $\bar{z}_i = \frac{1}{|N(S)|}$. For each node $v_i \in V \setminus S$, set $\bar{z}_i = 0$. For each node $u_j \in N(S)$, set $\bar{q}_j = \frac{1}{|N(S)|}$. For each node $u_j \in U \setminus N(S)$, set $\bar{q}_j = 0$. Hence, $\sum_{j=1}^{|U|} \bar{q}_j = \sum_{j: u_j \in N(S)} \bar{q}_j = |N(S)| \cdot \frac{1}{|N(S)|} = 1$, Constraint 4b is satisfied. Clearly, all other constraints are satisfied too. Thus, (\bar{z}, \bar{q}) is a feasible solution to the LP and as such it has value $\sum_{i=1}^{|V|} w_i \cdot \bar{z}_i$. Furthermore, $\sum_{i=1}^{|V|} w_i \cdot \bar{z}_i = \sum_{i: v_i \in S} w_i \cdot \bar{z}_i = \frac{\sum_{v \in S} w(v)}{|N(S)|}$. The first equality holds by the way we set the \bar{z}_i 's and the second by the way we assigned the w_i 's. Since $\sum_{i=1}^{|V|} w_i \cdot \bar{z}_i = \frac{\sum_{v \in S} w(v)}{|N(S)|}$ and $\sum_{i=1}^{|V|} w_i \cdot z_i$ is maximized by the program in (4), we have $\max \sum_{i=1}^{|V|} w_i \cdot z_i \geq \frac{\sum_{v \in S} w(v)}{|N(S)|}$. \square

Lemma 2 (Upper bound): Given a feasible solution of the program in (4) with value τ , we can construct $S \subseteq V$ such that $\frac{\sum_{v \in S} w(v)}{|N(S)|} \geq \tau$.

Proof: Consider a feasible solution (\bar{z}, \bar{q}) to the program in (4). Let $n^-(v_i) \subseteq U$ be the set of adjacent nodes to node $v_i \in V$ and $n^+(u_j) \subseteq V$ be the set of adjacent nodes to a node $u_j \in U$. Without loss of generality, we assume $\bar{z}_i = \min_{j: u_j \in n^-(v_i)} \bar{q}_j$, for all $i \in [1, |V|]$. This is because Constraint 4c holds, for any $u_j \in U$ that is adjacent to v_i and \bar{z}_i must be maximal due to (4a).

We define: (I) collections of sets S and N indexed by a parameter $r \geq 0$: $S(r) = \{i \mid \bar{z}_i \geq r\}$ and $N(r) = \{j \mid \bar{q}_j \geq r\}$; and (II) a function $F(r) = \sum_{i \in S(r)} w_i$ that outputs the total weight of nodes with indices in $S(r)$.

We show that an index of a node is contained in $S(r)$ if and only if the indexes of its adjacent nodes are contained in $N(r)$. First, we show that if $i \in S(r)$, then each of its corresponding j 's is contained in $N(r)$. Indeed, $i \in S(r)$ implies $\bar{z}_i = \min_{j: u_j \in n^-(v_i)} \bar{q}_j \geq r$. Thus, for every $u_j \in n^-(v_i)$ (i.e., for every adjacent node u_j of v_i), it holds that $\bar{q}_j \geq r$, which implies that each j that corresponds to i is contained in $N(r)$, by the definition of $N(r)$. Then, we show that if each j corresponding to an i is contained in $N(r)$, then $i \in S(r)$. Indeed, if each j corresponding to an i is contained in $N(r)$, then $\bar{q}_j \geq r$ holds, for each such j , by the definition of $N(r)$. Thus, $\min_{j: u_j \in n^-(v_i)} \bar{q}_j \geq r$ also holds. This implies that $\bar{z}_i \geq r$ and $i \in S(r)$, by the $S(r)$ definition.

We will prove that $\int_0^\infty |N(r)| dr = \sum_{j=1}^{|U|} \bar{q}_j$. We define r_j as the largest r resulting in a subset $N(r_j)$ of $N(r)$ with $|U| - \alpha_j$ nodes, where $\alpha_1 = 0$ and $\alpha_j \geq 1$, for $j \in [2, |U|]$ (see Fig. 2(a)). Thus, all nodes in U with $\bar{q}_j \geq r_j$ are contained in $N(r_j)$ and also in every $N(r_{j'})$, $j' \in [1, j]$. Furthermore, $\alpha_{j+1} - \alpha_j$ nodes in U are contained in $N(r_j) \setminus N(r_{j+1})$; these nodes have $\bar{q}_j = r_j$.

If all \bar{q}_j 's are equal (i.e., each $u_j \in U$ has $\bar{q}_j = r_1$), $\int_0^\infty |N(r)| dr = |N(r_1)| = |U| \cdot r_1 = \sum_{j=1}^{|U|} \bar{q}_j$. Otherwise, (5) holds (see also Fig. 2):

$$\int_0^\infty |N(r)| dr = (|U| - \alpha_1) \cdot r_1$$

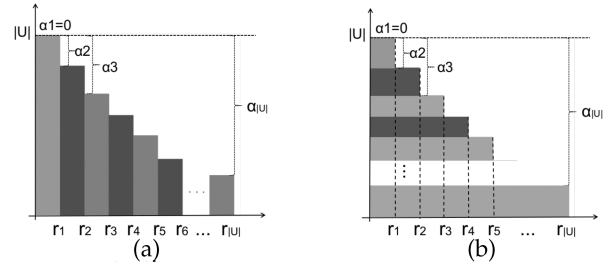


Fig. 2. Illustration of (a) the first and (b) the second equality of (5).

$$\begin{aligned} & + (|U| - \alpha_2) \cdot (r_2 - r_1) + \dots + (|U| - \alpha_{|U|}) \cdot (r_{|U|} - r_{|U|-1}) \\ & = r_1(\alpha_2 - \alpha_1) + r_2(\alpha_3 - \alpha_2) + \dots + r_{|U|}(|U| - \alpha_{|U|}) = \sum_{j=1}^{|U|} \bar{q}_j. \end{aligned} \quad (5)$$

The first equality holds due to the definition of $N(r)$ and r_j 's, and the second one holds trivially (see also Fig. 2(b)). The third equality holds because the $\alpha_{j+1} - \alpha_j$ nodes that are contained in $N(r_j) \setminus N(r_{j+1})$ have equal \bar{q}_j 's, as mentioned above. Thus, each summand in $r_1(\alpha_2 - \alpha_1) + \dots + r_{|U|}(|U| - \alpha_{|U|})$ corresponds to a group of nodes with equal \bar{q}_j 's, and this summand is equal to the sum of the \bar{q}_j 's in the group. Therefore, $r_1(\alpha_2 - \alpha_1) + \dots + r_{|U|}(|U| - \alpha_{|U|}) = \sum_{j=1}^{|U|} \bar{q}_j$.

In addition, $\sum_{j=1}^{|U|} \bar{q}_j = 1$ due to (4b), which holds because (\bar{z}, \bar{q}) is a feasible solution. Thus, we have proved that $\int_0^\infty |N(r)| dr = 1$.

Similarly, we have $\int_0^\infty F(r) dr = \sum_{i=1}^{|V|} w_i \cdot \bar{z}_i$, which is the value of the objective function of the program in (4). Let us denote this value by τ .

We claim that there exists an r such that $\frac{F(r)}{|N(r)|} \geq \tau$. Suppose that such an r does not exist. Then, for every r , it holds that $\frac{F(r)}{|N(r)|} < \tau$ which implies $\int_0^\infty F(r) dr < \tau \cdot \int_0^\infty |N(r)| dr = \tau \cdot 1 = \tau$. However, we have shown that $\int_0^\infty F(r) dr = \tau$, so we have a contradiction.

To find an r such that $\frac{F(r)}{|N(r)|} \geq \tau$, we check all combinatorially distinct sets $N(r)$ by setting $r = \bar{q}_j$ for every $u_j \in U$. Let r' be one r such that $\frac{F(r')}{|N(r')|} \geq \tau$. Then, we construct an $S = \{v_i \in V \mid \bar{z}_i \geq r'\}$ such that $\frac{F(r')}{|N(S)|} = \frac{\sum_{i: v_i \in S} w_i}{|N(S)|} \geq \tau$ and $\frac{\sum_{v \in S} w(v)}{|N(S)|} \geq \tau$. \square

Putting Lemmas 1 and 2 together, we obtain:

Theorem 1 (Optimality and Construction): Let OPT be the value of an optimal solution to the program in (4). Then, the following holds:

$$\max_{S \subseteq V} \frac{\sum_{v \in S} w(v)}{|N(S)|} = \text{OPT}. \quad (6)$$

Further, a set $S \subseteq V$ with maximum $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ can be constructed from an optimal solution to this program.

Proof: Due to Lemma 1, $\text{OPT} \geq \max_{S \subseteq V} \frac{\sum_{v \in S} w(v)}{|N(S)|}$ (consider $S^* = \arg \max_{S \subseteq V} \frac{\sum_{v \in S} w(v)}{|N(S)|}$). Due to Lemma 2,

$\max_{S \subseteq V} \frac{\sum_{v \in S} w(v)}{|N(S)|} \geq \text{OPT}$. Thus, (6) holds. The proof of Lemma 2 gives a construction of a set S that maximizes $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ from the optimal solution to the program in (4). \square

Our algorithm solves the program in (4) and then constructs an optimal solution S . Since this program is linear, its solution can be obtained in $O(n^{2.5})$ time [18], where n is the total number of variables and constraints; see [19] for the state of the art time complexity of linear programming. S can be constructed as in the proof of Lemma 2. However, it is also possible to construct S in $O(|V|)$ time given an optimal solution (z^*, q^*) of the program in (4). Specifically, we identify each $z_i^* \in z^*$ such that $z_i^* > 0$ in $O(|V|)$ time and construct $S = \{v_i \in V \mid z_i^* > 0\}$.

We will prove that $S = \{v_i \in V \mid z_i^* > 0\}$ is an optimal solution to HNSN. Since (z^*, q^*) is optimal for the program in (4), (x^*, y^*) is optimal for the program in (2), where x^* is comprised of each $x_i^* = z_i^*/y_0^*$, $i \in [1, |V|]$, y^* is comprised of each $y_j^* = q_j^*/y_0^*$, $j \in [1, |U|]$, and $y_0^* = \frac{1}{\sum_{j=1}^{|U|} y_j^*}$. Since (x^*, y^*)

is optimal, $y_0^* \cdot \sum_{i=1}^{|V|} w_i \cdot x_i^*$ is maximum, subject to the constraints of the program in (2), which are all satisfied.

For any $i \in [1, |V|]$, $v_i \in S$ implies $z_i^* > 0$. Further, $z_i^* > 0$ implies: (I) $x_i^* = 1$; and (II) $q_j^* > 0$, for each $u_j \in N(S)$, from (4c), which in turn implies $y_j^* = 1$, for each $u_j \in N(S)$. Also, for any $i \in [1, |V|]$, $v_i \notin S$ implies $z_i^* = 0$, which in turn implies $x_i^* = 0$, and $y_j^* = 0$, for each $u_j \notin N(S)$, due to (2a). Thus, for the S we constructed, $y_0^* \cdot \sum_{i=1}^{|V|} w_i \cdot x_i^* = \frac{1}{\sum_{j: u_j \in N(S)} y_j^*}$.

$\sum_{i=1}^{|V|} w_i \cdot x_i^* = \frac{1}{|N(S)|} \cdot \sum_{i=1}^{|V|} w_i \cdot x_i^* = \frac{\sum_{i: v_i \in S} w_i}{|N(S)|}$. Since the constraints of the program in (2) are satisfied, (2c) implies $\sum_{j: u_j \in N(S)} y_j^* = |N(S)| > 0$. Thus, $\frac{\sum_{i: v_i \in S} w_i}{|N(S)|} = \frac{\sum_{v \in S} w(v)}{|N(S)|}$ is maximum subject to $|N(S)| > 0$, and therefore S is an optimal solution.

III. ContractDecompose ALGORITHM

We propose ContractDecompose, an algorithm for solving HNSN exactly based on our LP algorithm. Its benefit is being exact unconditionally *and* efficient when the input graph $\mathcal{G}(U, V, E)$ is comprised of many connected components.

ContractDecompose is based on two observations about an optimal solution to HNSN: (I) it either contains all nodes of V having the same neighborhood, or none of them; and (II) it can be obtained from exactly one of the connected components of \mathcal{G} . We define a connected component as a subset of V in which every pair of not necessarily distinct nodes share a node from U in their neighborhood.² For example, in a graph $\mathcal{G}(U, V, E) = (\{u\}, \{v\}, \{(u, v)\})$ there is one component since the nodes in the pair (v, v) share u .

Observation I is based on the fact that adding into any feasible solution S of HNSN, any node which has the same neighborhood with a node already in S , does *not* change the denominator of the objective function $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ of HNSN but cannot decrease the

²This definition is equivalent to the standard definition of a connected component (as a maximal set of connected nodes in a graph) and simplifies our discussion and notation.

numerator, as all node weights are non-negative. Observation II is more involved; it follows from Theorem 2, whose proof is based on Lemma 3.

Lemma 3: Given the set $\mathcal{C} = \{C_1, \dots, C_k\}$ of connected components of the node set V of the input graph $\mathcal{G}(U, V, E)$ to HNSN, there is some $C_i \in \mathcal{C}$ such that $\frac{\sum_{v \in C_i} w(v)}{|N(C_i)|} \geq \frac{\sum_{v \in \cup_{i \in [k]} C_i} w(v)}{|N(\cup_{i \in [k]} C_i)|}$.

Proof: If $k = 1$, the statement holds trivially. For $k > 1$, we will prove it by contradiction.

For brevity, we define a function $g: 2^V \rightarrow \mathbb{R}_{\geq 0}$ such that $g(C) = \frac{\sum_{v \in C} w(v)}{|N(C)|}$, for any set of nodes $C \subseteq V$. Suppose that $g(C_i) < g(\cup_{i \in [1, k]} C_i)$ for all $i \in [1, k]$. Then, $g(C_i) = \frac{\sum_{v \in C_i} w(v)}{|N(C_i)|} < g(\cup_{i \in [k]} C_i)$ implies that for any $i \in [1, k]$:

$$\sum_{v \in C_i} w(v) < g\left(\bigcup_{i \in [k]} C_i\right) \cdot |N(C_i)|. \quad (7)$$

Furthermore, we have:

$$g\left(\bigcup_{i \in [k]} C_i\right) = \frac{\sum_{v \in C_1} w(v) + \dots + \sum_{v \in C_k} w(v)}{|N(C_1)| + \dots + |N(C_k)|} \quad (8)$$

$$< \frac{g\left(\bigcup_{i \in [k]} C_i\right) \cdot (|N(C_1)| + \dots + |N(C_k)|)}{|N(C_1)| + \dots + |N(C_k)|}. \quad (9)$$

(8) holds because the nodes in each C_i , $i \in [k]$, do not share neighbors in U and (9) is due to (7). However, the right part of (9) is clearly equal to $g\left(\bigcup_{i \in [k]} C_i\right)$, hence we arrive at a contradiction. \square

Theorem 2: An optimal solution to HNSN can be obtained from one of the connected components of the node set V of the input graph $\mathcal{G}(U, V, E)$ to HNSN.

Proof: Consider an optimal solution S^* and the graph $\mathcal{G}'(U', V', E')$ that is induced by the nodes in S^* and $N(S^*)$, i.e., $U' = N(S^*)$, $V' = S^*$, and $E' = \{(u, v) \mid u \in U' \wedge v \in V'\}$. If \mathcal{G}' is connected, the statement clearly holds. Thus, we will henceforth assume that \mathcal{G}' is not connected. We will denote the connected components of \mathcal{G}' by C'_1, \dots, C'_k . For brevity, we define a function $g: 2^V \rightarrow \mathbb{R}_{\geq 0}$ such that $g(C) = \frac{\sum_{v \in C} w(v)}{|N(C)|}$, for any set of nodes $C \subseteq V$. Clearly, $g(S^*) = g(V')$ and since S^* is optimal, $g(V') \geq g(C'_i)$, for any connected component C'_i , $i \in [1, k]$, of \mathcal{G}' . Furthermore, from Lemma 3, $g(V') \leq g(C'_i)$ for some C'_i , $i \in [1, k]$. Let this connected component be C' . Thus, it holds that $g(V') = g(C')$ and combined with $g(S^*) = g(V')$, we obtain $g(S^*) = g(C')$. This implies that an optimal solution to HNSN can be obtained from this C' , which is also a connected component of the node set V of \mathcal{G} . \square

Due to Observation I, ContractDecompose *contracts* every set of nodes in V that have the same neighborhood (i.e., it replaces every set $\{v_1, \dots, v_r\} \subseteq V$ such that $N(\{v_1\}) = \dots = N(\{v_r\})$ with a new node $v \in V$ such that $N(\{v\}) = N(\{v_1\}) = \dots = N(\{v_r\})$) and sets the weight $w(v)$ to $\sum_{i \in [1, r]} w(v_i)$. Due to Observation II, ContractDecompose

applies our exact LP algorithm to each of the connected components C_i , $i \in [1, k]$, of the input graph \mathcal{G} to construct a feasible solution S_i and returns as S the best of these solutions according to the objective function of HNSN, which is an optimal solution to HNSN by Theorem 2.

The pseudocode of ContractDecompose is shown in Algorithm 1. In Lines 1 to 18, it performs the node contraction process. This process starts by constructing two arrays of vectors, \mathcal{A} and \mathcal{B} : $\mathcal{A}[i]$ is a vector of the neighbors of $u_i \in U$ and $\mathcal{B}[j]$ is a vector of the neighbors of $v_j \in V$ (Lines 1 to 7). Then, in Line 8, ContractDecompose sorts \mathcal{B} using Most Significant Digit (MSD) radix sort [20], to efficiently identify groups of nodes in V that have the same neighborhood (i.e., vectors in \mathcal{B} with the same elements). Next, it iterates over the sorted \mathcal{B} and identifies a longest sequence of vectors in \mathcal{B} that have the same elements (Line 12). These vectors appear consecutively in \mathcal{B} , due to sorting, and correspond to a maximal set of nodes to be contracted. Subsequently, ContractDecompose contracts the nodes (Lines 13 to 17) and increases the index i of \mathcal{B} so that the next sequence of vectors is considered (Line 18). After that, ContractDecompose finds the connected components of \mathcal{G}' , the graph after node contraction, and applies LP to each component C_i , $i \in [1, k]$, of this graph, to construct an HNSN solution S_i for C_i . Last, it outputs the best among the constructed solutions (Line 23), which is an optimal solution to HNSN on \mathcal{G} , by Theorem 2.

Algorithm 1: ContractDecompose(\mathcal{G}, w).

```

/* Node contraction */
1  $\mathcal{A} \leftarrow$  array of  $|U|$  empty vectors
2  $\mathcal{B} \leftarrow$  array of  $|V|$  empty vectors
3 for each edge  $(u_i, v_j) \in E$  do
4   | Append  $v_j$  into the vector  $\mathcal{A}[i]$ 
5 for each vector  $\mathcal{A}[i]$  in  $\mathcal{A}$  do
6   | for each element  $v_j$  in the vector  $\mathcal{A}[i]$  do
7     | | Append  $u_i$  into the vector  $\mathcal{B}[j]$ 
8 Sort  $\mathcal{B}$  with MSD radix sort
9  $\mathcal{G}'(U', V', E') \leftarrow \mathcal{G}(U, V, E)$ 
10  $i \leftarrow 0; r \leftarrow 0$ 
11 while  $i < \mathcal{B}.size$  do
12   | Find a longest sequence  $\mathcal{B}[i], \dots, \mathcal{B}[r]$  of consecutive
13   | vectors in  $\mathcal{B}$  that are comprised of the same nodes in  $U$ 
14   | Remove  $v_i, \dots, v_r$  from  $V'$  and all their incident edges
15   | from  $E'$ 
16   | Add a node  $v$  into  $V'$ 
17   |  $w(v) \leftarrow \sum_{j=i}^r w(v_j)$ 
18   | for each element  $u$  in one of the vectors  $\mathcal{B}[i], \dots, \mathcal{B}[r]$  do
19     | | Add an edge  $(u, v)$  into  $E'$ 
20     |  $i \leftarrow r + 1$ 
/* Graph decomposition */
21 Find the connected components  $\{C_1, \dots, C_k\}$  of  $\mathcal{G}'$ 
22 for each  $i \in [1, k]$  do
23   |  $\mathcal{G}_i \leftarrow$  bipartite graph induced by the nodes in  $C_i$  and those
24   | in  $N(C_i)$ 
25   |  $S_i \leftarrow \text{LP}(\mathcal{G}_i, w)$ 
26 return  $S \leftarrow \arg \max_{S_i: i \in [1, k]} \frac{\sum_{v \in S_i} w(v)}{|N(S_i)|}$ 

```

Theorem 3: ContractDecompose can be implemented in $O(\sum_{i \in [1, k]} \mathcal{T}(C_i))$ time, where $\mathcal{T}(C_i)$ is the time needed to apply LP to the connected component C_i , $i \in [1, k]$, of graph \mathcal{G} .

All missing proofs are in Supplemental Material.

IV. GreedyApproximation ALGORITHM

GreedyApproximation is a near-linear time “peeling” approximation algorithm for HNSN. It provides a tight approximation ratio, as we prove in Theorem 6, and its solutions are near-optimal on real-world graphs, as we show in Section IX.

Algorithm 2: Greedy Approximation(\mathcal{G}, w).

```

1  $i \leftarrow 0$ 
2  $R_i \leftarrow U$  // set of remaining nodes
3 while  $R_i \neq \emptyset$  do
4   | Select  $u \in R_i$  s.t.  $\sum_{v \in N_{\mathcal{G}}(\{u\})} w(v)$  is minimum
5   | /* Update  $\mathcal{G}$  */
6   |  $E \leftarrow E \setminus \{(u', v') \mid v' \in N_{\mathcal{G}}(\{u\}) \wedge u' \in N_{\mathcal{G}}(\{v'\})\}$ 
7   |  $V \leftarrow V \setminus N_{\mathcal{G}}(\{u\})$ 
8   |  $U \leftarrow U \setminus \{u \mid u \in U \wedge N_{\mathcal{G}}(\{u\}) = \emptyset\}$ 
9   | /* Update this set of remaining nodes */
10  |  $i \leftarrow i + 1$ 
11  |  $R_i \leftarrow U$ 
12 return  $S \leftarrow \arg \max_{R_i: i \leq |U|} \frac{\sum_{v \in N_{\mathcal{G}}(R_i)} w(v)}{|R_i|}$ 

```

The pseudocode of GreedyApproximation is shown in Algorithm 2; $N_{\mathcal{G}}()$ denotes the neighbors of a subset of nodes of either U or V in $\mathcal{G}(U, V, E)$. The algorithm: (I) iteratively removes from the graph a node $u \in U$ whose neighbors have a minimum total weight (Line 4), as this does not substantially reduce the value of the objective function of HNSN; (II) updates the graph by removing all edges incident to u or to its neighbors, as well as these neighbors and all nodes in U (including u) whose degree has become zero (Lines 5 to 7); (III) updates the set R_i of remaining nodes (Lines 8 and 9); and (IV) outputs a set S comprised of the neighbors of R_i that is the best over all iterations (Line 10).

We show a near-linear time implementation:

Theorem 4: GreedyApproximation can be implemented in $O(|E| \cdot \log |U|)$ -time.

Theorem 5: GreedyApproximation finds an $\frac{1}{\max_{v \in V} |N(\{v\})|}$ -approximate solution to HNSN.

Proof: For any feasible solution $S \subseteq V$, it holds that:

$$\sum_{v \in S} w(v) \geq \frac{\sum_{u \in N(S)} \sum_{v \in N(\{u\})} w(v)}{\max_{v \in S} |N(\{v\})|}. \quad (10)$$

This is because $\sum_{u \in N(S)} \sum_{v \in N(\{u\})} w(v) = \sum_{v \in S} w(v) \cdot |N(\{v\})| \leq \sum_{v \in S} w(v) \cdot \max_{v \in S} |N(\{v\})|$. To see why the first equality holds note that the left-hand side of the equality counts $w(v)$ once for every neighbor of v , while the right-hand side counts it $|N(\{v\})|$ times once. The inequality follows from $|N(\{v\})| \leq \max_{v \in S} |N(\{v\})|$.

Now we prove the following, for any $u^* \in N(S^*)$ where S^* is any optimal solution to HNSN:

$$\sum_{v \in N(\{u^*\})} w(v) \geq \frac{\sum_{v \in S^*} w(v)}{|N(S^*)|}. \quad (11)$$

To see this, consider that we remove each neighbor $v \in N(\{u^*\})$ of any node $u^* \in N(S^*)$. This feasible solution clearly satisfies $\frac{\sum_{v \in S^*} w(v) - \sum_{v \in N(\{u^*\})} w(v)}{|N(S^*)| - 1} \leq \frac{\sum_{v \in S^*} w(v)}{|N(S^*)|}$. Thus,

$(\sum_{v \in S^*} w(v) - \sum_{v \in N(\{u^*\})} w(v)) \cdot |N(S^*)| \leq \sum_{v \in S^*} w(v) \cdot (|N(S^*)| - 1)$, which can be simplified to (11).

Let u^* be the first node from $N(S^*)$ that is removed during an iteration of the algorithm in which $S \supseteq S^*$. Thus, from (10), we have $\sum_{v \in S} w(v) \geq \frac{|N(S)| \sum_{v \in N(\{u^*\})} w(v)}{\max_{v \in S} |N(\{v\})|}$, since u^* is the node whose neighbors have a minimum sum of weights among all nodes in $N(S)$ (i.e., $\sum_{v \in N(\{u\})} w(v) \geq \sum_{v \in N(\{u^*\})} w(v)$). Therefore, we obtain $\frac{\sum_{v \in S} w(v)}{|N(S)|} \geq \frac{\sum_{v \in N(\{u^*\})} w(v)}{\max_{v \in S} |N(\{v\})|}$. From the latter inequality and (11), we obtain $\frac{\sum_{v \in S} w(v)}{|N(S)|} \geq \frac{1}{\max_{v \in V} |N(\{v\})|} \cdot \frac{\sum_{v \in S^*} w(v)}{|N(S^*)|}$.

The approximation ratio is $\frac{1}{\max_{v \in V} |N(\{v\})|}$ because the algorithm outputs the best subset S over all iterations. \square

Theorem 6: The approximation ratio of Greedy Approximation is tight for every integer $\max_{v \in V} |N(\{v\})| \geq 2$.

Proof: We prove this by constructing, given $\Delta_V \geq 2$ and a positive integer k , an instance $(\mathcal{G}(U, V, E), w)$ of HNSN s.t.:

- the maximum degree of the nodes in V is Δ_V ;
- all weights of the nodes in V are 1;
- the optimal solution value is Δ_V ;
- no two nodes in V have the same neighborhood;³
- the worst case solution output by GreedyApproximation has a value that tends to 1, as k goes to infinity.

That is, we show that GreedyApproximation can have a ratio of $\frac{1}{\Delta_V} = \frac{1}{\max_{v \in V} |N(\{v\})|}$ when applied on this \mathcal{G} .

We do this, by constructing a graph \mathcal{G} that is the union of a “sparse” and a “dense” part, which overlap in some $2\Delta_V - 2$ nodes of U . An optimal solution restricted to the sparse part will have a value of $\max_S \frac{\sum_{v \in S} w(v)}{|N(S)|} = 1$ whereas the dense part has a value of Δ_V . Moreover, all nodes in U except the $2\Delta_V - 2$ nodes that are in both parts, will have degree Δ_V , with no U -nodes having a smaller degree.

We describe the two parts of the graph separately; see also Fig. 3. The common part (i.e., the nodes in both the sparse and the dense part) consists of the U -nodes u_1 through $u_{2\Delta_V-2}$.

Sparse part: Let t be some integer such that $t > 2\Delta_V - 2$ (we give an exact value later). The sparse part of the graph consists of t V -nodes and t U -nodes. The latter consist of the common part u_1 through $u_{2\Delta_V-2}$, and $u_{2\Delta_V-1}$ through u_t . Now, for every $i \in [1, t]$, we create a node v_i and, for every $j \in [i, i + \Delta_V - 1]$, we add an edge between v_i and u_j (where $u_j = u_{j-t}$ if $j > t$). Now, this part of the graph looks like a sort of cycle, where every node in either U or V has degree Δ_V . Because there are t nodes in both U and V here, and all weights are 1, the weight/neighborhood ratio of this sparse part is $t/t = 1$.

Dense part: We add nodes u'_1 through u'_k to U . For every $i \in [1, k]$, we add nodes $v'_{i,1}$ through v'_{i,Δ_V} to V and, for every $j \in [1, \Delta_V]$, we add edges between u'_i and $v'_{i,j}$ and between $v'_{i,j}$ and each of u_j through $u_{j+\Delta_V-2}$ (which may include nodes of the common part). Now every node of u'_1, \dots, u'_k has degree Δ_V and every node added to V also has degree Δ_V . In this part of the graph, we added k new nodes to U in addition to the $2\Delta_V - 2$

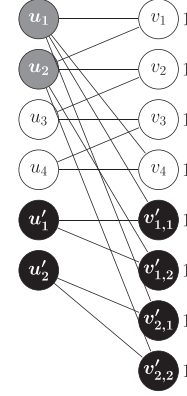


Fig. 3. $\mathcal{G}(U, V, E)$ for $\Delta_V = 2$, $t = 4$, and $k = 2$. The white (black) nodes are contained in the sparse (dense) part of \mathcal{G} , and the gray in both of these parts. The weights of the nodes in V appear on their right.

nodes of the common part. Furthermore, there are $k \cdot \Delta_V$ nodes in V for the dense part. This gives a weight/neighborhood ratio of $\frac{k \cdot \Delta_V}{k + 2\Delta_V - 2}$ for this part, which tends to Δ_V as k goes to infinity.

Putting this together, we have an instance in which every node in U , except u_1 through $u_{2\Delta_V-2}$, has degree Δ_V and all these nodes tie for having the smallest sum of neighboring weights. If one of the nodes in the sparse part of the graph is removed, its neighbors' neighbors will get a smaller degree. Thus, if the first iteration removes a node that is only in the sparse part, the entire sparse part (excluding the common part) will then be removed in the subsequent iterations and we end up with just the dense (and optimal) part after $t - 2\Delta_V + 2$ iterations which, as described before, has a value of Δ_V as k goes to infinity.

For example, in Fig. 3, after removing u_3 and u_4 , the node v_1 is left with the dense part. This gives a ratio of $\frac{k \cdot \Delta_V + \Delta_V - 1}{k + 2\Delta_V - 2}$, which tends to Δ_V as k goes to infinity. Moreover, it can be seen that Δ_V is also an upper bound on the optimal ratio of this instance: if the neighborhood of the solution contains m nodes out of u'_1 through u'_k , then it can contain at most $\Delta_V \cdot m$ V -nodes in the dense part, meaning the ratio must be strictly less than $\frac{\Delta_V \cdot m}{m} = \Delta_V$ (considering this denominator does not include any of the nodes u_1 through $u_{2\Delta_V-2}$, which would also be in the neighborhood). On the other hand, if only nodes unique to the dense part are removed for k iterations in a row, nothing but the sparse part will be left. This case is the worst case for the algorithm: the value of the solution decreases in every iteration, meaning that the first candidate solution (i.e., the entire graph) will be what is returned. There are $t + k$ nodes in U and $t + k \cdot \Delta_V$ nodes in V , so the ratio for the entire graph is $\frac{t + k \cdot \Delta_V}{t + k}$. If we set $t = k^2$, this equals $\frac{k^2 + k \cdot \Delta_V}{k^2 + k}$, which tends to 1 as k goes to infinity. Thus, the approximation ratio tends to $\frac{1}{\Delta_V} = \frac{1}{\max_{v \in V} |N(\{v\})|}$ in the worst-case for this graph. \square

V. ITERATIVE PEELING ALGORITHM VIA DSS

IterativePeeling is an iterative approximation algorithm. In its first iteration, it produces a feasible solution with the same approximation ratio as GreedyApproximation, while in its subsequent iterations it improves the quality of such a solution, until

³Otherwise, the instance might easily be reduced to an equivalent smaller instance for which the ratio is not tight.

it converges to an optimal solution, as we show theoretically. Interestingly, convergence happens after a very small number of iterations in practice (see Section IX).

The crux of IterativePeeling is a highly non-trivial reduction to the Densest Supermodular Subset (DSS) problem [13]. The reduction allows us to use an effective and efficient algorithm for DSS [13] as a subroutine in IterativePeeling.

Reduction from HNSN to DSS : We start by some definitions. A function $f : 2^V \rightarrow \mathbb{R}_{\geq 0}$ is *non-negative* if $f(S) \geq 0$ for any set $S \subseteq V$, *normalized* if $f(\{\}) = 0$, and *supermodular* if $f(X \cup \{x\}) - f(X) \leq f(Y \cup \{x\}) - f(Y)$ for any $X \subseteq Y \subseteq V$ and any $x \in V \setminus Y$. A normalized and non-negative supermodular f is also monotone [13]. The DSS problem is defined as follows:

Definition 1 (DSS [13]): Given a ground set V and a non-negative, normalized, and supermodular function $f : 2^V \rightarrow \mathbb{R}_{\geq 0}$, find $S \subseteq V$ that maximizes $\frac{f(S)}{|S|}$.

We will reduce HNSN to DSS via HNSN_S , an intermediate problem that makes the reduction easier, due to its objective function. The first step of our reduction is an approximation-preserving reduction from HNSN to HNSN_S (Lemma 4) and the second is an approximation-preserving reduction from HNSN_S to DSS (Lemma 6).

For a set of nodes $U' \subseteq U$, let $\tilde{N}(U') = \{v \in V \mid v \in N(U') \wedge N(\{v\}) \subseteq U'\}$, i.e., the set of neighbors of the nodes in U' which have *no neighbor* that is *not* contained in U' . HNSN_S is defined below.

Definition 2 (HNSN_S): Given a bipartite graph $\mathcal{G}(U, V, E)$, where each $v \in V$ has degree $d(v) > 0$, and a weight function $w : V \rightarrow \mathbb{R}_{\geq 0}$, find a set of nodes $U' \subseteq U$ and a set of nodes $\tilde{N}(U') \subseteq N(U')$ such that $\frac{\sum_{v \in \tilde{N}(U')} w(v)}{|U'|}$ is maximized.

Recall that, in HNSN, a subset S of V suffices to determine the quality of a solution, as all neighbors of S are considered; i.e., what determines the quality of a solution is the subgraph induced by the nodes in $S \cup N(S)$. On the other hand, in HNSN_S , a subset U' of U does not suffice to determine the quality of a solution as some neighbors of $N(U')$ may not be considered. Specifically, in HNSN_S , we only consider the neighbors of U' that have no other neighbors except U' (i.e., we consider precisely $\tilde{N}(U')$). This helps the reduction of Lemma 4; S and $N(S)$ in HNSN corresponds to $\tilde{N}(U')$ and U' , respectively, in HNSN_S .

Lemma 4: There is an approximation-preserving reduction from HNSN to HNSN_S . In particular, there is an instance $\mathcal{I}_{\text{HNSN}}$ of HNSN with solution value α if an instance $\mathcal{I}_{\text{HNSN}_S}$ of HNSN_S obtained from the reduction has solution value α .

The function $\sum_{v \in \tilde{N}(U')} w(v)$ in Def. 2 is clearly non-negative and normalized. Below, we prove that it is also supermodular. This is needed in the reduction, as the function f in DSS is non-negative, normalized, and supermodular.

Lemma 5: The function $\sum_{v \in \tilde{N}(U')} w(v)$ in HNSN_S is supermodular.

Proof: Let $f(U') = \sum_{v \in \tilde{N}(U')} w(v)$. It suffices to show that for any subsets U', U'' of U such that $U' \subseteq U'' \subseteq U$ and any

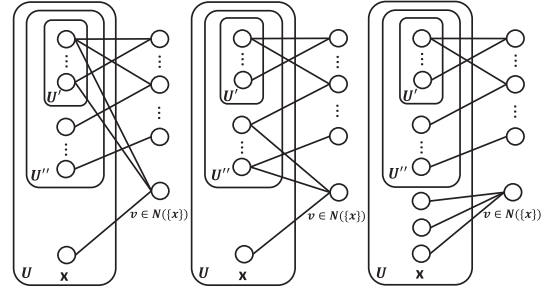


Fig. 4. Instance of Case Ia (left), Ib (center), and Ic (right).

node $x \in U \setminus U''$, it holds that:

$$f(U'' \cup \{x\}) - f(U'') \geq f(U' \cup \{x\}) - f(U').$$

Without loss of generality, we consider a single neighbor $v \in N(\{x\})$. There are three cases about how the nodes in $N(\{v\}) \setminus \{x\}$ are distributed, which are considered below.

Case I: In this case, $N(\{v\}) \setminus \{x\} \neq \emptyset$ and every $u \in N(\{v\}) \setminus \{x\}$ belongs in exactly one of the sets: (a) $U' \cap U''$, (b) $U'' \setminus U'$, or (c) $U \setminus U''$. Fig. 4 illustrates these cases.

(a): For every $u \in N(\{v\}) \setminus \{x\}$ s.t. $u \in U' \cap U''$ (i.e., all neighbors of v other than x are in $U' \cap U''$), we have:

$$f(U'' \cup \{x\}) - f(U'') = f(U' \cup \{x\}) - f(U') = w(v).$$

This is because adding x into U' adds $w(v)$ to $f(U'')$ and to $f(U')$, since $U' \subseteq U''$.

(b): For every $u \in N(\{v\}) \setminus \{x\}$ s.t. $u \in U'' \setminus U'$ (i.e., all neighbors of v other than x are in $U'' \setminus U'$), we have:

$$f(U'' \cup \{x\}) - f(U'') \geq 0 \text{ and } f(U' \cup \{x\}) - f(U') = 0.$$

This is because adding x to U'' adds $w(v)$ to $f(U'')$ (i.e., $f(U'' \cup \{x\}) - f(U'') \geq 0$ as the difference is equal to $w(v)$, which is non-negative) but it does not affect $f(U')$ (i.e., $f(U' \cup \{x\}) - f(U') = 0$).

(c): For every $u \in N(\{v\}) \setminus \{x\}$ s.t. $u \in U \setminus U''$ (i.e., all neighbors of v other than x are in $U \setminus U''$), we have:

$$f(U'' \cup \{x\}) - f(U'') = f(U' \cup \{x\}) - f(U') = 0.$$

This is because adding x to U'' (respectively, to U') does not affect $f(U'')$ (respectively, $f(U')$).

Case II: In this case, $N(\{v\}) \setminus \{x\} \neq \emptyset$ and the nodes in $N(\{v\}) \setminus \{x\}$ belong to at least 2 of the sets $U' \cap U''$, $U'' \setminus U'$, and $U \setminus U''$ (recall that $U' \subseteq U'' \subseteq U$). Since the nodes $N(\{v\}) \setminus \{x\}$ are contained in at least one of $U'' \setminus U'$ or $U \setminus U''$, $f(U' \cup \{x\}) - f(U') = 0$. However, $f(U'' \cup \{x\}) - f(U'') \geq 0$; it is equal to $w(v)$ when all nodes in $N(v) \setminus \{x\}$ are contained in U'' and 0 otherwise.

Case III: In this case, $N(\{v\}) \setminus \{x\} = \emptyset$ (i.e., $N(\{v\}) = x$). Thus, we have: $f(U'' \cup \{x\}) - f(U'') = f(U' \cup \{x\}) - f(U') = w(v)$. This is because adding x to U'' (respectively, to U') adds $w(v)$ to $f(U'')$ and (respectively, to $f(U')$).

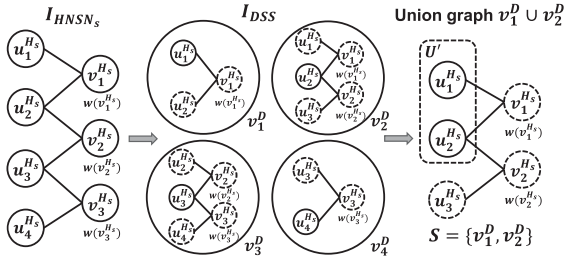


Fig. 5. Constructing \mathcal{I}_{DSS} from a given $\mathcal{I}_{\text{HNSN}_S}$: the subgraph induced by every node $u_i^{H_s}$, $i \in [1, 4]$, its neighbors, and the neighbors of these neighbors, is assigned to node v_i^D in \mathcal{I}_{DSS} . Thus, $V^D = \{v_1^D, \dots, v_4^D\}$. Let $\mathbf{S} = \{v_1^D, v_2^D\} \subseteq V^D$. Then, $f_D(\mathbf{S}) = \sum_{v \in \tilde{N}(\{u_1^{H_s}, u_2^{H_s}\})} w(v) = w(v_1^D)$, since $U' = \{u_1^{H_s}, u_2^{H_s}\}$ and $\tilde{N}(U') = v_1^D$ in the union graph on the right.

Summarizing, $f(U') = \sum_{v \in \tilde{N}(U')} w(v) : 2^U \rightarrow \mathbb{R}_{\geq 0}$ is supermodular, since for every $U' \subseteq U'' \subseteq U$ and $x \in U \setminus U''$, $f(U'' \cup \{x\}) - f(U'') \geq f(U' \cup \{x\}) - f(U')$ holds. \square

Lemma 6: There is an approximation-preserving reduction from HNSN_S to DSS. In particular, there is an instance $\mathcal{I}_{\text{HNSN}_S}$ of HNSN_S with solution value α if an instance \mathcal{I}_{DSS} of DSS obtained from the reduction has solution value α .

Proof: Given any HNSN_S instance $\mathcal{I}_{\text{HNSN}_S} : \mathcal{G}^{H_s}(U^{H_s}, V^{H_s}, E^{H_s})$ and $w : V^{H_s} \rightarrow \mathbb{R}_{\geq 0}$, we create an instance \mathcal{I}_{DSS} of DSS in polynomial time as follows (see also Fig. 5 for an example): (I) As ground set, we define $V^D = \{v_1^D, \dots, v_{|U^{H_s}|}^D\}$ with $v_i^D = \mathcal{G}^{H_s}[C_i]$, $i \in [1, |U^{H_s}|]$, where $C_i = N(\{u_i^{H_s}\}) \cup N(N(\{u_i^{H_s}\}))$ and $\mathcal{G}^{H_s}[C_i]$ is the induced subgraph of \mathcal{G}^{H_s} . Each node $v \in N(\{u_i^{H_s}\})$ of each C_i has a weight $w(v)$ equal to the weight of its corresponding node in V^{H_s} . (II) For any given $\mathbf{S} \subseteq V^D$, we define $f_D(\mathbf{S}) = \sum_{v \in \tilde{N}(U')} w(v)$, where $U' = \{u_i^{H_s} \mid v_i^D \in \mathbf{S}\}$ is a subset of nodes in the union graph $\bigcup_{v_i^D \in \mathbf{S}} v_i^D$. The function $f_D : 2^{V^D} \rightarrow \mathbb{R}_{\geq 0}$ is non-negative, normalized, and supermodular by Lemma 5.

We show that $\mathcal{I}_{\text{HNSN}_S}$ has an optimal solution \mathbf{U} if \mathcal{I}_{DSS} has an optimal solution \mathbf{S} : If $\mathbf{S} \subseteq V^D$ is an optimal solution to \mathcal{I}_{DSS} , then $\frac{f_D(\mathbf{S})}{|\mathbf{S}|} = \frac{\sum_{v \in \tilde{N}(U')} w(v)}{|\mathbf{S}|}$ is maximized, where $U' = \{u_i^{H_s} \mid v_i^D \in \mathbf{S}\}$. Then, in $\mathcal{I}_{\text{HNSN}_S}$, let the solution $\mathbf{U} = U' = \{u_i^{H_s} \mid v_i^D \in \mathbf{S}\}$. Clearly, $|\mathbf{U}| = |\mathbf{S}|$. Furthermore, by the correspondence between the weights of the nodes in V^D and of those in V^{H_s} , $\frac{\sum_{v \in \tilde{N}(\mathbf{U})} w(v)}{|\mathbf{U}|} = \frac{\sum_{v \in \tilde{N}(U')} w(v)}{|\mathbf{S}|}$ is maximized. Thus, \mathbf{U} is an optimal solution to $\mathcal{I}_{\text{HNSN}_S}$.

Since $\frac{\sum_{v \in \tilde{N}(\mathbf{U})} w(v)}{|\mathbf{U}|} = \frac{f_D(\mathbf{S})}{|\mathbf{S}|}$ holds, $\mathcal{I}_{\text{HNSN}_S}$ and \mathcal{I}_{DSS} have the same solution value, and the statement holds.

Based on Lemmas 4 and 6, we obtain: \square

Theorem 7: HNSN can be reduced to DSS with the same solution value in $O(|E|)$ time.

Iterative Peeling Algorithm: Due to Theorem 7, HNSN can be approximated using the Super – Greedy ++ algorithm for DSS from [13]. Theorem 9 states the offered guarantees. Clearly, Theorem 9 also implies that our algorithm converges to an optimal solution for any instance of HNSN.

We start by the following lemma, which is used in the proof of Theorem 9 and also later in our algorithm.

Lemma 7: For every $U' \subseteq U$ s.t. $N(N(U')) \subseteq U'$ and $u \in U'$, $\sum_{v \in \tilde{N}(U')} w(v) - \sum_{v \in \tilde{N}(U' \setminus \{u\})} w(v) = \sum_{v \in N(\{u\})} w(v)$.

We next state the following result from [13], which is used in the proof of Theorem 9, and then prove Theorem 9.

Theorem 8 ([13]): There exists an algorithm that outputs an $(1 - \epsilon)$ -approximation to DSS, for any $\epsilon \in (0, 1)$, after $T = O\left(\frac{\ln(n)}{\text{OPT}_{\text{DSS}} \cdot \epsilon^2} \cdot \max_{v \in V} (f(V) - f(V \setminus v))\right)$ iterations, where OPT_{DSS} is the value of an optimal solution to DSS and $f : 2^V \rightarrow \mathbb{R}_{\geq 0}$ is a non-negative, normalized, supermodular function.

Theorem 9: There exists an algorithm that outputs an $(1 - \epsilon)$ -approximation to HNSN, for any $\epsilon \in (0, 1)$, after $T = O\left(\frac{\ln(|U|)}{\text{OPT} \cdot \epsilon^2} \cdot \max_{u \in U} \sum_{v \in N(\{u\})} w(v)\right)$ iterations, where OPT is the value of an optimal solution to HNSN.

Proof: The algorithm reduces HNSN to DSS via Theorem 7 and uses the algorithm underlying Theorem 8 (i.e., Super – Greedy ++) as subroutine. By Theorem 7 and Theorem 8, it follows that the algorithm obtains an $(1 - \epsilon)$ -approximate solution of HNSN after $T = O\left(\frac{\ln(|U|)}{\text{OPT} \cdot \epsilon^2} \cdot \max_{u \in U} \sum_{v \in N(\{u\})} w(v)\right)$ iterations. This is because in the reduction (see also Lemmas 4, 5, and 6, and Theorem 1.3 of [13]) we set:

$$\begin{aligned} & \max_{v \in V} (f(V) - f(V \setminus v)) \\ &= \max_{u \in U} \left\{ \sum_{v \in \tilde{N}(U)} w(v) - \sum_{v \in \tilde{N}(U \setminus \{u\})} w(v) \right\} \\ &= \max_{u \in U} \left\{ \sum_{v \in N(\{u\})} w(v) \right\} \end{aligned}$$

(the last equality follows from Lemma 7, applied with $U' = U$), $\text{OPT}_{\text{DSS}} = \max_{U' \subseteq U} \frac{f(U')}{|U'|} = \max_{U' \subseteq U} \frac{\sum_{v \in \tilde{N}(U')} w(v)}{|U'|} = \text{OPT}$, $n = |U|$, and ϵ in DSS is equal to ϵ in HNSN. \square

Algorithm 3: IterativePeeling (\mathcal{G}, w, T).

```

1  $L^0 \leftarrow$  Array of 0's of size  $|U|$  // Initial loads
2 for  $t \in [1, T]$  do
3    $j \leftarrow 1$ 
4    $U'_{t,j} \leftarrow U$  // Initial solution for iteration  $t$ 
5   while  $U'_{t,j} \neq \emptyset$  do
6      $u_{t,j} \leftarrow \arg \min_{u \in U'_{t,j}} (L^{t-1}[u] + \sum_{v \in N_{\mathcal{G}}(\{u\})} w(v))$ 
7      $L^t[u_{t,j}] \leftarrow L^{t-1}[u_{t,j}] + \sum_{v \in N_{\mathcal{G}}(\{u_{t,j}\})} w(v)$ 
8     // Load of  $u_{t,j}$ 
9      $E \leftarrow E \setminus \{(u', v') \mid v' \in N_{\mathcal{G}}(\{u_{t,j}\}) \wedge u' \in N_{\mathcal{G}}(\{v'\})\}$ 
10     $V \leftarrow V \setminus N_{\mathcal{G}}(\{u_{t,j}\})$ 
11     $U'_{t,j+1} \leftarrow U'_{t,j} \setminus \{u \in U \mid N_{\mathcal{G}}(\{u\}) = \emptyset\}$ 
12     $j \leftarrow j + 1$ 
13 return  $S \leftarrow \arg \max_{U'_{t,j} : t \in [1, T], j \leq |U|} \frac{\sum_{v \in \tilde{N}(U'_{t,j})} w(v)}{|U'_{t,j}|}$ 

```

We employ Theorem 9 to deal with HNSN; the only difference is that we do not explicitly perform the reduction to DSS but operate directly on the graph \mathcal{G} that is input to HNSN. This leads

to a simpler algorithm, called IterativePeeling. As can be seen in Algorithm 3, IterativePeeling first initializes an array of loads (values that make node selection to consider information from the previous iteration) (Line 1) and then performs T iterations (Lines 2–11). In each iteration, it peels all nodes in U . Each time the loop in Line 5 is executed, it peels the node having a minimum sum of load and weights of its neighbors (Line 6), and it updates the loads, the graph \mathcal{G} , and the current solution (Lines 7 to 10). Last, the solution with the largest objective value among all solutions $U'_{t,j}$ that have been considered is returned (Line 12).

Note that IterativePeeling performs T iterations, each with a different vector of loads L . Also, the initial L^0 is a vector of 0's and the algorithm simply selects a node u with minimum $\sum_{v \in N_{\mathcal{G}}(\{u\})} w(v)$ in Line 6, which is also what GreedyApproximation does. Therefore, after the first iteration, IterativePeeling achieves the same approximation as in Theorem 5. In the next iteration t , the vector of loads L^{t-1} is used to select a node and to compute L^t . Note also that the value of T in Theorem 9 depends on OPT and hence cannot be determined before IterativePeeling runs, so that a desired approximation guarantee is achieved. However, IterativePeeling converges to an optimal solution after a small number of iterations in practice (see Section IX).

IterativePeeling is implemented in $O(T \cdot (|U| + |E| \cdot \log(|U|))) = O(T \cdot |E| \cdot \log(|U|))$ time, where T is the input parameter in Algorithm 3, as it is implemented based on GreedyApproximation.

VI. GREEDY HEURISTIC

Greedy is a near-linear time “peeling” heuristic for HNSN (see Algorithm 4), which: (I) iteratively removes from the input graph a node of V whose removal does not substantially reduce the value of the objective function of HNSN; (II) computes the value of the objective function for the set of remaining nodes after each iteration; (III) finds the best set R of remaining nodes over all iterations; (IV) considers, for each node u of U , all its neighbors with degree 1 as a group and finds the best such group Q ; and (V) outputs as solution the best set among R and Q . Steps I to III are implemented in Lines 2 to 9, while Steps IV and V in Lines 10 to 12.

Let $R_i \subseteq V$ be the set of remaining nodes up until iteration i . Greedy aims to find an R_i with large $\frac{\sum_{v \in R_i} w(v)}{|N(R_i)|}$, by removing from R_{i-1} a node v with small weight $w(v)$, so that $\sum_{v \in R_i} w(v)$ is still large, and with many neighbors, so that $|N(R_i)|$ becomes much smaller. As we show next, if there is a node v in R_i with at least one neighbor with degree 1 in the graph induced by the nodes in R_i and their adjacent nodes $N(R_i)$ (Line 3), one such node v with minimum ratio between $w(v)$ and the number of such neighbors is selected for removal (Line 4). Otherwise, a node v with minimum ratio between $w(v)$ and its degree is selected (Line 6). After each node removal, Greedy memorizes the set of remaining nodes (Line 7). Then, it stores the set R of remaining nodes with a largest objective value over all iterations (Line 9), because the objective function of HNSN is clearly non-monotone (i.e., it may increase or decrease after a node

Algorithm 4: Greedy(\mathcal{G}, w).

```

1  $i \leftarrow 0; R_i \leftarrow V$ 
2 while  $R_i \neq \emptyset$  do
3   if  $\exists v \in R_i$  s.t.  $|N(R_i)| - |N(R_i \setminus \{v\})| > 0$  then
4     | Select  $v \in R_i$  s.t.  $\frac{w(v)}{|N(R_i)| - |N(R_i \setminus \{v\})|}$  is minimum
5   else
6     | Select  $v \in R_i$  s.t.  $\frac{w(v)}{|N(\{v\})|}$  is minimum
7    $R_{i+1} \leftarrow R_i \setminus \{v\}$ 
8    $i \leftarrow i + 1$ 
9  $R \leftarrow \arg \max_{R_j: j \in [0, |V|]} \frac{\sum_{v \in R_j} w(v)}{|N(R_j)|}$ 
10 Select  $u \in U$  s.t.  $\sum_{v \in N(\{u\}): |N(\{v\})|=1} w(v)$  is maximum
11  $Q \leftarrow \{v \mid v \in N(\{u\}) \wedge |N(\{v\})| = 1\}$ 
12 return  $S \leftarrow \arg \max_{S' \in \{R, Q\}} \frac{\sum_{v \in S'} w(v)}{|N(S')|}$ 

```

removal). Next, the algorithm selects the node u of U whose neighbors with degree 1 have the largest total weight and stores these neighbors as Q (Lines 10 to 11). Such solutions are not considered during the previous steps, due to peeling (unless the last R_i is equal to Q). Last, the best among R and Q is returned (Line 12).

Efficient Implementation: A direct implementation of Greedy takes $O(|E| \cdot |V|)$ time because computing all ratios requires traversing \mathcal{G} , which takes $O(|V| + |E|) = O(|E|)$ time per iteration. We show a near-linear time implementation of Greedy in Theorem 10.

Theorem 10: Greedy can be implemented in $O(|E| \cdot \log(|V|))$ time.

The key ideas of the proof (in Supplemental Material) are: (I) To perform the check in Line 3 in $O(|E|)$ time over all iterations. For this, we show that the difference in Line 3 is equal to the number of neighbors of v that have degree 1 in the graph induced by the nodes in R_i and those in $N(R_i)$ and exploit hashing. (II) To perform the selection in Line 4 in $O(|E| \cdot \log(|V|))$ time over all iterations. For this, we identify nodes whose scores must be updated in an iteration, update their ratios, and find a minimum ratio, using a binary search tree and hashing, instead of computing all ratios from scratch to find a minimum. (III) To perform the selection in Line 6 in $O(|V| \cdot \log(|V|))$ time over all iterations using sorting. (IV) To construct R in Line 9 in $O(|E|)$ time, by computing the numerator and denominator of $\frac{\sum_{v \in R_j} w(v)}{|N(R_j)|}$ decrementally.

We also consider a variation of Greedy *without* Lines 3–5, referred to as FastGreedy. It takes $O(|E| + |V| \cdot \log(|V|))$ time and trades effectiveness for efficiency.

VII. MONEY LAUNDERING DETECTION APPLICATION

Money Laundering (ML) is the process of transforming crime profits into legitimate assets [6]. ML involves three steps [6]: (I) *placement*, in which illicit money from a criminal is placed into the financial system; (II) *layering*, in which this money is separated from its source through layering financial transactions that aim to elude detection; and (III) *integration*, in which the money, or a fraction of it, is returned to the criminal from what seem to be legitimate sources.

Smurfing [8], [9], [10], [11] is a key layering technique for attacking anti-money laundering systems. It involves a criminal who distributes their illicit money to peers. Then, the peers perform financial transactions, in which the money flows from their accounts to the accounts of other peers and eventually to the criminal's target account(s). The peers are called smurfs.

Single-target Attack: This attack was discussed in Section I and can be detected by solving HNSN. A node t that represents the target's account and its incident edges play no role in HNSN. Thus, t and these edges are omitted from the input to HNSN. We focus on graphs with no edges among nodes in V , as they are generally more suspicious (i.e., edges between nodes in V do not benefit the attackers, as discussed in Introduction) [8], [21].

Our suspiciousness measure is the objective function $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ of HNSN, with $w(v) = \frac{o_v}{i_v + b_v}$, for each node $v \in V$; o_v is the amount of money sent by v to t , i_v is the total amount of money v receives from its adjacent nodes in U , and b_v is the account balance of v before any money transfers take place. We consider the account balance, as it is not necessarily negligible (e.g., normal accounts may have a large balance) [11]. We assume $i_v + b_v > 0$; e.g., banks require a non-zero balance in order to maintain an account.

If a solution S of HNSN has large $\frac{\sum_{v \in S} w(v)}{|N(S)|}$, the subgraph induced by S and $N(S)$ models a highly suspicious money flow to t . This captures that: (I) Smurfs receive money from few others [8], [10]. This is because a criminal usually trusts a small number of peers (nodes in U) to distribute their money to, as a larger number increases the risk of being caught. (II) Smurfs often aim not to leave much money in their accounts, as their accounts may be frozen [9], [10].

Multi-target Attack: The attack involves multiple targets, representing multiple criminals or a criminal with multiple accounts. The input graph is a tripartite graph $\mathcal{G}(U, V, W, E)$, where W represents users receiving money from others in V . The degree of each node in W is in $[1, |V|]$. The targets correspond to a subset of W .

Our suspiciousness measure is $\frac{\sum_{v \in S} w(v)}{|N(S)| + |M(S)|}$, where $N(S)$ and each $w(v)$ is as before, and $M(S) = \{w \mid \exists (v, w) \in E \wedge v \in S, w \in W\}$ is the subset of nodes in W that receive money from the nodes in $S \subseteq V$. A set S with small $|M(S)|$ is preferred, as the number of targets is typically small (e.g., up to 8 for real ML flows detected in [11]). Our measure is fundamentally different from that of [10] and offers three benefits (see Supplemental Material for details): (I) It can be used to compare subgraphs with nodes transferring different amounts of money. (II) It considers the balance b_v , which is important as a larger b_v implies a higher account retention risk and thus that an account is less likely fraudulent. (III) It allows distinguishing between subgraphs with the same number of nodes and total ingoing and outgoing amounts but different topologies.

Detecting the Multi-target Attack: We extend HNSN for the multi-target case, as follows:

Problem 2 (Multi-target HNSN (MHNSN)): Given a tripartite graph $\mathcal{G}(U, V, W, E)$, where each $v \in V$ has at least one adjacent node in U and at least one adjacent node in W , and a weight function $w : V \rightarrow \mathbb{R}_{\geq 0}$, find a set $S \subseteq$

V of nodes such that $\frac{\sum_{v \in S} w(v)}{|N(S)| + |M(S)|}$ is maximized, where $N(S) = \{u \mid \exists (u, v) \in E \wedge u \in U, v \in S\}$ and $M(S) = \{w \mid \exists (v, w) \in E \wedge v \in S, w \in W\}$.

Each $v \in V$ must have at least one adjacent node in U and at least one adjacent node in W , as we are interested in detecting money flows from nodes in U to target nodes in W through v . This ensures that at least one node in U and at least one node in W is connected to a node in S and hence $|N(S)| > 0$ and $|M(S)| > 0$. Thus, at least one smurf in U sends money to another smurf in S that then sends money to nodes in W . As in the single-target case, $w(v) = \frac{o_v}{i_v + b_v}$, for each $v \in V$.

The LP algorithm for HNSN can easily be adapted to solve MHNSN exactly in polynomial time. Greedy and FastGreedy can also be easily adapted to tackle MHNSN. The details of these algorithms are provided in Supplemental Material. We also show an approximation-preserving reduction (Theorem 11) from MHNSN to HNSN. This has two important implications: (I) An $O(|E| \log(|U| + |W|))$ -time $\frac{1}{\max_{v \in V} (|N(\{v\})| + |M(\{v\})|)}$ -approximation algorithm for MHNSN can be obtained via GreedyApproximation. (II) An $(1 - \epsilon)$ -approximation algorithm to MHNSN can be obtained via IterativePeeling, for any $\epsilon \in (0, 1)$.

Theorem 11: There is an approximation-preserving reduction from MHNSN to HNSN. In particular, there is an instance $\mathcal{I}_{\text{MHNSN}}$ of MHNSN with solution value α if an instance $\mathcal{I}_{\text{HNSN}}$ of HNSN obtained from the reduction has solution value α .

Note that we do not perform the reduction explicitly. We simply modify GreedyApproximation, or IterativePeeling, so that it iteratively peels a node $v \in U \cup W$ with minimum $\sum_{v \in n(u)} w(v)$ (along with all its adjacent nodes and the edges of these nodes), as nodes in U or W are treated the same.

VIII. RELATED WORK

Related Problems: HNSN is a weighted and unconstrained variant of the minimum k union problem; the latter has been studied theoretically in [1], [2]. As discussed in Section I, minimum k union can be formulated as a small set expansion problem. Alternatively, it can be formulated as a problem asking for k hyperedges of a hypergraph with the minimum size of their union, for a given integer k [2]. HNSN can be viewed as a weighted and unconstrained version of this formulation, where each $u \in U$ corresponds to a hypernode, each $v \in V$ to a hyperedge with a weight $w(v)$, and the neighbors of v are the hypernodes of the hyperedge corresponding to v . HNSN and minimum k union are related to dense subgraph problems (see [22], for a survey). Two well-known such problems are the densest subgraph problem [12], [23], which asks for a subgraph with maximum average degree, and the densest k -subgraph problem [24], which asks for a subgraph of k nodes with maximum average degree, for a given integer k . The equivalent of these problems in hypergraphs are the densest subhypergraph problem [25] and the densest k -subhypergraph problem [1], [2]. Both are well studied [2], [25], [26], [27].

Related Applications: Profitable product set discovery falls into the area of utility mining [28]. This area mainly focuses on identifying patterns with sufficiently high utility, while HNSN

seeks to identify sets having a largest ratio between utility and the number of distinct items contained in them, as such sets can be beneficial to a retailer [4].

Team formation can be formulated as a subgraph detection problem on a bipartite graph, and there are recent approaches that adopt this formulation [5], [29], [30]. However, these approaches detect fundamentally different types of subgraphs than our approach. In particular, they consider community search problems on bipartite graphs based on an adaptation of the notion of k -core [31].

Money laundering detection [6] can be performed using rule-based or machine-learning approaches (see [7], for a survey). Our work is relevant to approaches for detecting smurfing attacks from graph data [8], [9], [10], [11], [32], [33], [34], [35]. The approaches of [11], [32] output all subgraphs that are isomorphic to certain graph patterns while our approach outputs the most suspicious subgraph, without restricting its topology to that of specific patterns. The approach of [9] considers a streaming setting and tracks statistics for each account over time to identify suspicious accounts. The approach of [33] is applied to a sequence of directed, edge-weighted graphs having a common node set and aims to classify each node as suspicious or not. It utilizes a supervised approach taking into account various statistics about financial transactions. The approach of [34] has the same goal as that of [33] but is applied to an undirected, edge-weighted graph and follows a semi-supervised approach. The approach of [35] is applied to a directed, edge-weighted, temporal graph and aims to: (I) detect a dense temporal subgraph with high value in measures capturing structural, temporal, and monetary information; and (II) identify the flow of money laundering funds. For task I, it proposes a peeling algorithm; and for task II it uses a maximum flow algorithm. Different from the above approaches, we use an undirected bipartite graph as input (the direction of edges is implicit; always from nodes in U to nodes in V). Also, we aim to detect smurfing attacks (which involve transactions between the peers of a criminal) and thus our algorithms output a connected subgraph of the input graph. Classifying nodes as suspicious or not, as the approaches of [34] and [33] do, is interesting but unrelated to the HNSN problem, and dealing with temporal aspects in our problem is left for future work. Due to their different input data and/or outputs, none of the above approaches is an alternative to our work.

The closest works to ours based on the type of data they are applied to and the attacks that they consider are [10] and [8]. The work of [10] proposes a suspiciousness measure, which has some weaknesses, addressed by our measure (see Section VII), and a greedy algorithm, called FLOWSCOPE. The work of [8] proposes AA-SMURF, a heuristic for detecting a specific single-target smurfing attack in which $|U| = 1$ and $|V| \geq 3$.

IX. EXPERIMENTAL EVALUATION

A. Data and Setup

Datasets: We conducted experiments using 13 real-world datasets, as well as using synthetic datasets (see Table II and Supplemental Material for details).

TABLE II
DATASETS CHARACTERISTICS (k AND c DENOTE THE NUMBER OF CONNECTED COMPONENTS AND THE CONTRACTED NODE PERCENTAGE, RESPECTIVELY; IN SYN_c , $c \in \{10\%, 20\%, 30\%, 40\%, 50\%\}$)

Datasets	$ U $	$ V $	$ E $	k
Foodmart (FM)	1,559	4,141	18,319	1
E-commerce (EC)	3,468	14,975	174,354	12
Liquor (LI)	4,026	52,131	410,609	165
Fruithut (FR)	1,265	181,970	652,773	4
YooChoose (YC)	107,276	234,300	507,266	22,033
Kosarak (KS)	41,270	990,002	8,019,015	271
Connecticut (CN)	458	394,707	1,127,525	117
Digg (DI)	12,471	872,622	22,624,727	13
NotreDame (ND)	127,823	383,640	1,470,404	3,142
IMDB (IM)	303,617	896,302	3,782,463	7,885
NBA Shot (NBA)	129	1,603	13,726	1
ACM Citation (ACM)	751,407	739,969	2,265,837	1
Czech Financial Dataset (CFD)	60,606	1,496	138,256	1,496
AML _H	35,583	31,491	106,514	6,271
AML _L	65,117	42,840	224,991	7,834
SYN	200,000	200,000	80,000,000	1
SYN _k	100,000	150,000	50,000,000	{1,2,5,10,25}
SYN _c	10,000	10,000	80,000	1

The first six datasets in Table II were obtained from <https://bit.ly/3S5y0de>. These datasets contain utilities (profits) of items (nodes in U) which are summed up to obtain the transaction (node $v \in V$) utility, following [3]. Thus, these datasets model the profitable product set discovery application (see A.1 in Section I), where a transaction corresponds to a bundle of products. The next four datasets in Table II were obtained from <https://sparse.tamu.edu/>. These datasets model unweighted bipartite graphs, and we used unary weights for them.

For the team formation application (see A.2 in Section I), we used the NBA and ACM datasets in Table II, obtained from <https://bit.ly/3Yu6M5Q> and <https://bit.ly/3WzWbEc>, respectively. NBA represents LA Lakers players who have scored in at least one regular-season NBA match over two decades (nodes in U) and the matches involving these players (nodes in V). An edge connects a player (node $u \in U$) to a match (node $v \in V$) if the player scored in this match. The weight of a node $v \in V$ is the total points scored by the players in this specific match. In ACM, each node in U represents an author who has co-authored at least one paper and each node in V corresponds to a paper co-authored by at least one author represented by a node in U . An edge connects an author (node $u \in U$) to a paper (node $v \in V$) if the author co-authored the paper. The weight of a node v is the number of citations the paper has received.

For the money laundering detection application (see A.3 in Section I), we utilized three datasets: the Czech Financial Dataset (CFD), which contains anonymous money transfers from a Czech bank, and two anti-money laundering datasets with ground truth, AML_H and AML_L. CFD has been used in [9], [10] and is available from <https://bit.ly/2OzDm2R>. CFD has been pre-processed by keeping transactions from and to other banks, creating a tripartite graph $\mathcal{G}(U, V, W, E)$ (see Section VII). AML_H and AML_L are obtained from <https://bit.ly/4c8q170>, and they have been widely used in recent anti-money laundering research [32], [36], [37]. In AML_H, 0.63% of the transactions (edges) are labeled as fraudulent, compared to 0.36% in AML_L.

We constructed three types of synthetic datasets: SYN, SYN_k, and SYN_c. Each SYN dataset models a bipartite graph that

has one connected component and no nodes that can be contracted. The edges in each SYN dataset are formed between randomly selected nodes. The degrees of the nodes in V are in $[1, 50]$, and they are sampled from a Zipfian distribution with parameter 0.1; such power-law degree distributions are often encountered in real graphs. The weight of every node v in V is set to $w(v) = r \cdot |N(\{v\})|$, where r is chosen uniformly at random from $[0.5, 100]$. We constructed SYN datasets with varying number of U and V nodes and varying number of edges; specifically $|U|$ and $|V|$ are up to 200,000 each and $|E|$ up to 80 million.

Each SYN_k dataset models a bipartite graph with $k \in \{1, 2, 5, 10, 25\}$ connected components. All SYN_k datasets have the same $|U|$, $|V|$, $|E|$ and no nodes that can be contracted. All components in these datasets are constructed as in SYN datasets and their $|U|$, $|V|$, and $|E|$ are the same.

Each SYN_c dataset models a bipartite graph with a percentage $c \in \{10\%, 20\%, 30\%, 40\%, 50\%\}$ of nodes that can be contracted. All SYN_c datasets have the same $|U|$, $|V|$, and $|E|$ and one connected component. Each SYN_c dataset is constructed in two steps. First, for the given c and a fixed $r = 10$, we construct a SYN dataset with $|U| = 10\,000$; $|V| = 10\,000 - c \cdot 10\,000$; and $|E| = 80\,000 - c \cdot 10\,000 \cdot r$. Then, we add $c \cdot 10\,000$ nodes in V with the same r randomly selected neighbors (and thus $c \cdot 10\,000 \cdot r$ new edges in E). This ensures that all $c \cdot 10\,000$ nodes will be contracted. Last, we set each weight $w(v)$, $v \in V$, as in the SYN dataset.

Setup: In the applications of profitable product set discovery and team formation (A.1 and A.2 in Section I), we evaluated our algorithms, ContractDecompose (CD), IterativePeeling (IP), GreedyApproximation (GAR), Greedy (GR), and FastGreedy (FGR), by comparing them to GreedRatio [14], [15] (GRR), a state-of-the-art method for minimizing a ratio of non-negative monotone submodular functions. As in [14], we ensured that there is at least one $v \in V$ with $w(v) > 0$ and applied GRR to try to minimize $\frac{|N(S)|}{\sum_{v \in S} w(v)}$, which is equivalent to maximizing the objective function of HNSN. This is because: (I) $|N(S)|$ is non-negative, monotone (as $0 < |N(S)| \leq |N(T)|$, for any $S \subseteq T \subseteq V$), and submodular (as a coverage function [38]); and (II) $\sum_{v \in S} w(v)$ is non-negative monotone (as $w(v) \in \mathbb{R}_{\geq 0}$) and submodular (as it is clearly modular [38]). GRR takes $O(|E| \cdot |V|)$ time and cannot be sped up by lazy evaluation [39], which is applicable for minimizing the ratio of a modular to a submodular function [14]. In addition, we compared our algorithms to FlowScope (FL) [10] and AA-SMURF [8] in the context of money laundering detection (A.3 in Section I). FL takes $O(|E| \cdot \log(|V|))$ time when applied to a tripartite graph as in our case. On the other hand, AA-SMURF takes $O(|E| \cdot (|U| + |V| + |W|) + (|U| \cdot |W|)^2)$ time in the worst case; recall that W represents the users who receive money from other users in V . We did not compare our algorithms to [9], [11], [32], [33], [34], [35], as they deal with different problems and/or are applied to different settings (see Section VIII).

We used T (the maximum number of iterations) equal to 10 in IP, unless specified otherwise, and measured the effectiveness of the solution S^i of IP right after iteration i using the ratio

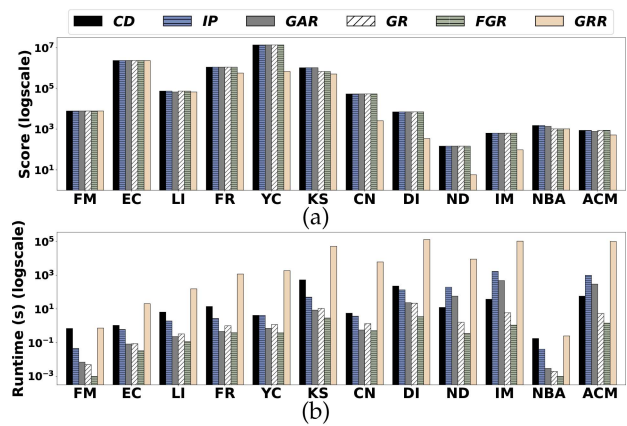


Fig. 6. (a) Score $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ for all algorithms on real datasets. (b) Runtime (s) on real datasets.

$\frac{\sum_{v \in S^i} w(v)}{|N(S^i)|} / \text{OPT}$, where OPT is the value of an optimal solution. We refer to this ratio as *Accuracy* at iteration i and report its mean and sample standard deviation when IP is applied to a set of random subgraphs.

All experiments ran on an AMD EPYC 7702 CPU with 512 GB RAM. We implemented our algorithms in C++ and used Gurobi 9.1.2 in LP and CD. The implementation of FL and AA-SMURF was obtained from <https://bit.ly/3LHigea> and <https://bit.ly/3yFUQn5>, respectively. See <https://bit.ly/3ymM9hG> for our code and datasets.

B. Effectiveness and Efficiency on Real Data

We examined the effectiveness and efficiency of our algorithms⁴ in the context of the profitable product set discovery (A.1) and team formation (A.2) applications using the first 12 real datasets of Table II.

The results evaluating the effectiveness of all HNSN algorithms (i.e., $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ for a solution S of HNSN) are in Fig. 6(a). Note that: (I) CD always found an optimal solution, as expected. (II) IP always found an optimal solution within $T = 10$ iterations. (III) GAR found an optimal solution in all datasets except LI, NBA, and ACM. In these datasets, it achieved solutions that were within 90.6%, 91.5%, and 89.1% of the optimal, respectively. This is encouraging, as its (tight) approximation ratio of $\frac{1}{\max_{v \in V} |N(\{v\})|}$ implies less effectiveness in the worst case (e.g., the smallest ratio over all datasets was $\frac{1}{3420} = 0.00029$ and corresponds to DI but GAR found an optimal solution when applied to DI). (IV) Both GR and its variant, FGR, found near-optimal solutions that are within 94.4% of the optimal on average, over all the tested datasets. The results of the experiments of Fig. 6(a) in terms of runtime are in Fig. 6(b). As can be seen, all our algorithms substantially outperformed GRR. For example, our slowest algorithm, CD, was *more than two orders of magnitude*

⁴We omit the results for LP, as it achieved the same solutions with CD but less efficiently (LP was still much faster than GRR).

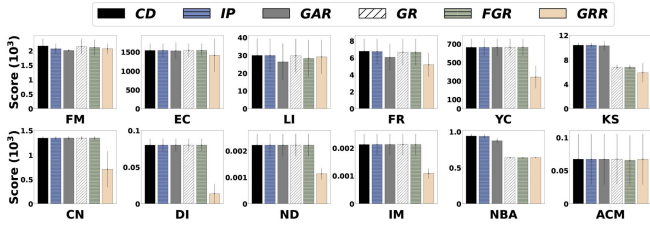


Fig. 7. Mean and sample standard deviation of score $\frac{\sum_{w \in S} w(v)}{|N(S)|}$ over random subgraphs created from real datasets.

faster than GRR on average and up to more than three orders of magnitude faster. Note that: (I) CD performed reasonably well; its runtime was comparable to that of IP. (II) GAR was 3.5 times faster on average than IP. (III) GR was 16.6 times faster on average than GAR. (IV) FGR was the fastest algorithm, outperforming GR by 4.9 times on average. The results show that our exact algorithm CD is practical, and they align with the time complexity analyses of our approximation and heuristic algorithms; these algorithms take near-linear time, whereas GRR takes $O(|E| \cdot |V|)$ time.

C. Effectiveness and Efficiency on Synthetic Data

To thoroughly evaluate effectiveness, we applied all algorithms on 500 small graphs, each constructed by sampling a number of records (a record corresponds to a node in V) uniformly at random from a dataset. We performed this process on all datasets of Section IX-B, sampling 10000 records from EC, LI, YC, KS, CN, and DI, and 1,000 from all other datasets. The results in Fig. 7 are analogous to those in Fig. 6(a); our algorithms found optimal (CD) or near-optimal solutions (the solutions of IP, GAR, GR, and FGR were within 99.54%, 96.80%, 94.15%, and 93.40% of the optimal, respectively). Notably, the order of effectiveness of these algorithms in this experiment is the exact reverse of their order of efficiency. On the other hand, the solutions of GRR were on average 36% worse than the optimal. For example, our worst-performing (but most efficient) algorithm, FGR, was 76.8% more effective than GRR on average. Furthermore, the difference between FGR and GRR on all datasets except LI was statistically significant at p -value < 0.005 , which was computed using Welch's test, while for LI the difference was not statistically significant (p -value > 0.005). In the latter dataset, GAR was 3% worse than GRR, but it was more effective by 96% on average over all the datasets.

In addition, we measured runtime using subgraphs of SYN with varying $|U|$, $|V|$, or $|E|$ (see Fig. 8). Again, all our algorithms outperformed GRR in terms of efficiency. Specifically, CD, IP, GAR, GR and FGR were, on average, faster than GRR by 3.2, 40.7, 294.8, 503.4 and 10237.5 times, respectively. That is, the order of effectiveness of these algorithms was the exact reverse of their order of efficiency. Notably, our faster algorithms, GR and FGR, needed less than 2 minutes and 3 seconds, respectively, to process the largest SYN dataset, which contains 80 million edges, while GRR needed 4.6 hours. The results of Fig. 8 are in line with the time complexity analyses.

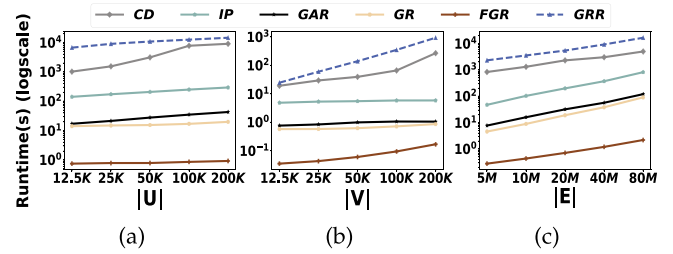


Fig. 8. Runtime (s) on SYN and varying: (a) $|U|$ with $|V| = 100 K$ and $|E| = 20 M$; (b) $|V|$ with $|U| = 50 K$ and $|E| = 20 M$; and (c) $|E|$ with $|U| = 50 K$ and $|V| = 100 K$.

TABLE III
COMPARISON BETWEEN LP AND CD ON REAL DATASETS

Dataset	% of Contracted Nodes c	# of Connected Components k	# of Merged Components	Runtime of LP (s)	Runtime of CD (s)	Efficiency Improvement
FM	1.16%	1	1	0.79	0.70	11.66%
EC	7.12%	12	10	1.15	1.06	7.58%
LI	5.86%	165	134	6.64	6.51	1.90%
FR	42.4%	4	4	361.93	14.00	96.13%
YC	52.19%	22,033	12,489	5.29	4.17	21.19%
KS	38.71%	271	267	20.02	5.59	42.98%
CN	83.33%	117	17	927.33	528.72	72.08%
DI	40.07%	13	10	478.81	227.88	52.41%
ND	48.87%	3,142	1,438	20.91	12.32	41.08%
IM	49.45%	7,885	3,881	62.16	37.77	39.23%
NBA	26.08%	1	1	0.51	0.18	65.02%
ACM	18.39%	1	1	396.13	57.41	85.51%

All algorithms except CD scale linearly with $|E|$, and logarithmically with either $|U|$ or $|V|$. Note that the runtime of GR and GRR increased slightly with $|U|$ due to data structures operating on U whose terms in the complexity formulas are absorbed. Also, note that IP was 10 times slower than GAR, as we used $T = 10$.

From Sections IX-B and IX-C, it is clear that our exact algorithms are reasonably fast (faster than GRR, which performs worse), our approximation algorithms find near-optimal solutions much faster than the exact algorithms, and our heuristics are slightly less effective but much faster than our approximation algorithms.

D. Efficiency Improvement From LP to CD

LP vs CD on Real Data. Table III shows that CD is faster than LP by 44.73% on average and by up to 96.13%. As expected from our time complexity analysis (see Section III), the improvement is larger when $\sum_{i \in [1, k]} \mathcal{T}(C_i)$ is smaller than \mathcal{T} . That is, CD becomes faster when: (I) many nodes in V are contracted; and (II) after the node contraction there are many connected components in the graph. However, invoking the Linear Programming solver incurs some overhead which does not pay off in the case of too small connected components. Therefore, in our implementation, we merged components with fewer than 100 nodes in their V part into a single component iteratively. Each iteration stops when the new component contains 500 or more nodes in its V part. The largest efficiency improvement (96.13%) was observed in FR, where 42.4% of the nodes were contracted and there were only 4 connected components after node contraction, and the smallest in LI (1.9%), where only 5.86% of nodes were contracted and there were still 134 connected components after merging the small connected components.

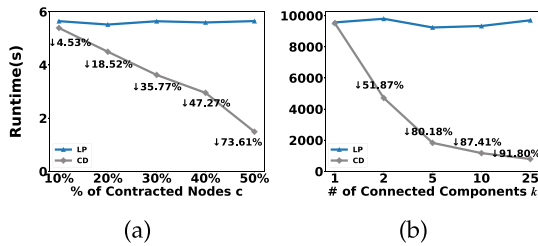


Fig. 9. Runtime (s) of LP and CD and varying: (a) % of Contracted Nodes on SYN_c ; and (b) # of Connected Components k on SYN_k .

LP vs CD on SYN_c and SYN_k . We also examined the impact of node contraction and of graph decomposition *separately* on the efficiency of CD using the SYN_c and SYN_k datasets. That is, we either omit Lines 1 to 18 from Algorithm 1 and use \mathcal{G} , or replace Lines 19 to 23 with “**return** $S \leftarrow LP(\mathcal{G}', w)$ ”.

Fig. 9(a) shows the runtime of LP and CD on SYN_c , for varying percentage of contracted nodes c . As c increases, the difference between CD and LP increases, since the LP algorithm in Line 22 of Algorithm 1 runs on a smaller (single-component) graph in terms of V and E . For example, when 50% of nodes are contracted, CD was 73.61% faster than LP.

Fig. 9(b) shows the runtime of LP and CD on SYN_k , for a graph with $|U| = 100,000$, $|V| = 150,000$, and $|E| = 50,000,000$, and varying number of connected components k . As k increases, the difference in runtime between CD and LP increases, since the connected components become smaller in terms of $|U|$, $|V|$, and $|E|$ and thus the total time of the LP algorithm invocations in CD (Algorithm 1) is smaller. For example, when $k = 2$, CD was 51.89% faster than LP and when $k = 25$, it was 91.80% faster. The marginal gain in efficiency decreases as k increases, since the solver is invoked more often, which incurs an overhead. To reduce such overheads, one can execute Lines 20 to 22 in parallel.

E. Convergence of IP

We next evaluated the Accuracy (see Section IX-A) of IP using the real-world datasets of Section IX-B. IP found an optimal solution *right after the first iteration* in the case of 9 out of 12 datasets. In the LI (respectively, NBA and ACM) dataset, it found a solution within 90.6% (respectively, 91.45% and 89.05%) of the optimal right after the first iteration and an optimal solution right after the third (respectively, third and second) iteration.

We also examined the effectiveness of IP using the random subgraphs sampled from the 12 real datasets used in the experiments of Fig. 7. IP found an optimal solution *right after the first iteration* in all the tested subgraphs of YC, CN, DI, and ND. The results for the subgraphs of the remaining datasets are shown in Fig. 10. IP found an optimal solution right after the fourth iteration in the case of EC and IM, a solution that is within 99.4% of the optimal in the case of LI, FR, KS, NBA and ACM, and a solution within 96.1% of the optimal in the case of FM.

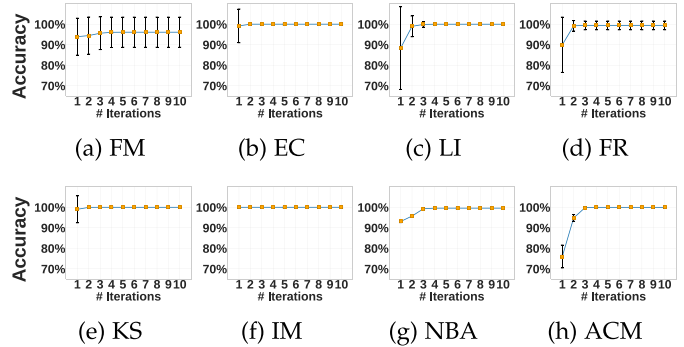


Fig. 10. Convergence of IP with $T = 10$ on random subgraphs created from real datasets.

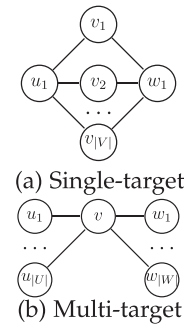


Fig. 11. Types of injected patterns.

F. Money Laundering Detection

We examined the effectiveness of our algorithms for money laundering detection (A.3) using the CFD dataset, which has no ground truth, and the AML_H and AML_L datasets, which have ground truth. IP was configured with $T = 30$.

CFD Dataset: We first detected the most suspicious subgraph from CFD using each of our algorithms for HNSN. All our algorithms found the same solution; a subgraph with 21 nodes (12 source, 1 middle, and 8 target) and small balance before and also after the bank transactions (3,120 and 19,672, respectively). FL detected a much larger subgraph with 122 nodes (87 source, 1 middle, and 34 target) and much larger balance before and also after the transactions (15,087 and 92,581, respectively). The smaller size and balance of the subgraph detected by our algorithms suggest that it is more suspicious [10], [11] than the one detected by FL. Also, this subgraph is 3.22 times more suspicious than that of FL, according to our measure. AA-SMURF did not produce a solution, as the CFD dataset does not have a subgraph that can be detected by this algorithm (see Section VIII).

Since CFD has no ground truth, we followed the evaluation methodology of [8], [9], [10]. That is, we injected fraudulent patterns in CFD and examined whether or not they can be detected. Specifically, we injected two types of patterns that are well-known to correspond to real-world smurfing attacks [8], [11], [32], [40]; see Fig. 11. In the single-target setting, we compared our algorithms to both FL and AA-SMURF. In the

TABLE IV
PERCENTAGE OF DETECTED FRAUDULENT PATTERNS BY OUR METHODS AND BY FL [10] FOR DIFFERENT MEAN VALUES

Mean (μ)	0.7	0.75	0.8	0.85	0.9
Our methods	100%	100%	100%	100%	100%
FL	18%	31%	49%	68%	66%

TABLE V
COMPARISON BETWEEN OUR ALGORITHMS, FL, AND AA-SMURF ON THE AML_H DATASET IN TERMS OF EFFECTIVENESS, RUNTIME, AND SPACE EFFICIENCY

Dataset	AML_H			
	Measurement	Fraudulent Ratio of Edges	Runtime (s)	Memory Usage (GB)
Our methods	$\frac{\sum_{v \in S} w(v)}{ N(S) + M(S) }$	12 / 12	4.81	0.09
FL	0.06	0 / 4	8.24	0.22
AA-SMURF	0.43	6 / 10	13387.16	265.06

TABLE VI
COMPARISON BETWEEN OUR ALGORITHMS, FL, AND AA-SMURF ON THE AML_L DATASET IN TERMS OF EFFECTIVENESS, RUNTIME, AND SPACE EFFICIENCY

Dataset	AML_L			
	Measurement	Fraudulent Ratio of Edges	Runtime (s)	Memory Usage (GB)
Our methods	$\frac{\sum_{v \in S} w(v)}{ N(S) + M(S) }$	12 / 12	10.55	0.23
FL	0.50	2 / 2	17.25	0.34
AA-SMURF	-	-	-	-

The '-'s denote that AA-SMURF could not run (out of memory error) in our machine that has 512 GB RAM.

multi-target setting, we only compared our algorithms with FL because AA-SMURF cannot detect multi-target patterns (see Section VIII). We generated over 1,000 different patterns. To favor FL, we used balance $b_v = 0$, for each $v \in V$.

For the single-target patterns, we used $|V| \in [1, 18]$ and sampled each weight $w(v) \in (0, 1]$ from a normal distribution $\mathcal{N}(\mu, \sigma = 0.1)$. We tried all combinations of $|V| \in \{1, 2, 4, 6, \dots, 18\}$ and $\mu \in \{0.5, 0.55, 0.6, \dots, 1\}$. All our algorithms for HNSN detected *all* injected patterns, whereas FL did not detect *any* injected pattern when $|V| \geq 8$ and AA-SMURF did not detect *any* injected patterns when $|V| = 1$ or $|V| = 2$, as by design it only detects single-source, single-target patterns with $|V| \geq 3$.

For the multi-target patterns, we applied the adaptations of our methods to deal with MHNSN (see Section VII) and modified GAR to output the best among the solution of GAR and a solution comprised of the best node in V , to account for the fact that several nodes in V in CFD are good candidates due to their very small neighborhood and large weight. This modification did not change the approximation ratio of GAR and had a negligible impact on its runtime. We considered two configurations for the patterns. In the first, we tried all combinations of $|U|, |W| \in \{1, 2, \dots, 6\}$ and $w(v) \in \{0.5, 0.55, \dots, 1\}$. Each node $v \in V$ had the same $w(v)$. Again, all our algorithms detected all injected patterns, whereas FL did not detect *any* injected pattern when $w(v) < 0.8$. In the second, we sampled $w(v) \in (0, 1]$ from a normal distribution $\mathcal{N}(\mu, \sigma = 0.1)$ with $\mu \in \{0.7, 0.75, \dots, 0.9\}$. For each mean value μ , we constructed 100 patterns with $|U|$ and $|W|$ selected uniformly at random from $[1, 6]$. Note from Table IV that all our methods detected *all* injected patterns, while FL detected between 18% and 66% of them. GAR without the modification outperformed FL; in the experiment of Table IV its scores were 39%, 60%, 71%, 96%, and 95%.

In sum, our methods detected a more suspicious graph than FL from the real dataset CFD, while AA-SMURF did not detect a suspicious subgraph. Our methods also detected injected patterns more accurately than FL and are applicable to the multi-target setting unlike AA-SMURF. Last, our methods were many times faster than FL, and also faster than AA-SMURF with the exception of IP (see Supplemental Material for details).

AML_H and AML_L Datasets: We compared our algorithms against FL and AA-SMURF using the labeled AML_H and AML_L datasets; see Tables IV and V. Since these datasets have ground truth, we did not inject fraudulent patterns into them. Interestingly, in both datasets, our algorithms detected suspicious patterns (all edges were labeled fraudulent) with one source node, six middle nodes, and one target node (i.e., of the same type as that in Fig. 11(a)). This observation confirms that the patterns used in our experiments with the CFD dataset are realistic and is in line with prior works (e.g., [8], [11], [32], [40]), which state that this type of patterns often correspond to money laundering activities. FL detected a subgraph with no fraudulent edges in the AML_H dataset, while, in the AML_L dataset, it detected a subgraph that does not correspond to smurfing attacks (i.e., a solution with 1 source, 1 middle, and 1 target node and 2 edges). AA-SMURF detected a subgraph with 6 out of 10 fraudulent edges in AML_H , while it could not run on AML_L (due to lack of memory, in a machine with 512 GB RAM).

Our algorithms were also more efficient than FL (by 41.63% in AML_H and by 38.8% in AML_L) and 3 orders of magnitude more efficient than AA-SMURF in AML_H . Last, our algorithms required less memory than FL (by 59.1% in AML_H and 32.3% in AML_L) and three orders of magnitude less memory than AA-SMURF.

X. CONCLUSION

We introduced HNSN, a weighted and unconstrained variant of the well-known minimum k union problem. We showed that HNSN can be solved exactly in polynomial time via linear programming and proposed several algorithms for it: an exact algorithm based on node contraction, graph decomposition and linear programming; and three peeling algorithms offering different effectiveness/efficiency trade-offs. Our experiments showed that our algorithms find optimal or near-optimal solutions, outperforming a natural baseline, and that they can detect money laundering much more effectively and efficiently than the two state-of-the-art methods that are the closest to our work. The ability to define the general combinatorial problem HNSN and solve it via exact and approximation algorithms that are effective in different applications is mostly because its input contains a bipartite graph. However, it should be clear from Section VIII that more general settings are also useful in money laundering detection. For example, a general graph would allow modeling longer paths between the peers of a criminal, while temporal graphs would capture temporal characteristics. It is thus interesting to study HNSN-like problems for money laundering detection in such settings.

REFERENCES

- [1] E. Chlamtác, M. Dinitz, and Y. Makarychev, “Minimizing the union: Tight approximations for small set bipartite vertex expansion,” in *Proc. 28th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2017, pp. 881–899.
- [2] E. Chlamtác, M. Dinitz, C. Konrad, G. Kortsarz, and G. Rabanca, “The densest K-subhypergraph problem,” *SIAM J. Discrete Math.*, vol. 32, no. 2, pp. 1458–1477, 2018.
- [3] V. S. Tseng, B.-E. Shie, C.-W. Wu, and P. S. Yu, “Efficient algorithms for mining high utility itemsets from transactional databases,” *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 8, pp. 1772–1786, Aug. 2013.
- [4] A. Payne, M. Christopher, M. Clark, and H. Peck, *Relationship Marketing for Competitive Advantage*. London, U.K.: Butterworth-Heinemann, 1995.
- [5] K. Wang, W. Zhang, X. Lin, Y. Zhang, L. Qin, and Y. Zhang, “Efficient and effective community search on large-scale bipartite graphs,” in *Proc. Int. Conf. Data Eng.*, 2021, pp. 85–96, doi: [10.1109/ICDE51399.2021.00015](https://doi.org/10.1109/ICDE51399.2021.00015).
- [6] C. S. Chaudhari, *A Guide to Risk Based Internal Audit System in Banks*. Chennai, India: Notion Press, Inc., 2017.
- [7] Z. Chen, L. D. Khoa, E. N. Teoh, A. Nazir, E. K. Karuppiah, and K. S. Lam, “Machine learning techniques for anti-money laundering (AML) solutions in suspicious transaction detection: A review,” *Knowl. Inf. Syst.*, vol. 57, no. 2, pp. 245–285, 2018, doi: [10.1007/s10115-017-1144-z](https://doi.org/10.1007/s10115-017-1144-z).
- [8] M. Lee et al., “AutoAudit: Mining accounting and time-evolving graphs,” in *Proc. IEEE Int. Conf. Big Data*, 2020, pp. 950–956, doi: [10.1109/Big-Data50022.2020.9378346](https://doi.org/10.1109/Big-Data50022.2020.9378346).
- [9] X. Sun et al., “Monlad: Money laundering agents detection in transaction streams,” in *Proc. ACM Int. Conf. Web Search Data Mining*, 2022, pp. 976–986, doi: [10.1145/3488560.3498418](https://doi.org/10.1145/3488560.3498418).
- [10] X. Li et al., “FlowScope: Spotting money laundering based on graphs,” in *Proc. Conf. Assoc. Adv. Artif. Intell.*, 2020, pp. 4731–4738.
- [11] M. Starnini et al., “Smurf-based anti-money laundering in time-evolving transaction networks,” in *Proc. Eur. Conf. Mach. Learn.*, 2021, pp. 171–186, doi: [10.1007/978-3-030-86514-6_11](https://doi.org/10.1007/978-3-030-86514-6_11).
- [12] M. Charikar, “Greedy approximation algorithms for finding dense components in a graph,” in *Proc. Approximation Algorithms Combinatorial Optim.*, Berlin, Heidelberg: Springer-Verlag, 2000, pp. 84–95.
- [13] C. Chekuri, K. Quanrud, and M. R. Torres, “Densest subgraph: Supermodularity, iterative peeling, and flow,” in *Proc. 28th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2022, pp. 1531–1555, doi: [10.1137/1.9781611977073.64](https://doi.org/10.1137/1.9781611977073.64).
- [14] W. Bai, R. K. Iyer, K. Wei, and J. A. Bilmes, “Algorithms for optimizing the ratio of submodular functions,” in *Proc. Int. Conf. Mach. Learn. Appl.*, 2016, pp. 2751–2759. [Online]. Available: <http://proceedings.mlr.press/v48/baib16.html>
- [15] C. Qian, J. Shi, Y. Yu, K. Tang, and Z. Zhou, “Optimizing ratio of monotone set functions,” in *Proc. Int. Joint Conf. Artif. Intell.*, 2017, pp. 2606–2612, doi: [10.24963/ijcai.2017/363](https://doi.org/10.24963/ijcai.2017/363).
- [16] H. Chen, G. Loukides, R. Gwadera, and S. P. Pissis, “Heavy nodes in a small neighborhood: Algorithms and applications,” in *Proc. SIAM Int. Conf. Data Mining*, 2023, pp. 307–315, doi: [10.1137/1.9781611977653.ch35](https://doi.org/10.1137/1.9781611977653.ch35).
- [17] A. Charnes and W. W. Cooper, “Programming with linear fractional functionals,” *Nav. Res. Logistics Quart.*, vol. 9, no. 3/4, pp. 181–186, 1962. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800090303>
- [18] P. M. Vaidya, “Speeding-up linear programming using fast matrix multiplication (extended abstract),” in *Proc. 30th Annu. Symp. Found. Comput. Sci.*, 1989, pp. 332–337, doi: [10.1109/SFCS.1989.63499](https://doi.org/10.1109/SFCS.1989.63499).
- [19] S. Jiang, Z. Song, O. Weinstein, and H. Zhang, “A faster algorithm for solving general LPs,” in *Proc. Conf. Symp. Theory Comput.*, New York, NY, USA, 2021, pp. 823–832, doi: [10.1145/3406325.3451058](https://doi.org/10.1145/3406325.3451058).
- [20] R. Paige and R. E. Tarjan, “Three partition refinement algorithms,” *SIAM J. Comput.*, vol. 16, no. 6, pp. 973–989, 1987, doi: [10.1137/0216062](https://doi.org/10.1137/0216062).
- [21] M. Levi and P. Reuter, “Money laundering,” *Crime Justice*, vol. 34, pp. 289–375, 2006.
- [22] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal, *A Survey of Algorithms for Dense Subgraph Discovery*. Berlin, Germany: Springer, 2010, pp. 303–336, doi: [10.1007/978-1-4419-6045-0_10](https://doi.org/10.1007/978-1-4419-6045-0_10).
- [23] E. Harb, K. Quanrud, and C. Chekuri, “Faster and scalable algorithms for densest subgraph and decomposition,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2022, pp. 26966–26979.
- [24] U. Feige, G. Kortsarz, and D. Peleg, “The dense K-subgraph problem,” *Algorithmica*, vol. 29, no. 3, pp. 410–421, 2001, doi: [10.1007/s004530010050](https://doi.org/10.1007/s004530010050).
- [25] D. J. Huang and A. B. Kahng, “When clusters meet partitions: New density-based methods for circuit decomposition,” in *Proc. Eur. Des. Test Conf.*, 1995, pp. 60–64, doi: [10.1109/EDTC.1995.470419](https://doi.org/10.1109/EDTC.1995.470419).
- [26] S. K. Bera, S. Bhattacharya, J. Choudhari, and P. Ghosh, “A new dynamic algorithm for densest subhypergraphs,” in *Proc. Int. Conf. World Wide Web*, 2022, pp. 1093–1103, doi: [10.1145/3485447.3512158](https://doi.org/10.1145/3485447.3512158).
- [27] C. E. Tsourakakis, “The K-clique densest subgraph problem,” in *Proc. Int. Conf. World Wide Web*, 2015, pp. 1122–1132, doi: [10.1145/2736277.2741098](https://doi.org/10.1145/2736277.2741098).
- [28] W. Gan, J. C.-W. Lin, P. Fournier-Viger, H.-C. Chao, V. S. Tseng, and P. S. Yu, “A survey of utility-oriented pattern mining,” *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 4, pp. 1306–1327, Apr. 2021.
- [29] K. Zhou, J. Xin, J. Chen, X. Zhang, B. Wang, and Z. Wang, “Effective and efficient community search with size constraint on bipartite graphs,” *Inf. Sci.*, vol. 647, 2023, Art. no. 119511.
- [30] Y. Zhang, K. Wang, W. Zhang, W. Ni, and X. Lin, “Size-bounded community search over large bipartite graphs,” in *Proc. Annu. Int. Conf. Extending Database Technol.*, 2024, pp. 320–331.
- [31] D. Ding, H. Li, Z. Huang, and N. Mamoulis, “Efficient fault-tolerant group recommendation using alpha-beta-core,” in *Proc. Conf. Inf. Knowl. Manage.*, 2017, pp. 2047–2050.
- [32] B. Egressy, L. Von Niederhäusern, J. Blanuša, E. Altman, R. Wattenhofer, and K. Atasu, “Provably powerful graph neural networks for directed multigraphs,” in *Proc. Conf. Assoc. Adv. Artif. Intell.*, 2024, pp. 11838–11846.
- [33] X. Luo, X. Han, W. Zuo, Z. Xu, Z. Wang, and X. Wu, “A dynamic transaction pattern aggregation neural network for money laundering detection,” in *Proc. IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun.*, 2022, pp. 818–826.
- [34] X. Luo, X. Han, W. Zuo, X. Wu, and W. Liu, “MLaD 2: A semi-supervised money laundering detection framework based on decoupling training,” *IEEE Trans. Inf. Forensics Secur.*, vol. 19, pp. 4518–4533, 2024.
- [35] D. Lin, J. Wu, Y. Yu, Q. Fu, Z. Zheng, and C. Yang, “DenseFlow: Spotting cryptocurrency money laundering in ethereum transaction graphs,” in *Proc. Int. Conf. World Wide Web*, 2024, pp. 4429–4438.
- [36] B. Oztas, D. Cetinkaya, F. Adedoyin, M. Budka, H. Dogan, and G. Aksu, “Enhancing anti-money laundering: Development of a synthetic transaction monitoring dataset,” in *Proc. IEEE Int. Conf. E-Bus. Eng.*, 2023, pp. 47–54.
- [37] J. Blanuša et al., “Graph feature preprocessor: Real-time extraction of subgraph-based features from transaction graphs,” 2024, *arXiv:2402.08593*.
- [38] A. Krause and D. Golovin, “Submodular function maximization,” *Tractability*, vol. 3, pp. 71–104, 2013.
- [39] M. Minoux, “Accelerated greedy algorithms for maximizing submodular set functions,” in *Proc. Optim. Techn.: Proc. 8th IFIP Conf. Optim. Techn.*, Berlin, Heidelberg, 1978, pp. 234–243.
- [40] E. Altman, J. Blanuša, L. Von Niederhäusern, B. Egressy, A. Anghel, and K. Atasu, “Realistic synthetic financial transactions for anti-money laundering models,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2024, pp. 29851–29874.

Ling Li is currently working toward the PhD degree in the Department of Informatics at King’s College London. Her research interests are graph mining algorithms and applications.

Hilde Verbeek is currently working toward the PhD degree in the Networks & Optimization Group at CWI. Her research interests are string and graph algorithms.

Huiping Chen is an assistant professor with the School of Computer Science, University of Birmingham. Her research interests are in data mining and data privacy.

Leen Stougie is a senior researcher with CWI and a full professor with the Vrije Universiteit, both in Amsterdam. His research focuses on combinatorial optimization.

Grigorios Loukides (Senior Member, IEEE) is an associate professor with King's College London. His research focuses on data mining.

Robert Gwadera is a senior data engineer with the UBS AG Switzerland specializing in research and development of data processing solutions for financial services.

Solon P. Pissis is a senior researcher with CWI and an associate professor with the Vrije Universiteit, both in Amsterdam. His research focuses on theory of algorithms and their application in data mining.